



3<sup>a</sup> edición

# Gráficos por computadora con OpenGL

[www.librosite.net/hearn](http://www.librosite.net/hearn)

Donald Hearn  
M. Pauline Baker

PEARSON  
Prentice  
Hall



# Gráficos por computadora con OpenGL



# Gráficos por computadora con OpenGL

**Tercera edición**

**DONALD HEARN**

**M. PAULINE BAKER**

*Indiana University - Purdue University*

**Traducción**

*Vuelapluma*



Madrid • México • Santa Fe de Bogotá • Buenos Aires • Caracas • Lima  
Montevideo • San Juan • San José • Santiago • São Paulo • White Plains •



**Datos de catalogación bibliográfica**

**GRÁFICOS POR COMPUTADORA CON OPENGL  
DONALD HEARN; M. PAULINE BAKER**

**Pearson Educación S.A., Madrid, 2006**

**ISBN-10: 84-205-3980-5**

**ISBN-13: 978-84-832-2708-4**

**Materia: Informática, 681.3**

**Formato: 195 x 250 mm.**

**Páginas: 918**

Todos los derechos reservados.

Queda prohibida, salvo excepción prevista en la Ley, cualquier forma de reproducción, distribución, comunicación pública y transformación de esta obra sin contar con autorización de los titulares de propiedad intelectual. La infracción de los derechos mencionados puede ser constitutiva de delito contra la propiedad intelectual (*arts. 270 y sgts. Código Penal*).

**DERECHOS RESERVADOS**

© 2006 por PEARSON EDUCACIÓN S.A.

Ribera del Loira, 28

28042 Madrid

Gráficos por computadora con OpenGL

DONALD HEARN; M. PAULINE BAKER

ISBN-10: 84-205-3980-5

ISBN-13: 978-84-205-3980-5

Depósito Legal:

Authorized translation from the English language edition, entitled COMPUTER GRAPHICS WITH OPENGL, 3rd Edition by HEARN, DONALD; BAKER, M. PAULINE, published by Pearson Education, Inc, publishing as Prentice Hall, Copyright © 2004

Equipo editorial

Editor: Miguel Martín-Romo

Técnico editorial: Marta Caicoya

Equipo de producción

Director: José A. Clares

Técnico: María Alvear

Diseño de cubierta: Equipo de diseño de Pearson Educación S.A.

Impreso por:

IMPRESO EN ESPAÑA - PRINTED IN SPAIN

Este libro ha sido impreso con papel y tintas ecológicos

A nuestra gente

*Dwight, Rose, Jay y Millie*

# Contenido

Prefacio	xix		
<b>1 Introducción a los gráficos por computadora</b>	<b>2</b>	♦ Ratones	59
1.1 Gráficos y diagramas	3	♦ <i>Trackballs y spaceballs</i>	60
1.2 Diseño asistido por computadora	5	♦ <i>Joysticks</i>	61
1.3 Entornos de realidad virtual	10	♦ Guantes de datos	61
1.4 Visualización de datos	12	♦ Digitalizadores	62
1.5 Educación y formación	19	♦ Escáneres de imagen	64
1.6 Arte por computadora	23	♦ Paneles tactiles	64
1.7 Entretenimiento	28	♦ Lapiceros ópticos	66
1.8 Procesamiento de imágenes	31	♦ Sistemas de voz	67
1.9 Interfaz gráfica de usuario	32	2.5 Dispositivos de copia impresa	67
1.10 Resumen	33	2.6 Redes gráficas	69
Referencias	33	2.7 Gráficos en Internet	69
2.5	2.8 Software gráfico	70	
♦ Representaciones con coordenadas	70	♦ Funciones gráficas	72
♦ Estándares de software	73	♦ Otros paquetes gráficos	74
♦ Introducción a OpenGL	74	2.9 Introducción a OpenGL	74
♦ Sintaxis básica de OpenGL	74	♦ Bibliotecas relacionadas	75
♦ Archivos de cabecera	76	♦ Gestión de la ventana de visualización	76
♦ empleando GLUT	76	♦ Un programa OpenGL completo	78
♦ Resumen	81	2.10 Resumen	81
Referencias	82	Resumen	81
Ejercicios	82	Referencias	82
<b>2 Introducción a los sistemas gráficos</b>	<b>34</b>		
2.1 Dispositivos de visualización de vídeo	35		
♦ Tubos de refresco de rayos catódicos	35		
♦ Pantallas por barrido de líneas	38		
♦ Pantallas de barrido aleatorio	41		
♦ Monitores TRC de color	42		
♦ Pantallas planas	44		
♦ Dispositivos de visualización tridimensional	47		
♦ Sistemas estereoscópicos y de realidad virtual	47		
2.2 Sistema de barrido de líneas	50		
♦ Controlador de vídeo	51		
♦ Procesador de pantalla de líneas de barrido	53		
2.3 Estaciones de trabajo gráficas y sistemas de visualización	54		
2.4 Dispositivos de entrada	58		
♦ Teclados, cajas de botones y botones de selección	58		
<b>3 Primitivas gráficas</b>	<b>84</b>		
3.1 Sistemas de coordenadas de referencia	86		
♦ Coordenadas de pantalla	86		
♦ Especificaciones absolutas y relativas de coordenadas	87		
3.2 Especificación de un sistema bidimensional de referencia universal en OpenGL	88		

3.3	Funciones de punto en OpenGL	88	3.16	Funciones OpenGL de relleno de áreas poligonales	139
3.4	Funciones OpenGL para líneas	91	3.17	Matrices de vértices OpenGL	145
3.5	Algoritmos de dibujo de líneas	92	3.18	Primitivas de matrices de píxeles	148
♦	Ecuaciones de las líneas	92	3.19	Funciones OpenGL para matrices de píxeles	148
♦	Algoritmo DDA	94	♦	Función de mapa de bits de OpenGL	148
♦	Algoritmo de Bresenham para dibujo de líneas	96	♦	Función OpenGL para mapas de píxeles	150
♦	Visualización de polilíneas	100	♦	Operaciones OpenGL de manipulación de búferes	151
3.6	Algoritmos paralelos de dibujo de líneas	100	3.20	Primitivas de caracteres	153
3.7	Almacenamiento de los valores en el búfer de imagen	102	3.21	Funciones OpenGL de caracteres	155
3.8	Funciones OpenGL para curvas	103	3.22	Particionamiento de imágenes	156
3.9	Algoritmos para generación de círculos	104	3.23	Listas de visualización de OpenGL	156
♦	Propiedades de los círculos	104	♦	Creación y denominación de una lista de visualización OpenGL	156
♦	Algoritmo del punto medio para círculos	106	♦	Ejecución de listas de visualización OpenGL	157
3.10	Algoritmos de generación de elipses	111	♦	Borrado de listas de visualización OpenGL	158
♦	Propiedades de las elipses	111	3.24	Función OpenGL de redimensionamiento de la ventana de visualización	158
♦	Algoritmo del punto medio para la elipse	113	3.25	Resumen	161
3.11	Otras curvas	120	Programas de ejemplo	165	
♦	Secciones cónicas	121	Referencias	173	
♦	Polinomios y curvas de tipo <i>spline</i>	123	Ejercicios	174	
3.12	Algoritmos paralelos para curvas	124			
3.13	Direccionamiento de píxeles y geometría de los objetos	124	<b>4 Atributos de las primitivas gráficas</b>	<b>178</b>	
♦	Coordenadas de cuadrícula de pantalla	124			
♦	Mantenimiento de las propiedades geométricas de los objetos visualizados	125	4.1	Variable de estado de OpenGL	180
3.14	Primitivas de áreas rellenas	127	4.2	Color es escala de grises	180
3.15	Áreas de relleno poligonales	128	♦	Las componentes de color RGB	180
♦	Clasificaciones de los polígonos	129	♦	Tablas de color	181
♦	Identificación de polígonos cóncavos	129	♦	Escala de grises	182
♦	División de los polígonos cóncavos	130	♦	Otros parámetros de color	182
♦	División de un polígono convexo en un conjunto de triángulos	131	4.3	Funciones de color de OpenGL	183
♦	Pruebas dentro-fuera	132	♦	Los modos de color RGB y RGBA de OpenGL	183
♦	Tablas de polígonos	135	♦	Modo de color indexado de OpenGL	184
♦	Ecuaciones de un plano	136	♦	Fundido de color en OpenGL	185
♦	Caras poligonales anteriores y posteriores	138	♦	Matrices de color en OpenGL	186
			♦	Otras funciones de color en OpenGL	187
			4.4	Atributos de los puntos	188
			4.5	Atributos de las líneas	188

♦ El grosor de las líneas	188	♦ Técnicas de filtrado	224
♦ Estilo de las líneas	190	♦ Ajuste de fase de los píxeles	225
♦ Opciones de plumilla y brocha	191	♦ Compensación de diferencias en la intensidad de las líneas	225
4.6 Atributos de las curvas	193	♦ Suavizado de los límites de las áreas	226
4.7 Funciones OpenGL para los atributos de los puntos	195	4.18 Funciones OpenGL de suavizado	228
4.8 Funciones OpenGL para los atributos de las líneas	196	4.19 Funciones de consulta de OpenGL	229
♦ Función de grosor de línea de OpenGL	196	4.20 Grupos de atributos de OpenGL	229
♦ La función de estilo de línea de OpenGL	196	4.21 Resumen	230
♦ Otros efectos de línea de OpenGL	198	Referencias	233
4.9 Atributos de relleno de áreas	199	Ejercicios	233
♦ Estilos de relleno	199		
♦ Relleno de regiones con fundido de color	200		
4.10 Algoritmo general de relleno de polígonos mediante líneas de barrido	202	5.1 Transformaciones geométricas bidimensionales básicas	238
4.11 Relleno de polígonos convexos mediante líneas de barrido	206	♦ Traslaciones bidimensionales	238
4.12 Relleno de regiones con límites curvos mediante líneas de barrido	207	♦ Rotaciones bidimensionales	240
4.13 Métodos de relleno de áreas con límites irregulares	207	♦ Cambio de escala bidimensional	242
♦ Algoritmo de relleno por contorno	207	5.2 Representación matricial y coordenadas homogéneas	244
♦ Algoritmo de relleno por inundación	211	♦ Coordenadas homogéneas	245
4.14 Funciones OpenGL para atributos de relleno de áreas	211	♦ Matriz de traslación bidimensional	245
♦ Función de relleno con patrón de OpenGL	212	♦ Matriz de rotación bidimensional	246
♦ Patrones de textura e interpolación de OpenGL	213	♦ Matriz de cambio de escala bidimensional	246
♦ Métodos OpenGL para modelos alámbricos	214	5.3 Transformaciones inversas	246
♦ La función de cara frontal de OpenGL	216	5.4 Transformaciones compuestas bidimensionales	247
4.15 Atributos de los caracteres	217	♦ Traslaciones compuestas bidimensionales	248
4.16 Funciones OpenGL para los atributos de caracteres	220	♦ Rotaciones compuestas bidimensionales	248
4.17 Suavizado	220	♦ Cambios de escala compuestos bidimensionales	248
♦ Supermuestreo de segmentos de línea recta	222	♦ Rotación general sobre un punto de pivote bidimensional	249
♦ Máscaras de ponderación de subpíxeles	223	♦ Cambio de escala general de puntos fijos bidimensionales	249
♦ Muestreo por área de segmentos de línea recta	224	♦ Directrices generales para el cambio de escala bidimensional	250
		♦ Propiedades de la concatenación de matrices	251
		♦ Transformaciones compuestas bidimensionales generales y eficiencia de cálculo	252

		<b>6</b>	<b>Visualización bidimensional</b>	<b>304</b>
	♦ Transformación bidimensional de sólido-rígido	253	6.1 <i>Pipeline</i> de visualización bidimensional	305
	♦ Construcción de matrices de rotación bidimensionales	254	6.2 La ventana de recorte	307
	♦ Ejemplo de programación de matrices bidimensionales compuestas	255	♦ Ventana de recorte en coordenadas de visualización	308
5.5	Otras transformaciones bidimensionales	260	♦ Ventana de recorte en coordenadas universales	308
	♦ Reflexión	260	6.3 Normalización y transformaciones de visor	309
	♦ Inclinar	263	♦ Mapeo de la ventana de recorte en un visor normalizado	310
5.6	Métodos de rasterización para transformaciones geométricas	265	♦ Mapeo de la ventana de recorte a un cuadrado normalizado	311
5.7	Transformaciones de rasterización en OpenGL	266	♦ Visualización de cadenas de caracteres	313
5.8	Transformaciones entre sistemas de coordenadas bidimensionales	267	♦ Efectos de división de pantalla y múltiples dispositivos de salida	313
5.9	Transformaciones geométricas en un espacio tridimensional	270	6.4 Funciones de visualización bidimensional de OpenGL	314
5.10	Traslaciones tridimensionales	270	♦ Modo de proyección de OpenGL	314
5.11	Rotaciones tridimensionales	271	♦ Función de ventana de recorte de GLUT	314
	♦ Rotaciones de ejes de coordenadas tridimensionales	272	♦ Función de visor de OpenGL	315
	♦ Rotaciones tridimensionales generales	274	♦ Creación de una ventana de visualización con GLUT	315
	♦ Métodos de cuaternios para rotaciones tridimensionales	280	♦ Establecimiento del modo y del color de la ventana de visualización con GLUT	316
5.12	Cambio de escala tridimensional	284	♦ Identificador de la ventana de visualización con GLUT	316
5.13	Transformaciones compuestas tridimensionales	287	♦ Borrado de una ventana de visualización con GLUT	316
5.14	Otras transformaciones tridimensionales	290	♦ Ventana de visualización actual con GLUT	317
	♦ Reflexiones tridimensionales	290	♦ Repositionamiento y cambio de tamaño de una ventana de visualización con GLUT	317
	♦ Inclinaciones tridimensionales	290	♦ Gestión de múltiples ventanas de visualización con GLUT	317
5.15	Transformaciones entre sistemas de coordenadas tridimensionales	291	♦ Subventanas de GLUT	318
5.16	Transformaciones afines	292	♦ Selección de la forma del cursor de pantalla en una ventana de visualización	318
5.17	Funciones de transformaciones geométricas en OpenGL	292	♦ Visualización de objetos gráficos en una ventana de visualización de GLUT	319
	♦ Transformaciones básicas en OpenGL	293		
	♦ Operaciones con matrices en OpenGL	293		
	♦ Pilas de matrices en OpenGL	295		
	♦ Ejemplos de programas de transformaciones geométricas OpenGL	296		
5.18	Resumen	299		
	Referencias	301		
	Ejercicios	301		

♦ Ejecución del programa de aplicación	319	♦ Visualización tridimensional y estereoscópica	360
♦ Otras funciones de GLUT	319	7.2 <i>Pipeline</i> de visualización tridimensional	360
♦ Ejemplo de programa de visualización bidimensional en OpenGL	320	7.3 Parámetros de coordenadas de visualización tridimensional	362
6.5 Algoritmos de recorte	322	♦ Vector normal del plano de visualización	362
6.6 Recorte de puntos bidimensionales	323	♦ El vector vertical	363
6.7 Recorte de líneas bidimensionales	323	♦ Sistema de referencia de coordenadas de visualización uvn	363
♦ Recorte de líneas de Cohen-Sutherland	324	♦ Generación de efectos de visualización tridimensionales	364
♦ Recorte de líneas de Liang-Barsky	330	7.4 Transformación de coordenadas universales a coordenadas de visualización	366
♦ Recorte de líneas de Nicholl-Lee-Nicholl	333	7.5 Transformaciones de proyección	368
♦ Recorte de líneas con ventanas de recorte poligonales no rectangulares	335	7.6 Proyecciones ortogonales	368
♦ Recorte de líneas utilizando ventanas de recorte con límites no lineales	335	♦ Proyecciones ortogonales axonométricas e isométricas	369
6.8 Recorte de áreas de relleno poligonales	336	♦ Coordenadas de proyección ortogonal	369
♦ Recorte de polígonos de Sutherland-Hodgman	338	♦ Ventana de recorte y volumen de visualización de proyección ortogonal	370
♦ Recorte de polígonos de Weiler-Atherton	343	♦ Transformación de normalización para una proyección ortogonal	372
♦ Recorte de polígonos utilizando ventanas de recorte poligonales no rectangulares	345	7.7 Proyecciones paralelas oblicuas	374
♦ Recorte de polígonos utilizando ventanas de recorte con límites no lineales	345	♦ Proyecciones paralelas oblicuas en diseño	374
6.9 Recorte de curvas	346	♦ Perspectivas caballera y cabinet	376
6.10 Recorte de textos	347	♦ Vector de proyección paralela oblicua	376
6.11 Resumen	349	♦ Ventana de recorte y volumen de visualización de proyección paralela oblicua	377
Referencias	352	♦ Matriz de transformación para proyección paralela oblicua	378
Ejercicios	352	♦ Transformación de normalización para una proyección paralela oblicua	379
<b>7 Visualización tridimensional</b>	<b>354</b>	7.8 Proyecciones en perspectiva	379
7.1 Panorámica de los conceptos de visualización tridimensional	355	♦ Transformación de coordenadas para la proyección en perspectiva	380
♦ Visualización de una escena tridimensional	355	♦ Ecuaciones de la proyección en perspectiva: casos especiales	381
♦ Proyecciones	356	♦ Puntos de fuga para las proyecciones en perspectiva	383
♦ Pistas de profundidad	356	♦ Volumen de visualización para proyección en perspectiva	383
♦ Identificación de líneas y superficies visibles	358		
♦ Representación de superficies	358		
♦ Despiece y secciones transversales	359		

♦ Matriz de transformación para la proyección en perspectiva	385	♦ Funciones para poliedros regulares de GLUT	416
♦ Frustrum de proyección en perspectiva simétrico	386	♦ Ejemplo de programa de poliedros con GLUT	418
♦ Frustrum de proyección en perspectiva oblicua	390	8.3 Superficies curvadas	420
♦ Coordenadas de transformación normalizadas para proyección en perspectiva	392	8.4 Superficies cuádricas	420
7.9 Transformación del visor y coordenadas de pantalla tridimensionales	395	♦ Esfera	420
7.10 Funciones de visualización tridimensional OpenGL	396	♦ Elipsoide	421
♦ Función de transformación de visualización OpenGL	396	♦ Toro	421
♦ Función de proyección ortogonal OpenGL	397	8.5 Supercuádricas	422
♦ Función OpenGL de proyección en perspectiva simétrica	398	♦ Superelipse	422
♦ Función general de proyección de perspectiva OpenGL	398	♦ Superelipsoide	423
♦ Visores OpenGL y ventanas de visualización	399	8.6 Funciones OpenGL para superficies cuádricas y superficies cúbicas	424
♦ Ejemplo de programa OpenGL para visualización tridimensional	399	♦ Funciones para superficies cuádricas de GLUT	424
7.11 Algoritmos de recorte tridimensional	401	♦ Función de GLUT para la generación de una tetera con una superficie cúbica	425
♦ Recorte en coordenadas homogéneas tridimensionales	402	♦ Funciones para la generación de superficies cuádricas de GLU	425
♦ Códigos de región tridimensional	402	♦ Ejemplo de programa que utiliza las funciones de creación de superficies cuádricas de GLUT y GLU	428
♦ Recorte tridimensional de puntos y líneas	403	8.7 Objetos din form (blobby)	429
♦ Recorte de polígonos tridimensionales	406	8.8 Representaciones con <i>splines</i>	431
♦ Recorte de curvas tridimensionales	407	♦ <i>Splines</i> de interpolación y de aproximación	432
♦ Planos de recorte arbitrarios	407	♦ Condiciones de continuidad paramétricas	434
7.12 Planos de recorte opcionales en OpenGL	409	♦ Condiciones de continuidad geométrica	434
7.13 Resumen	410	♦ Especificaciones de <i>splines</i>	435
Referencias	412	♦ Superficies con <i>splines</i>	436
Ejercicios	412	♦ Recorte de superficies con <i>splines</i>	437
<b>8 Representaciones de objetos tridimensionales</b>	<b>414</b>	8.9 Métodos de interpolación con <i>splines</i> cúbicos	437
8.1 Poliedros	416	♦ <i>Splines</i> cúbicos naturales	438
8.2 Funciones para poliedros en OpenGL	416	♦ Interpolación de Hermite	438
♦ Funciones de áreas de relleno de polígonos en OpenGL	416	♦ <i>Splines</i> cardinales	441
		♦ <i>Splines</i> de Kochanek-Bartels	444
		8.10 Curvas con <i>splines</i> de Bézier	445
		♦ Ecuaciones de las curvas de Bézier	445
		♦ Ejemplo de un programa de generación de curvas de Bézier	446
		♦ Propiedades de las curvas de Bézier	450

♦ Técnicas de diseño utilizando curvas de Bézier	450	♦ Procedimientos para generación de fractales	493
♦ Curvas de Bézier cúbicas	451	♦ Clasificación de fractales	494
8.11 Superficies de Bézier	454	♦ Dimensión fractal	494
8.12 Curvas con <i>splines</i> B	454	♦ Construcción geométrica de fractales deterministas autosimilares	497
♦ Ecuaciones de una curva con <i>splines</i> B	454	♦ Construcción geométrica de fractales estadísticamente autosimilares	499
♦ Curvas con <i>splines</i> B periódicos y uniformes	457	♦ Métodos de construcción de fractales afines	501
♦ Curvas con <i>splines</i> B cúbicos y periódicos	460	♦ Métodos de desplazamiento aleatorio del punto medio	504
♦ Curvas con <i>splines</i> B abiertos y uniformes	461	♦ Control de la topografía del terreno	506
♦ Curvas con <i>splines</i> B no uniformes	464	♦ Fractales autocuadráticos	507
8.13 Superficies con <i>splines</i> B	464	♦ Fractales autoinversos	519
8.14 <i>Splines</i> Beta	465	8.24 Gramáticas de formas y otros métodos procedimentales	521
♦ Condiciones de continuidad de los <i>splines</i> Beta	465	8.25 Sistemas de partículas	524
♦ Representación matricial de <i>splines</i> beta cúbicos y periódicos	466	8.26 Modelado basado en las características físicas	526
8.15 <i>Splines</i> racionales	467	8.27 Visualización de conjuntos de datos	529
8.16 Conversión entre representaciones de <i>splines</i>	469	♦ Representaciones visuales de campos escalares	529
8.17 Visualización de curvas y superficies con <i>splines</i>	470	♦ Representaciones visuales de campos vectoriales	533
♦ Regla de Horner	470	♦ Representaciones visuales de campos de tensores	535
♦ Cálculos de diferencias hacia adelante	471	♦ Representaciones visuales de campos de datos multivariantes	536
♦ Métodos de subdivisión	472	8.28 Resumen	537
8.18 Funciones OpenGL de aproximación con <i>splines</i>	474	Referencias	540
♦ Funciones OpenGL para curvas con <i>splines</i> de Bézier	474	Ejercicios	542
♦ Funciones OpenGL para superficies con <i>splines</i> de Bézier	477	<b>9 Métodos de detección de superficies visibles</b>	<b>546</b>
♦ Funciones GLU para curvas con <i>splines</i> B	480	9.1 Clasificación de los algoritmos de detección de superficies visibles	547
♦ Funciones GLU para la creación de superficies con <i>splines</i> B	482	9.2 Detección de caras posteriores	548
♦ Funciones GLU para el recorte de superficies	484	9.3 Método del búfer de profundidad	549
8.19 Representaciones de barrido	485	9.4 Método del búfer A	553
8.20 Métodos de geometría constructiva de sólidos	486	9.5 Método de la línea de barrido	554
8.21 Árboles octales	489	9.6 Método de orientación de la profundidad	556
8.22 Árboles BSP	492	9.7 Método del árbol BSP	558
8.23 Métodos de geometría fractal	492	9.8 Método de la subdivisión de áreas	559

9.9	Métodos de árboles octales	562	♦ Reflexión especular y modelo de Phong	588
9.10	Método de proyección de rayos	563	♦ Reflexiones difusa y especular combinadas	592
9.11	Comparación de los métodos de detección de visibilidad	564	♦ Reflexiones especular y difusa para múltiples fuentes luminosas	592
9.12	Superficies curvas	565	♦ Emisión de luz superficial	592
	Representación de superficies curvas	565	♦ Modelo básico de iluminación con focos y con atenuación de la intensidad	594
	Diagramas de contorno de superficies	566	♦ Consideraciones relativas al color RGB	594
9.13	Métodos de visibilidad para imágenes alámbricas	566	♦ Otras representaciones del color	596
	♦ Algoritmos de visibilidad de superficies para representaciones alámbricas	567	♦ Luminancia	596
	♦ Algoritmo de variación de intensidad con la profundidad para representaciones alámbricas	567	10.4 Superficies transparentes	597
9.14	Funciones OpenGL de detección de visibilidad	568	♦ Materiales translúcidos	597
	♦ Funciones OpenGL de eliminación de polígonos	569	♦ Refracción de la luz	598
	♦ Funciones OpenGL de gestión del búfer de profundidad	569	♦ Modelo básico de transparencia	600
	♦ Métodos OpenGL para visibilidad de superficies en representaciones alámbricas	570	10.5 Efectos atmosféricos	601
	♦ Función OpenGL para variación de la intensidad con la profundidad	571	10.6 Sombras	601
9.15	Resumen	571	10.7 Parámetros de la cámara	602
	Referencias	573	10.8 Visualización de la intensidad de la luz	602
	Ejercicios	573	♦ Distribución de los niveles de intensidad del sistema	603
			♦ Corrección gamma y tablas de sustitución de vídeo	604
			♦ Visualización de imágenes de plano continuo	605
<b>10</b>	<b>Modelos de iluminación y métodos de representación superficial</b>	<b>576</b>	10.9 Patrones de semitono y técnicas de aleatorización	606
10.1	Fuentes luminosas	578	♦ Aproximaciones de semitonos	607
	♦ Fuentes luminosas puntuales	578	♦ Técnicas de aleatorización	610
	♦ Fuentes luminosas infinitamente distantes	579	10.10 Métodos de representación de polígonos	613
	♦ Atenuación radial de la intensidad	579	♦ Representación superficial con intensidad constante	614
	♦ Fuentes de luz direccionales y efectos de foco	580	♦ Representación de superficies por el método de Gouraud	614
	♦ Atenuación angular de la intensidad	581	♦ Representación superficial de Phong	617
	♦ Fuentes luminosas complejas y el modelo de Warn	582	♦ Representación superficial rápida de Phong	617
10.2	Efectos de iluminación superficial	583	10.11 Métodos de trazado de rayos	618
10.3	Modelos básicos de iluminación	584	♦ Algoritmo básico de trazado de rayos	620
	♦ Luz ambiente	584	♦ Cálculos de intersección entre rayos y superficie	622
	♦ Reflexión difusa	584	♦ Intersecciones entre rayos y esferas	623

♦ Intersecciones entre rayos y poliedros	625	♦ Funciones de representación superficial OpenGL	667
♦ Reducción de los cálculos de intersección con los objetos	626	♦ Operaciones de semitonos en OpenGL	668
♦ Métodos de subdivisión espacial	626	10.21 Funciones de texturas en OpenGL	668
♦ Simulación de los efectos de enfoque de la cámara	630	♦ Funciones OpenGL para texturas lineales	669
♦ Trazado de rayos con <i>antialiasing</i>	632	♦ Funciones OpenGL para texturas superficiales	672
♦ Trazado de rayos distribuido	634	♦ Funciones OpenGL para texturas volumétricas	673
10.12 Modelo de iluminación de radiosidad	638	♦ Opciones de color OpenGL para patrones de texturas	674
♦ Términos de la energía radiante	638	♦ Opciones OpenGL para el mapeado de texturas	674
♦ Modelo básico de radiosidad	639	♦ Envolvimiento de texturas en OpenGL	675
♦ Método de radiosidad mediante refinamiento progresivo	643	♦ Copia de patrones de texturas OpenGL desde el búfer de imagen	675
10.13 Mapeado de entorno	646	♦ Matrices de coordenadas de texturas en OpenGL	676
10.14 Mapeado de fotones	646	♦ Denominación de los patrones de textura OpenGL	676
10.15 Adición de detalles a las superficies	647	♦ Subpatrones de textura en OpenGL	677
10.16 Modelado de los detalles superficiales mediante polígonos	650	♦ Patrones de reducción de texturas en OpenGL	677
10.17 Mapeado de texturas	650	♦ Bordes de texturas en OpenGL	678
♦ Patrones de textura lineales	650	♦ Texturas proxy en OpenGL	679
♦ Patrones de textura superficial	651	♦ Texturado automático de superficies cuádricas	679
♦ Patrones de textura volumétricos	654	♦ Coordenadas de textura homogéneas	679
♦ Patrones de reducción de texturas	655	♦ Opciones adicionales para texturas en OpenGL	680
♦ Métodos de texturado procedimental	655	10.22 Resumen	680
10.18 Mapeado de relieve	656	Referencias	683
10.19 Mapeado del sistema de referencia	658	Ejercicios	684
10.20 Funciones OpenGL de iluminación y representación de superficies	659		
♦ Función OpenGL para fuentes luminosas puntuales	659		
♦ Especificación de la posición y el tipo de una fuente lumínosa en OpenGL	659		
♦ Especificación de los colores de las fuentes luminosas en OpenGL	660		
♦ Especificación de coeficientes de atenuación radial de la intensidad para una fuente lumínosa OpenGL	661		
♦ Fuentes luminosas direccionales en OpenGL (focos)	661		
♦ Parámetros de iluminación globales en OpenGL	662	<b>11 Métodos interactivos de entrada e interfaces gráficas de usuario</b>	<b>688</b>
♦ Función OpenGL de propiedad de una superficie	664	11.1 Datos de entrada gráficos	689
♦ Modelo de iluminación OpenGL	665	11.2 Clasificación lógica de los dispositivos de entrada	689
♦ Efectos atmosféricos en OpenGL	665	♦ Dispositivos localizadores	690
♦ Funciones de transparencia OpenGL	666	♦ Dispositivos de trazo	690
		♦ Dispositivos de cadena de caracteres	690

♦ Dispositivos evaluadores	690	♦ Realimentación	726
♦ Dispositivos de elección	691	11.9 Resumen	727
♦ Dispositivos de selección	692	Referencias	730
11.3 Funciones de entrada para datos gráficos	694	Ejercicios	730
♦ Modos de entrada	694		
♦ Realimentación mediante eco	695		
♦ Funciones de retrollamada	695		
11.4 Técnicas interactivas de construcción de imágenes	695	<b>12 Modelos y aplicaciones del color</b>	<b>734</b>
♦ Métodos básicos de posicionamiento	695	12.1 Propiedades de la luz	735
♦ Arrastre de objetos	696	♦ El espectro electromagnético	735
♦ Restricciones	696	♦ Características psicológicas del color	737
♦ Cuadrículas	696	12.2 Modelos de color	738
♦ Métodos de banda elástica	696	♦ Colores primarios	738
♦ Campo de gravedad	698	♦ Conceptos intuitivos del color	738
♦ Métodos interactivos de dibujo	699	12.3 Primarios estándar y diagrama cromático	739
11.5 Entornos de realidad virtual	699	♦ El modelo de color XYZ	739
11.6 Funciones OpenGL para dispositivos de entrada interactiva	700	♦ Valores XYZ normalizados	740
♦ Funciones de ratón GLUT	700	♦ Diagrama cromático de la CIE	740
♦ Funciones de teclado GLUT	705	♦ Gamas de colores	741
♦ Funciones GLUT para tabletas gráficas	710	♦ Colores complementarios	741
♦ Funciones GLUT para una <i>spaceball</i>	710	♦ Longitud de onda dominante	741
♦ Funciones GLUT para cajas de botones	711	♦ Pureza	742
♦ Funciones GLUT para diales	711	12.4 El modelo de color RGB	742
♦ Operaciones de selección en OpenGL	711	12.5 El modelo de color YIQ y los modelos relacionados	744
11.7 Funciones de menú OpenGL	717	♦ Los parámetros YIQ	744
♦ Creación de un menú GLUT	717	♦ Transformaciones entre los espacios de color RGB e YIQ	745
♦ Creación y gestión de múltiples menús GLUT	720	♦ Los sistemas YUV e YCrCb	745
♦ Creación de submenús GLUT	720	12.6 Los modelos de color CMY y CMYK	745
♦ Modificación de los menús GLUT	723	♦ Los parámetros CMY	745
11.8 Diseño de una interfaz gráfica de usuario	724	♦ Transformaciones entre los espacios de color CMY y RGB	746
♦ El diálogo con el usuario	724	12.7 El modelo de color HSV	747
♦ Ventanas e iconos	724	♦ Los parámetros HSV	747
♦ Adaptación a los distintos niveles de experiencia	725	♦ Selección de sombras, tintas y tonalidades	748
♦ Coherencia	726	♦ Transformaciones entre los espacios de color HSV y RGB	749
♦ Minimización de la memorización	726	12.8 El modelo de color HLS	750
♦ Cancelación de acciones y tratamiento de errores	726	12.9 Selección y aplicaciones del color	751
		12.10 Resumen	751
		Referencias	752
		Ejercicios	752

<b>13</b>	<b>Animación por computadora</b>	<b>754</b>	<b>15</b>	<b>Formatos de archivos gráficos</b>	<b>790</b>	
13.1	Métodos de barrido para las animaciones por computadora	756	15.1	Configuraciones de archivos de imagen	791	
♦	Doble búfer	756	15.2	Métodos de reducción de color	792	
♦	Generación de animaciones mediante operaciones de barrido	757	♦	Reducción uniforme de color	792	
13.2	Diseño de secuencias de animación	757	♦	Reducción de color por popularidad	792	
13.3	Técnicas tradicionales de animación	759	♦	Reducción de color de corte medio	793	
13.4	Funciones generales de animación por computadora	760	15.3	Técnicas de compresión de archivos	793	
13.5	Lenguajes de animación por computadora	760	♦	Codificación de longitud de recorrido	794	
13.6	Sistemas de forogramas clave	761	♦	Codificación LZW	794	
♦	Morfismo	762	♦	Otros métodos de compresión mediante reconocimiento de patrones	795	
♦	Simulación de aceleraciones	764	♦	Codificación Huffman	795	
13.7	Especificaciones de movimientos	767	♦	Codificación aritmética	798	
♦	Especificación directa del movimiento	767	♦	Trasformada discreta del coseno	799	
♦	Sistemas dirigidos por objetivos	768	15.4	Composición de la mayoría de formatos de archivo	801	
♦	Cinemática y dinámica	768	♦	JPEG: Joint Photographic Experts Group	801	
13.8	Animación de figuras articuladas	769	♦	CGM: Computer-Graphics Metafile Format	803	
13.9	Movimientos periódicos	771	♦	TIFF: Tag Image-File Format	803	
13.10	Procedimientos de animación en OpenGL	772	♦	PNG: Portable Network-Graphics Format	803	
13.11	Resumen	775	♦	XBM: X Window System Bitmap Format y XPM: X Window System Pixmap Format	804	
	Referencias	776	♦	Formato Adobe Photoshop	804	
	Ejercicios	777	♦	MacPaint: Macintosh Paint Format	804	
<b>14</b>	<b>Modelado jerárquico</b>	<b>778</b>		♦	PICT: Formato Picture Data	804
14.1	Conceptos básicos de modelado	779		♦	BMP: Formato Bitmap	805
♦	Representaciones de los sistemas	779		♦	PCX: Formato de archivo PC Paintbrush	805
♦	Jerarquías de símbolos	781		♦	TGA: Formato Truevision Graphics-Adapter	805
14.2	Paquetes de modelado	782		♦	GIF: Graphics Interchange Format	805
14.3	Métodos generales de modelado jerárquico	784	15.5	Resumen	805	
♦	Coordenadas locales	784		Referencias	806	
♦	Transformaciones de modelado	785		Ejercicios	806	
♦	Creación de estructuras jerárquicas	785				
14.4	Modelado jerárquico mediante listas de visualización OpenGL	787	<b>A</b>	<b>Matemáticas para gráficos por computadora</b>	<b>809</b>	
14.5	Resumen	787				
	Referencias	788				
	Ejercicios	788	A.1	Sistemas de coordenadas	809	

♦ Coordenadas de pantalla cartesianas bidimensionales	809	♦ Derivada direccional	831
♦ Sistemas de referencia cartesianos bidimensionales estándar	809	♦ Forma general del operador gradiente	831
♦ Coordenadas polares en el plano xy	810	♦ Operador de Laplace	831
♦ Sistemas de referencia cartesianos tridimensionales estándar	811	♦ Operador divergencia	832
♦ Coordenadas de pantalla cartesianas tridimensionales	812	♦ Operador rotacional	832
♦ Sistemas de coordenadas curvilíneas tridimensionales	812	A.11 Teoremas de transformación integrales	833
♦ Ángulo sólido	814	♦ Teorema de Stokes	833
A.2 Puntos y vectores	814	♦ Teorema de Green para una superficie plana	833
♦ Propiedades de los puntos	814	♦ Teorema de divergencia	835
♦ Propiedades de los vectores	814	♦ Ecuaciones de transformación de Green	835
♦ Suma de vectores y multiplicación escalar	816	A.12 Área y centroide de un polígono	835
♦ Producto escalar de dos vectores	816	♦ Área de un polígono	836
♦ Producto vectorial de dos vectores	817	♦ Centroide de un polígono	836
A.3 Tensores	818	A.13 Cálculo de las propiedades de los poliedros	838
A.4 Vectores base y tensor métrico	818	A.14 Métodos numéricos	839
♦ Determinación de los vectores base para un espacio de coordenadas	819	♦ Resolución de sistemas de ecuaciones lineales	839
♦ Bases ortonormales	820	♦ Determinación de raíces de ecuaciones no lineales	841
♦ Tensor métrico	820	♦ Evaluación de integrales	842
A.5 Matrices	821	♦ Resolución de ecuaciones diferenciales ordinarias	844
♦ Multiplicación por un escalar y suma de matrices	822	♦ Resolución de ecuaciones diferenciales parciales	845
♦ Multiplicación de matrices	822	♦ Métodos de ajuste de curvas por mínimos cuadrados para conjuntos de datos	846
♦ Traspuesta de una matriz	823	<b>Bibliografía</b>	<b>849</b>
♦ Determinante de una matriz	823	<b>Índice</b>	<b>867</b>
♦ Inversa de una matriz	824	<b>Índice de funciones OpenGL</b>	<b>893</b>
A.6 Números complejos	824		
♦ Aritmética compleja básica	825		
♦ Unidad imaginaria	825		
♦ Conjugado complejo y módulo de un número complejo	826		
♦ División compleja	826		
♦ Representación en coordenadas polares de un número complejo	826		
A.7 Cuaternios	827		
A.8 Representaciones no paramétricas	828		
A.9 Representaciones paramétricas	829		
A.10 Operadores diferenciales	829		
♦ Operador gradiente	830		

# Prefacio

La infografía, es decir, los gráficos por computadora, continua siendo una de las áreas más excitantes y de más rápido crecimiento de la moderna tecnología. Desde la aparición de la primera edición de este libro, los métodos infográficos se han convertido en una característica estándar del software de aplicación y de los sistemas informáticos en general. Los métodos infográficos se aplican de forma rutinaria en el diseño de la mayoría de los productos, en los simuladores para actividades de programación, en la producción de videos musicales y anuncios de televisión, en las películas, en el análisis de datos, en los estudios científicos, en las intervenciones médicas y en muchísimas otras aplicaciones. Hoy en día, se utiliza una gran variedad de técnicas y de dispositivos hardware en estas diversas áreas de aplicación, y hay muchas otras técnicas y dispositivos actualmente en desarrollo. En particular, buena parte de las investigaciones actuales en el campo de la infografía están relacionadas con la mejora de la efectividad, del realismo y de la velocidad de generación de imágenes. Para conseguir una vista realista de una escena natural, un programa gráfico puede simular los efectos de las reflexiones y refracciones reales de la luz en los objetos físicos. Como consecuencia, la tendencia actual en los gráficos por computadora consiste en incorporar mejores aproximaciones de los principios físicos dentro de los algoritmos gráficos, con el fin de simular mejor las complejas interacciones existentes entre los objetos y el entorno de iluminación.

## Características de la tercera edición

El material de esta tercera edición ha evolucionado a partir de una serie de notas utilizadas en diversos cursos que hemos impartido a lo largo de los años, incluyendo cursos de introducción a la infografía, infografía avanzada, visualización científica, temas especiales y cursos de proyecto. Cuando escribimos la primera edición de este libro, muchos aplicaciones y muchos cursos sobre gráficos sólo trataban con métodos bidimensionales, así que decidimos separar las explicaciones relativas a las técnicas gráficas bidimensionales y tridimensionales. En la primera parte del libro se proporcionaba una sólida base sobre los procedimientos infográficos bidimensionales, mientras que los métodos tridimensionales se analizaban en la segunda mitad. Ahora, sin embargo, las aplicaciones gráficas tridimensionales resultan comunes y muchos cursos introductorios a la infografía tratan principalmente con métodos tridimensionales o introducen los gráficos tridimensionales en una etapa relativamente temprana. Por tanto, una de las principales características de esta tercera edición es la integración de los temas relativos a gráficos tridimensionales y bidimensionales.

También hemos ampliado el tratamiento de la mayoría de los temas para incluir análisis de desarrollos recientes y nuevas aplicaciones. Los temas generales que se cubren en esta tercera edición incluyen: componentes hardware y software actuales de los sistemas gráficos, geometría fractal, trazado de rayos, splines, modelos de iluminación, representación de superficies, iluminación por computadora, realidad virtual, implementaciones paralelas de algoritmos gráficos, antialiasing, supercuádricas, árboles BSP, sistemas de partículas, modelado físico, visualización científica, radiosidad, mapeado de relieve y morfismo. Algunas de las principales áreas de ampliación son la animación, las representaciones de objetos, la pipeline de visualización tridimensional, los modelos de iluminación, las técnicas de representación superficial y el mapeado de texturas.

Otro cambio importante en esta tercera edición es la introducción del conjunto de rutinas gráficas OpenGL, que ahora se utiliza ampliamente y que está disponible en la mayoría de los sistemas informáticos.

El paquete OpenGL proporciona una amplia y eficiente colección de funciones independientes del dispositivo para la creación de imágenes infográficas, utilizando un programa escrito en un lenguaje de propósito general tal como C o C++. OpenGL ofrece bibliotecas auxiliares para el manejo de operaciones de entrada y de salida, que requieren interacción con los dispositivos, y para procedimientos gráficos adicionales como la generación de formas cilíndricas, objetos esféricos y B-splines.

## Ejemplos de programación

En esta tercera edición se proporcionan más de veinte programas C++ completos, utilizando la biblioteca de rutinas gráficas disponible en el popular paquete OpenGL. Estos programas ilustran las aplicaciones de las técnicas básicas de construcción de imágenes, las transformaciones geométricas bidimensionales y tridimensionales, los métodos de visualización bidimensionales y tridimensionales, las proyecciones en perspectiva, la generación de splines, los métodos fractales, la entrada interactiva mediante el ratón, las operaciones de selección, la visualización de menús y submenús y las técnicas de animación. Además, se proporcionan más de 100 fragmentos de programas C++/OpenGL para ilustrar la implementación de algoritmos infográficos de recorte, efectos de iluminación, representación superficial, mapeado de texturas y muchos otros métodos infográficos.

## Conocimientos requeridos

No asumimos que el lector tenga ninguna familiaridad previa con los gráficos por computadora, pero sí que debe tener unos ciertos conocimientos de programación y de estructuras básicas de datos, tales como matrices, listas de punteros, archivos y organizaciones de registros. En los algoritmos infográficos se utilizan diversos métodos matemáticos, y estos métodos se explican con un cierto detalle en el apéndice. Los temas matemáticos cubiertos en el apéndice incluyen técnicas diversas que van desde la geometría analítica hasta el análisis numérico, pasando por el álgebra lineal, el análisis vectorial y tensorial, los números complejos, los cuaternios y el cálculo básico.

Esta tercera edición puede utilizarse tanto como un texto para estudiantes que no tengan conocimientos previos de infografía, cuanto como referencia para los profesionales de los gráficos por computadora. El énfasis del libro se pone en los principios básicos necesarios para diseñar, utilizar y comprender los sistemas infográficos, junto con numerosos programas de ejemplo que ilustran los métodos y aplicaciones relativos a cada tema.

## Estructuraciones sugeridas para el curso

Para un curso de un semestre, puede elegirse un subconjunto de temas que traten acerca de los métodos bidimensionales o de una combinación de métodos bidimensionales y tridimensionales, dependiendo de las necesidades de cada curso concreto. Un curso de dos semestres puede cubrir los conceptos gráficos básicos y los algoritmos en el primer semestre y los métodos tridimensionales avanzados en el segundo. Para el lector autodidacta, los capítulos iniciales pueden usarse para comprender los conceptos gráficos, suplementando estos conceptos con temas seleccionados de los capítulos posteriores.

En los primeros cursos universitarios, puede organizarse un curso de introducción a la infografía utilizando materiales seleccionados de los Capítulos 2 a 6, 11 y 13. Pueden elegirse las adecuadas secciones de estos capítulos para cubrir únicamente los métodos bidimensionales, o bien pueden añadirse temas relativos a los gráficos tridimensionales extraídos de estos capítulos, junto con determinado materiales de los Capítulos 7 y 10. Otros temas, como las representaciones fractales, las curvas splines, el mapeado de texturas, los métodos basados en búfer de profundidad o los modelos de color, pueden introducirse en un primer curso sobre infografía. Para los cursos universitarios posteriores de carácter introductorio, puede hacerse más énfasis en la visualización tridimensional, el modelado tridimensional, los modelos de iluminación y los métodos de representación de superficies. En general, sin embargo, una secuencia de dos semestres constituye un mejor marco

para cubrir adecuadamente los fundamentos de los métodos infográficos bidimensionales y tridimensionales, incluyendo las representaciones mediante splines, la representación de superficies y el trazado de rayos. También pueden ofrecerse cursos dedicados a temas especiales, para los que se requiera un conocimiento básico de infografía como prerequisito, centrando esos curso en una o dos áreas seleccionadas como por ejemplo técnicas de visualización, geometría fractal, métodos basados en splines, trazado de rayos, radiosidad y animación por computadora.

El Capítulo 1 ilustra la diversidad de aplicaciones infográficas existentes, examinando los numerosos tipos distintos de imágenes que se generan mediante software gráfico. En el Capítulo 2 se presenta el vocabulario básico del campo de la infografía, junto con una introducción a los componentes hardware y software de los sistemas gráficos, una introducción detallada a OpenGL y un programa OpenGL complejo de ejemplo. Los algoritmos fundamentales para la representación y visualización de objetos simples se proporcionan en los Capítulos 3 y 4. Estos dos capítulos examinan los métodos para generar componentes gráficos de las imágenes tales como polígonos y círculos; para establecer el color, tamaño y otros atributos de los objetos; y para implementar dichos métodos en OpenGL. El Capítulo 5 analiza los algoritmos para realizar transformaciones geométricas tales como la rotación y el cambio de escala. En los Capítulos 6 y 7, se proporcionan explicaciones detalladas de los procedimientos utilizados para mostrar vistas de escenas bidimensionales y tridimensionales. Los métodos para la generación de imágenes de objetos complejos, tales como superficies cuádricas, splines, fractales y sistemas de partículas se explican en el Capítulo 8. En el Capítulo 9 exploramos las diversas técnicas infográficas que se utilizan para identificar los objetos visibles en una escena tridimensional. Los modelos de iluminación y los métodos para aplicar condiciones de iluminación en la escena se examinan en el Capítulo 10, mientras que los métodos para la entrada gráfica interactiva y para el diseño de interfaces gráficas de usuario se repasan en el Capítulo 11. Los diversos modelos de color que resultan útiles en la infografía se analizan en el Capítulo 12, donde también se proporcionan consideraciones relativas al diseño de imágenes en color. Las técnicas de animación por computadora se exploran en el Capítulo 13. Los métodos para el modelado jerárquico de sistemas complejos se presentan en el Capítulo 14 y, finalmente, en el Capítulo 15 se hace un repaso de los principales formatos de archivos gráficos.

## Agradecimientos

Son muchas las personas que han contribuido a este proyecto de diversas formas a lo largo de los años. Nos gustaría expresar de nuevo nuestro agradecimiento a las organizaciones y personas que nos han proporcionado imágenes y otros materiales, así como a los estudiantes de los diversos cursos y seminarios sobre infografía y visualización que hemos impartido, los cuales nos han proporcionado numerosos comentarios útiles. Estamos en deuda con todos aquellos que nos han proporcionado comentarios, que han realizado revisiones, que nos han hecho llegar sugerencias para mejorar el material cubierto en el libro y que nos han ayudado de numerosas otras formas, y queríamos de forma expresa disculparnos con todas aquellas personas a las que nos hayamos olvidado de mencionar. Damos nuestro agradecimiento a Ed Angel, Norman Badler, Phillip Barry, Brian Barsky, Hedley Bond, Bart Braden, Lara Burton, Robert Burton, Greg Chwelos, John Cross, Steve Cunningham, John DeCatrell, Victor Duvanecko, Gary Eerkes, Parris Egbert, Tony Faustini, Thomas Foley, Thomas Frank, Don Gillies, Andrew Glassner, Jack Goldfeather, Georges Grinstein, Eric Haines, Robert Herbst, Larry Hodges, Carol Hubbard, Eng-Kiat Koh, Mike Krogh, Michael Laszlo, Suzanne Lea, Michael May, Nelson Max, David McAllister, Jeffrey McConnell, Gary McDonald, C. L. Morgan, Greg Nielson, James Oliver, Lee-Hian Quek, Laurence Rainville, Paul Ross, David Salomon, Günther Schrack, Steven Shafer, Cliff Shaffer, Pete Shirley, Carol Smith, Stephanie Smullen, Jeff Spears, William Taffe, Wai Wan Tsang, Spencer Thomas, Sam Uselton, David Wen, Bill Wicker, Andrew Woo, Angelo Yfantis, Marek Zaremba, Michael Zyda y a los numerosos revisores anónimos. También queremos dar las gracias a nuestro editor Alan Apt, a Toni Holm y al equipo de Colorado por su ayuda, sus sugerencias, su apoyo y, por encima de todo, su paciencia durante la preparación de esta tercera edición. También deseamos dar las gracias a nuestros editores de producción a su equipo, Lynda Castillo, Camille Trentacoste, Heather Scott, Xiaohong Zhu, Vince

O'Brien, Patricia Burns, Kathy Ewing y David Abel; agradecemos de verdad su valiosa ayuda y su cuidada atención al detalle.

### **Advertencia al lector**

La versión en inglés del libro, *Computer Graphics with OpenGL*, está impresa en color. Sin embargo, *Gráficos por computadora en OpenGL*, que es la versión traducida al español se ha impreso en blanco y negro, por lo que todas las imágenes y fotografías se presentan en escala de grises. El lector debe entonces sobreentender que cuando en el texto se hace referencia a códigos de colores, siempre debe pensarse en la imagen o fotografía en color.



# Gráficos por computadora con OpenGL

# Introducción a los gráficos por computadora



Una escena de una película de dibujos animados generada por computadora de Saguaro-Entertainer. (*Cortesía de SOFTIMAGE, Inc.*)

- |            |                                 |             |                                |
|------------|---------------------------------|-------------|--------------------------------|
| <b>1.1</b> | Gráficos y diagramas            | <b>1.6</b>  | Arte por computadora           |
| <b>1.2</b> | Diseño asistido por computadora | <b>1.7</b>  | Entretenimiento                |
| <b>1.3</b> | Entornos de realidad virtual    | <b>1.8</b>  | Procesamiento de imágenes      |
| <b>1.4</b> | Visualización de datos          | <b>1.9</b>  | Interfaces gráficas de usuario |
| <b>1.5</b> | Educación y formación           | <b>1.10</b> | Resumen                        |

Los gráficos por computadora se han convertido en una potente herramienta para la producción rápida y económica de imágenes. Prácticamente no existe ninguna tarea en la que la representación gráfica de la información no pueda aportar alguna ventaja y, por tanto, no sorprende encontrar gráficos por computadora en muchos sectores. Aunque las primeras aplicaciones de ciencia e ingeniería requerían equipos caros y aparatosos, los avances en la tecnología informática han hecho de los gráficos interactivos una herramienta muy útil. Actualmente, los gráficos por computadora se usan a diario en campos tan diversos como las ciencias, las artes, la ingeniería, los negocios, la industria, la medicina, las administraciones públicas, el entretenimiento, la publicidad, la educación, la formación y en aplicaciones caseras. E incluso podemos transmitir imágenes alrededor del mundo a través de Internet. La Figura 1.1 presenta un breve resumen de diversas aplicaciones gráficas en simulaciones, formación y representación de datos. Antes de entrar en detalle sobre cómo hacer gráficos con una computadora, vamos a hacer una pequeña visita a una galería de aplicaciones gráficas.

## 1.1 GRÁFICOS Y DIAGRAMAS

---

Las primeras aplicaciones de los gráficos por computadora fueron para visualizar gráficos de datos que, frecuentemente, se imprimían con impresoras de caracteres. Todavía la representación gráfica de datos es una de las aplicaciones más comunes, pero hoy podemos generar fácilmente gráficos que muestren complejas relaciones entre datos para realizar informes escritos o para presentarlos mediante diapositivas, transparencias o animaciones en video. Los gráficos y los diagramas se usan comúnmente para realizar resúmenes financieros, estadísticos, matemáticos, científicos, de ingeniería y económicos, para realizar informes de investigación, resúmenes de gestión, boletines de información al consumidor y otros tipos de publicaciones. Hay disponibles una gran variedad de paquetes gráficos y de dispositivos para estaciones de trabajo, así como servicios comerciales para convertir imágenes generadas en la pantalla de una computadora en películas, diapositivas o transparencias para presentaciones o para guardar en archivos. Ejemplos típicos de visualización de datos son los diagramas lineales, de barras, los diagramas de tarta, gráficos de superficie, diagramas de contorno y otras muchas representaciones en las que se muestran las relaciones entre múltiples parámetros en espacios de dos, tres o más dimensiones.

Las Figuras 1.1 y 1.2 muestran ejemplos de representaciones de datos en dos dimensiones. Estas dos figuras presentan ejemplos básicos de gráficos lineales, diagramas de barras y diagramas de tarta. En este último podemos ver cómo se resalta la información desplazando radialmente las diferentes secciones produciendo un diagrama de tarta en «explosión».

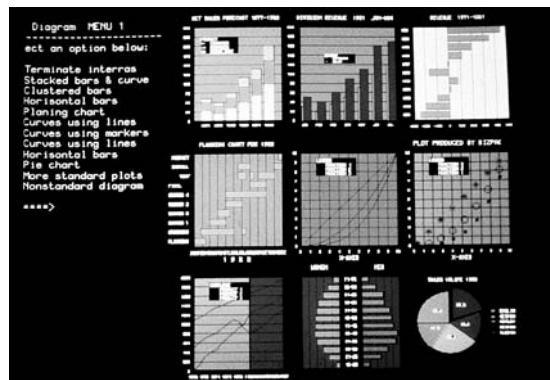
Los diagramas y gráficos tridimensionales se usan para mostrar información adicional, aunque algunas veces simplemente se usan para causar efecto, dando mayor dramatismo y haciendo más atractivas las presentaciones de relaciones entre datos. La Figura 1.3 muestra un diagrama de barras 3D combinado con información geográfica. La Figura 1.4 proporciona ejemplos de gráficos tridimensionales con efectos dramáticos.

#### 4 CAPÍTULO 1 Introducción a los gráficos por computadora

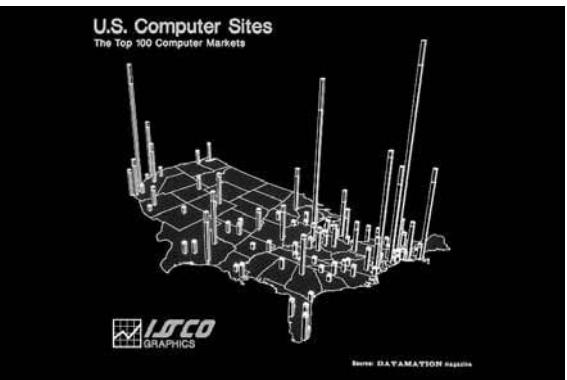
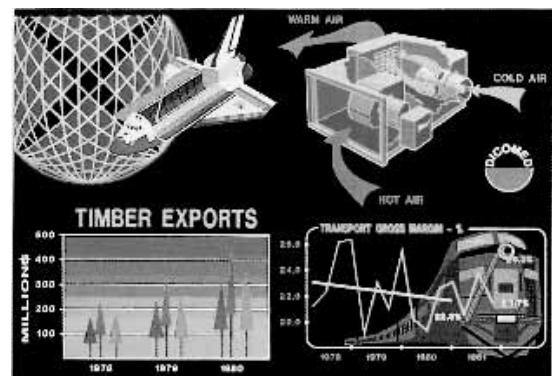
Otro ejemplo de graficos de datos 3D es la representación de superficie, como la que se ilustra en la Figura 1.5, en la que se muestra una superficie equipotencial y su contorno bidimensional proyectado.

La Figura 1.6 muestra un diagrama de tiempos para la planificación de tareas. Los diagramas de tiempo y los grafos de planificación de tareas se usan para la dirección de proyectos y para monitorizar y ordenar en el tiempo el progreso de los mismos.

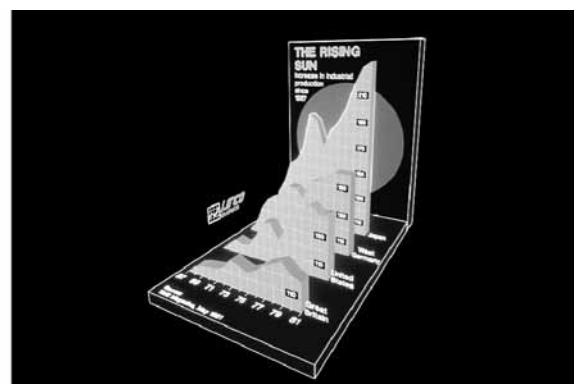
**FIGURA 1.1.** Ejemplos de gráficos por computadora utilizados en diversas áreas. (Cortesía de DICO-MED Corporation.)



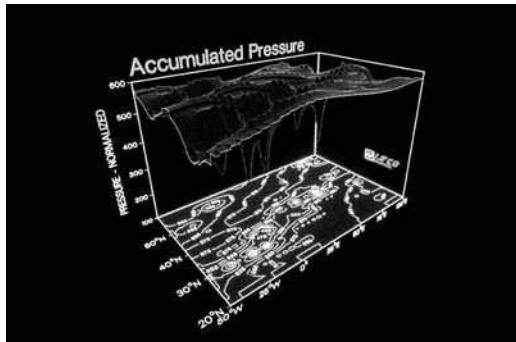
**FIGURA 1.2.** Gráficos bidimensionales de línea, diagramas de barras y diagramas de tarta. (Cortesía de UNIRAS, INC.)



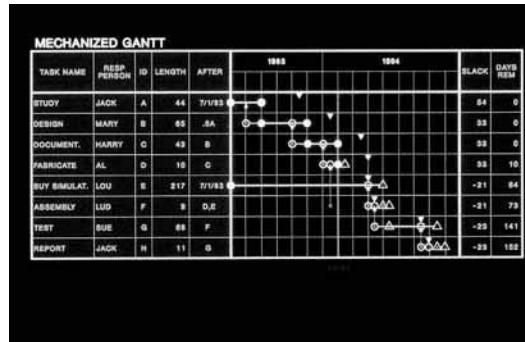
**FIGURA 1.3.** Conjuntos de datos codificados usando dos colores presentados como diagramas de barras tridimensionales en la superficie de una región geográfica. (Reimpreso con permiso de ISSCO Graphics, San Diego, California.)



**FIGURA 1.4.** Dos gráficos tridimensionales diseñados con efecto dramático. (Reimpreso con permiso de ISSCO Graphics, San Diego, California.)



**FIGURA 1.5.** Representación de contornos bidimensionales en un plano de tierra, con un campo de potencial. (Reimpreso con permiso de ISSCO Graphics, San Diego, California.)

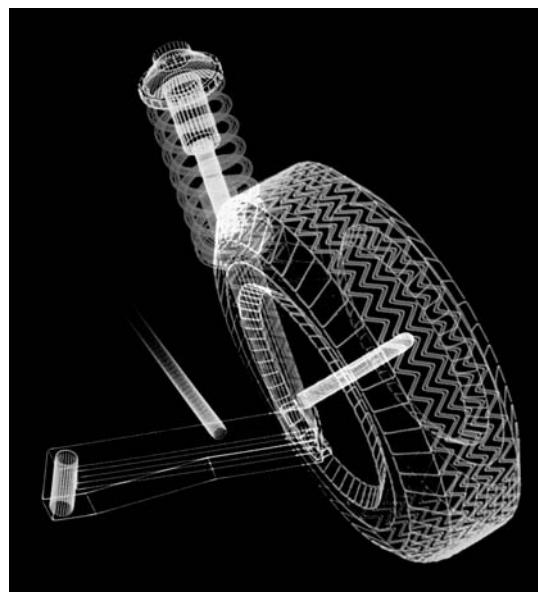


**FIGURA 1.6.** Un diagrama de tiempos para la planificación de tareas y otras informaciones relevantes para las tareas de un proyecto. (Reimpreso con permiso de ISSCO Graphics, San Diego, California)

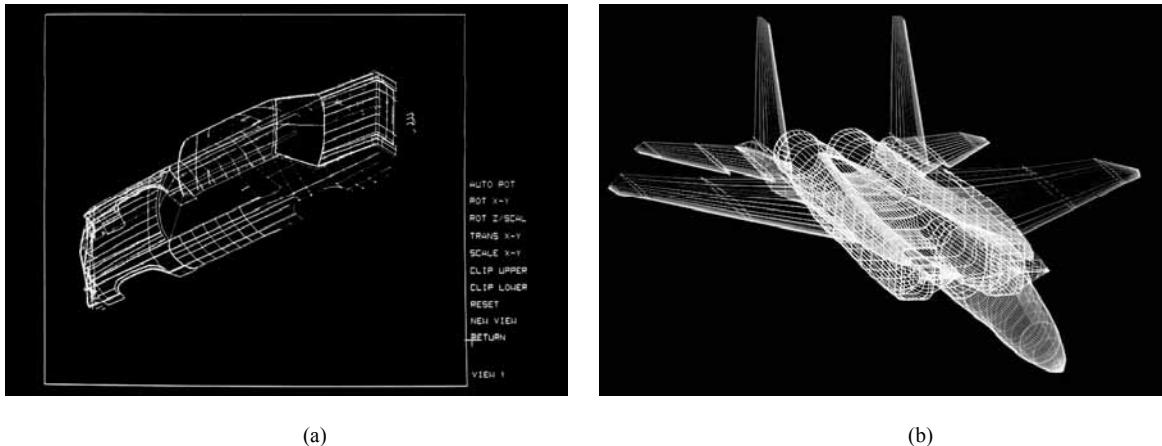
## 1.2 DISEÑO ASISTIDO POR COMPUTADORA

Uno de los mayores usos de los gráficos por computadora se encuentra en los procesos de diseño, particularmente en arquitectura e ingeniería, aunque ahora muchos productos se diseñan por computadora. Generalmente, se conoce como (CAD, Computer Aided Design, diseño asistido por computadora) o CADD (Computer Aided Drafting and Design). Estos métodos se emplean rutinariamente en el diseño de edificios, automóviles, aeronaves, barcos, naves espaciales, computadoras, telas, electrodomésticos y muchos otros productos.

En algunas aplicaciones de diseño, los objetos se visualizan primero en su modelo alámbrico mostrando su forma general y sus características internas. El modelo alámbrico permite a los diseñadores ver rápidamente los efectos de los ajustes interactivos que se hacen en las formas sin esperar a que la superficie completa de los objetos esté completamente generada. Las Figuras 1.7 y 1.8 proporcionan ejemplos de modelos alámbricos en aplicaciones de diseño.



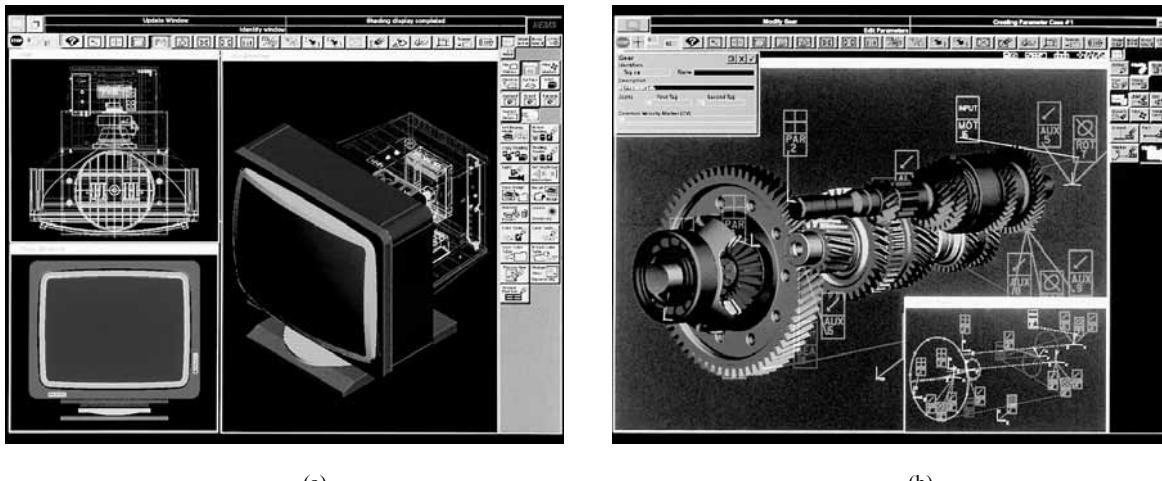
**FIGURA 1.7.** Modelo alámbrico con código de colores de ensamblado de una rueda de automóvil. (Cortesía de Evans y Sutherland.)



(a)

(b)

**FIGURA 1.8.** Modelos alámbricos con código de colores del diseño del cuerpo de un avión y un automóvil. (Cortesía de (a) Corporación Peritek y (b) Evans y Sutherland.)



(a)

(b)

**FIGURA 1.9.** Estaciones de trabajo CAD con múltiples ventanas. (Cortesía de Intergraph Corporation.)

Los paquetes de software CAD suelen proporcionar al diseñador un entorno multiventana, como se ve en las Figuras 1.9 y 1.10. Las diferentes ventanas pueden mostrar secciones aumentadas o diferentes vistas de los objetos.

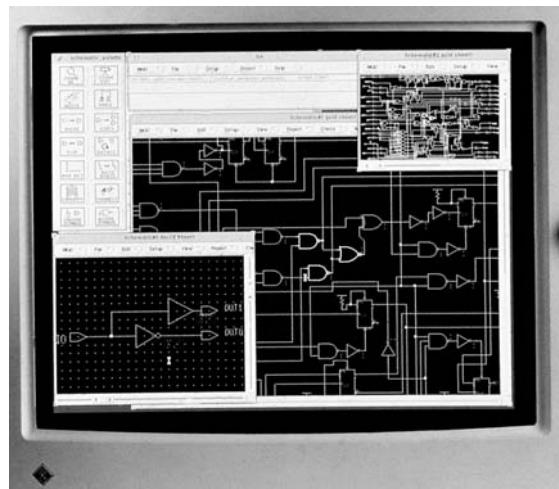
Circuitos como el mostrado en la Figura 1.10 y redes de comunicación, de suministro de agua y otras herramientas se construyen colocando de forma repetida unas pocas formas gráficas. Las formas empleadas en un diseño representan los diferentes componentes del circuito o de la red. Los paquetes de diseño suelen suministrar formas estándar de mecánica, electricidad, electrónica y circuitos lógicos. Para otro tipo de aplicaciones, el diseñador puede crear símbolos personalizados para construir la red o el circuito. El sistema es entonces diseñado mediante sucesivas copias de los componentes posicionadas en el plano y unidas automáticamente entre sí mediante enlaces proporcionados por el paquete gráfico. Esto permite que el diseñador pueda probar diferentes configuraciones para el circuito y así optimizar al mínimo el número de componentes usados o el espacio requerido para el sistema.

Las animaciones también se usan frecuentemente en las aplicaciones de CAD. Animaciones en tiempo real del modelo alámbrico de las formas son muy útiles para comprobar rápidamente el funcionamiento de un vehículo o un sistema como se puede ver en la Figura 1.11.

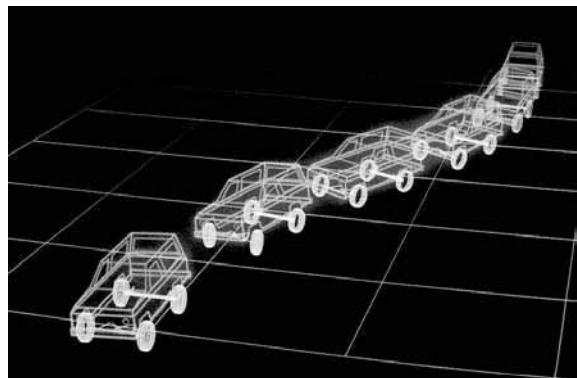
Dado que las imágenes en modelo alámbrico no muestran las superficies, los cálculos para cada segmento de animación pueden realizarse rápido para así producir movimientos suaves en la pantalla. También el modelo alámbrico permite ver en el interior del vehículo y observar los componentes internos durante el movimiento.

Cuando los diseños del objeto están completos o casi completos, se aplican condiciones reales de iluminación y de representación de superficies para producir visualizaciones que mostrarán la apariencia final del producto. Ejemplos de esto se proporcionan en la Figura 1.12. También se generan visualizaciones realistas para anunciar automóviles y otros vehículos usando efectos de luz y escenas de fondo (Figura 1.13).

El proceso de fabricación también va unido a la descripción computerizada de los objetos diseñados, de modo que el proceso de fabricación del producto puede ser automatizado, utilizando métodos que son conocidos como CAM (Computer-Aided Manufacturing, fabricación asistida por computadora). El plano de una placa de circuito, por ejemplo, puede transformarse en la descripción individualizada de los procesos necesarios para construir el circuito electrónico. Algunas piezas mecánicas se fabrican a partir de las descripciones de cómo las superficies se tienen que formar con las máquinas herramienta. La Figura 1.14 muestra las rutas



**FIGURA 1.10.** Aplicación para el diseño de circuitos electrónicos, usando múltiples ventanas y componentes lógicos codificados mediante colores. (Cortesía de Sun Microsystems.)



**FIGURA 1.11.** Simulación del comportamiento de un vehículo al cambiar de carril. (Cortesía de Evans & Sutherland y Mechanical Dynamics, Inc.)



(a)

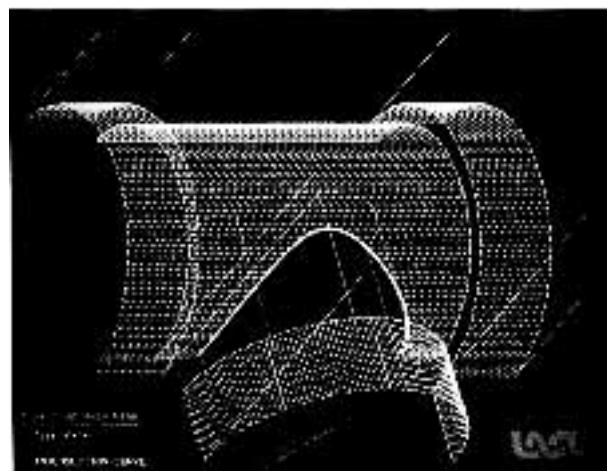


(b)

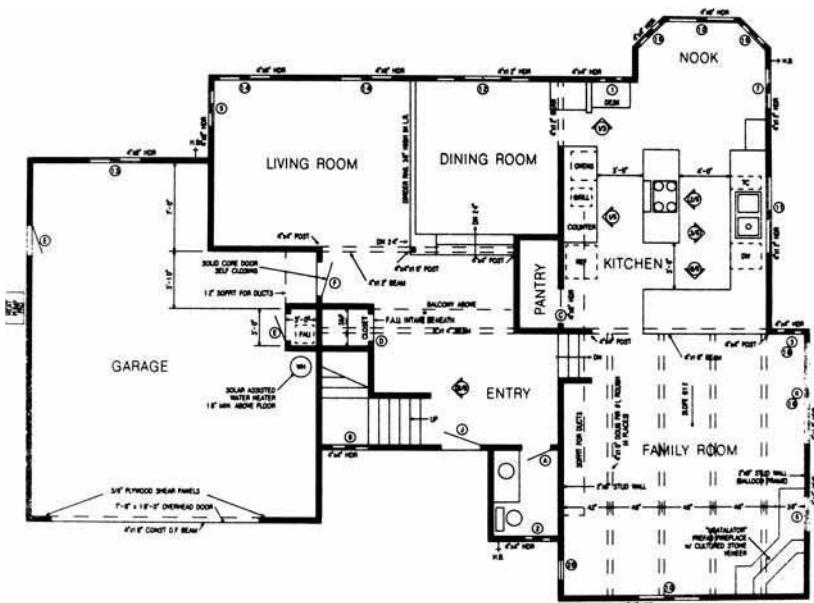
**FIGURA 1.12.** Presentaciones realistas de diseños de ingeniería. (Cortesía de (a) Intergraph Corporation (b) Evans & Sutherland.)



**FIGURA 1.13.** Mediante software gráfico se aplican efectos de iluminación de estudio y técnicas de visualización realista de superficies, con el fin de crear anuncios de productos terminados. Esta imagen generada por computadora de un Chrysler Laser fue generada a partir de datos proporcionados por la Compañía Chrysler. (*Cortesía de Eric Haines, Autodesk Inc.*)



**FIGURA 1.14.** Un esquema de CAD para describir el control numérico del mecanizado de una pieza. La superficie de la pieza está en un tono de gris y las trayectorias de la herramienta en otro. (*Cortesía de Los Alamos National Laboratory.*)



**FIGURA 1.15.** Esquema CAD de arquitectura para el diseño de un edificio. (*Cortesía de Precision Visuals, Inc Boulder, Colorado.*)

que tienen que tomar las máquinas sobre las superficies de un objeto durante su construcción. Las máquinas con control numérico preparan la fabricación de acuerdo con estos diseños de fabricación.

Los arquitectos usan métodos de gráficos interactivos para diseñar los planos de distribución de pisos, como se muestra en la Figura 1.15, en los que se indica la posición de las habitaciones, puertas, ventanas, escaleras, estanterías, encimeras y otras características del edificio. Trabajando a partir de la visualización en un monitor del plano de un edificio, un diseñador eléctrico puede definir el cableado, los enchufes eléctricos y los sistemas de prevención de incendios. También con un paquete de planificación de instalaciones se puede optimizar el espacio en una oficina o en una fábrica.

Vistas realistas de diseños arquitectónicos, como el de la Figura 1.16 permiten tanto a los arquitectos como a sus clientes estudiar la apariencia de un edificio o de un conjunto de edificios, como un campus o un complejo industrial. Además para visualizaciones realistas de exteriores de edificios, los paquetes de arquitectura CAD proporcionan utilidades que permiten experimentar con planos interiores tridimensionales y de iluminación (Figura 1.17).

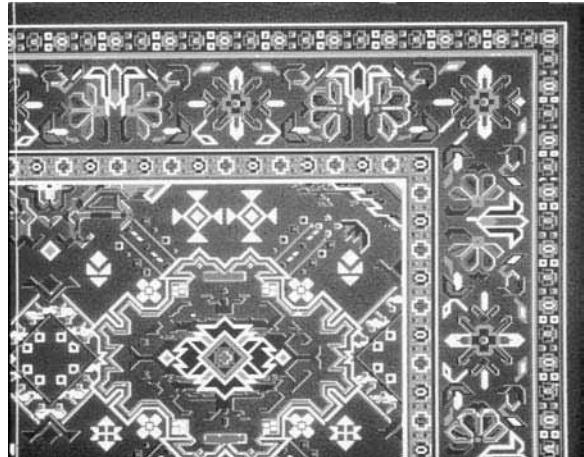
Pueden diseñarse muchos otros tipos de sistemas y productos usando tanto paquetes generales de CAD como desarrollos especiales de software de CAD. La Figura 1.18, por ejemplo, muestra un patrón de alfombra diseñado con un sistema CAD.



**FIGURA 1.16.** Presentaciones realistas tridimensionales de diseños de edificios. (a) Una perspectiva a nivel de calle del proyecto de un centro de negocios. (*Cortesía de Skidmore, Owings & Merrill*). (b) Visualización arquitectónica de un atrio creado para una animación por computadora por Marialine Prieur, Lyon, Francia. (*Cortesía de Thomson Digital Image, Inc*)



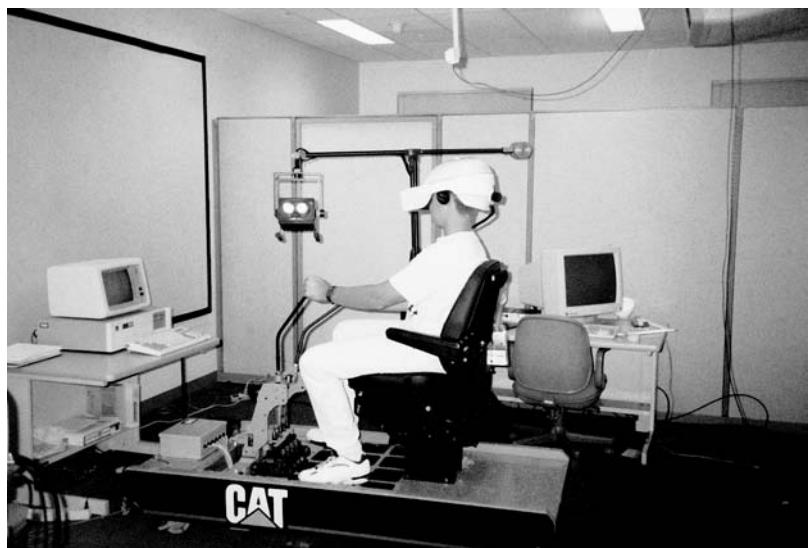
**FIGURA 1.17.** Un pasillo de un hotel que proporciona la sensación de movimiento mediante el posicionamiento de lámparas a lo largo de una trayectoria ondulada y crea la sensación de entrada posicionando una torre de luz a la entrada de cada habitación. (*Cortesía de Skidmore, Owings & Merrill*.)



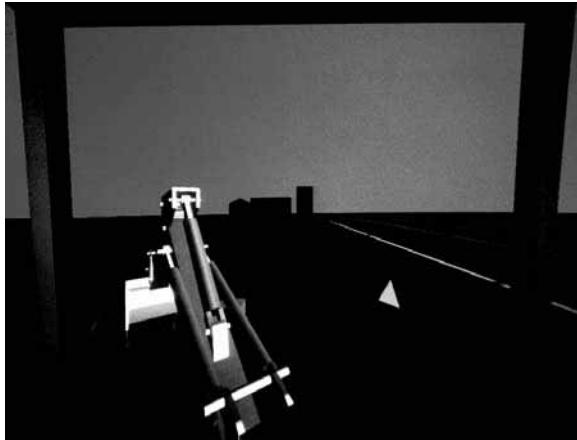
**FIGURA 1.18.** Patrón de alfombra oriental creado con métodos de diseño con gráficos por computadora. (Cortesía de la Lexidata Corporation).

## 1.3 ENTORNOS DE REALIDAD VIRTUAL

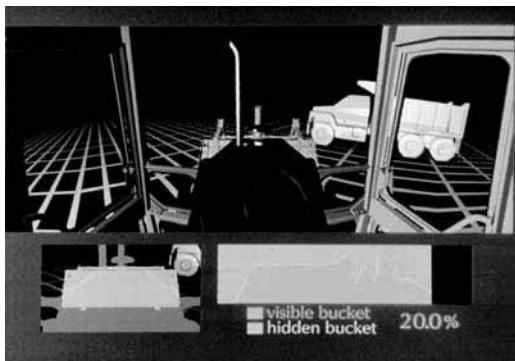
Una aplicación más reciente de los gráficos por computadora es la creación de los entornos de realidad virtual en los que el usuario puede interactuar con los objetos en una escena tridimensional. Dispositivos hardware especializados proporcionan efectos de visión tridimensional y permiten al usuario tomar los objetos de la escena. Los entornos de realidad virtual animados se usan frecuentemente para formar a los operadores de equipo pesado o para analizar la efectividad de diferentes configuraciones de cabina y localizaciones de control. Mientras el operador del tractor de la Figura 1.19 manipula los controles, el equipo de la cabeza presenta una visión estereoscópica (Figura 1.20) de la pala delantera o del cazo trasero, como si el operador estuviera en el asiento del tractor. Esto permite al diseñador explorar varias posiciones para la pala o el cazo trasero que pudieran entorpecer la visión del operador, lo que puede tenerse en cuenta para el diseño global del tractor. La Figura 1.21 muestra una vista compuesta con un ángulo de visión amplio desde el asiento del tractor, visualizada en un monitor estándar de video en lugar de una escena tridimensional virtual. La Figura 1.22 muestra una vista del tractor que puede visualizarse en una ventana separada o en otro monitor.



**FIGURA 1.19.** Prácticas con un tractor en un entorno de realidad virtual. Cuando los controles se mueven el operador ve la pala delantera, el cazo trasero y los alrededores a través del equipo de la cabeza. (Cortesía de National Center for Supercomputing Applications, Universidad de Illinois en Urbana-Champaign y Caterpillar, Inc.)



**FIGURA 1.20.** Una vista en el visiocasco del cazo trasero presentada al operador del tractor en un entorno de realidad virtual. (Cortesía de National Center for Supercomputing Applications, Universidad de Illinois en Urbana-Champaign y Caterpillar, Inc.)



**FIGURA 1.21.** Vista del operador de la pala delanteira compuesta de varias secciones para generar una vista de gran angular en un monitor estandar. (Cortesía de National Center for Supercomputing Applications, Universidad de Illinois en Urbana-Champaign y Caterpillar, Inc.)



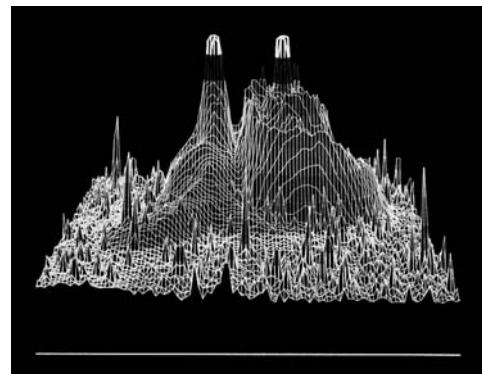
**FIGURA 1.22.** Vista del tractor en un monitor estandar (Cortesía de National Center for Supercomputing Applications, Universidad de Illinois en Urbana-Champaign y Caterpillar, Inc.)

Con los sistemas de realidad virtual, tanto los diseñadores como los demás pueden moverse alrededor e interactuar con los objetos de diferentes maneras. Los diseños arquitectónicos pueden examinarse mediante un paseo simulado a través de habitaciones o alrededor de los exteriores del edificio para poder apreciar mejor el efecto global de un diseño particular. Mediante un guante especial, podemos, incluso agarrar objetos en una escena y girarlos o moverlos de un sitio a otro.

## 1.4 VISUALIZACIÓN DE DATOS

La generación de representaciones gráficas de conjuntos de datos o procesos de naturaleza científica, de ingeniería o de medicina es otra nueva aplicación de los gráficos por computadora. Generalmente, esto se conoce como **visualización científica**. Y el término **visualización de negocios** se usa para conjuntos de datos relacionados con el comercio, la industria y otras áreas no científicas.

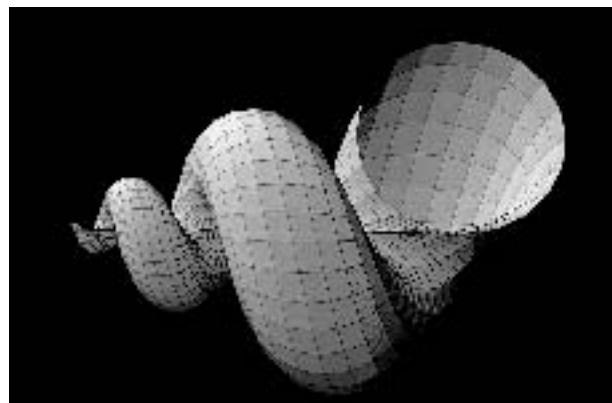
Investigadores, analistas y demás, frecuentemente necesitan tratar con grandes cantidades de información o estudiar el comportamiento de procesos de elevada complejidad. Las simulaciones numéricas por computadora, por ejemplo, normalmente, producen grandes cantidades de archivos de datos que contienen miles o incluso millones de valores. De modo similar, las cámaras de un satélite u otras fuentes de grabación acumulan archivos de datos más rápido de lo que pueden ser interpretados. Rastrear estos grandes conjuntos de números para determinar tendencias y relaciones es un proceso tedioso y totalmente ineffectivo. Sin embargo, si los datos se convierten a una formato gráfico, las tendencias y relaciones aparecen inmediatamente. La Figura 1.23 muestra un ejemplo de un gran conjunto de valores que han sido convertidos en una visualización codificada mediante colores de alturas relativas sobre un plano de tierra. Una vez que hemos representado los valores de densidad en esta forma, podemos ver el patrón global de los datos.



**FIGURA 1.23.** Diagrama que usa un código de colores con 16 millones de puntos de densidad de brillo relativo observado de la Nebulosa Whirlpool donde se revelan dos galaxias distintas. (Cortesía de Los Alamos National Laboratory)



**FIGURA 1.24.** Representación de funciones de curvas matemáticas en varias combinaciones de colores (en escala de grises en la imagen). (Cortesía de Melvin L. Prueitt, Los Alamos National Laboratory)



**FIGURA 1.25.** Para producir esta función tridimensional se utilizaron efectos de iluminación y técnicas de representación de superficies. (Cortesía de Wolfram Research Inc., el creador de Mathematica)

Existen muchas clases distintas de conjuntos de datos, por lo que los esquemas de visualización efectivos dependen de las características de los datos. Una colección de datos puede contener valores escalares, vectores, tensores de orden superior o cualquier combinación de estos tipos de datos. Los conjuntos de datos pueden estar distribuidos sobre una región bidimensional en el espacio, una región tridimensional o un espacio de dimensión superior. La codificación mediante colores es una manera de visualizar un conjunto de datos. Otras técnicas de visualización incluyen la representación de perfiles, la representación de superficies de valor constante u otras regiones del espacio y formas especialmente diseñadas para la representación de diferentes tipos de datos.

Las técnicas de visualización también se usan como ayuda para la comprensión de procesos complejos y funciones matemáticas. Una representación en color (en escala de grises en la figura) de una función matemática se muestra en la Figura 1.24. En la Figura 1.25 se puede ver la representación de una superficie. El objeto de la Figura 1.26 se generó mediante procedimientos fractales usando cuaternios. En la Figura 1.27 se muestra una estructura topológica.

Los científicos están desarrollando métodos para la visualización de datos de carácter genérico. La Figura 1.28 muestra una técnica genérica para la representación y el modelado de datos distribuidos sobre una superficie esférica.



**FIGURA 1.26.** Un objeto de cuatro dimensiones proyectado en un espacio de tres dimensiones y representado en una pantalla de video de dos dimensiones, con código de colores. El objeto se generó utilizando cuaternios y procedimientos cuadráticos de fractales, con un octante substrazado para mostrar la complejidad del conjunto de Julia. (*Cortesía de John C. Hart, Departamento de Ciencias de la Computación, Universidad de Illinois, Urbana-Champaign.*)

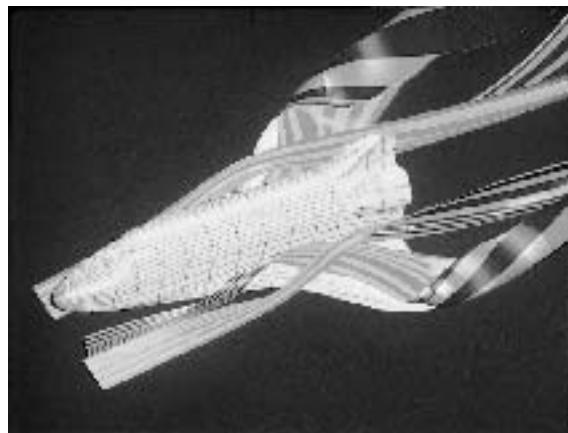


**FIGURA 1.27.** Cuatro vistas en tiempo real de una animación interactiva que estudia las superficies mínimas («snails») en las tres esferas proyectadas en un espacio euclídeo tridimensional. (*Cortesía de George Francis, Departamento de Matemáticas y del National Center for Supercomputing Applications, Universidad de Illinois, Urbana-Champaign. 1993.*)

**FIGURA 1.28.** Un método para realizar representaciones gráficas y modelar datos distribuidos sobre una superficie esférica. (Cortesía de Grieg Nelson, Departamento de Ciencias de la Computación, Universidad del Estado de Arizona.)



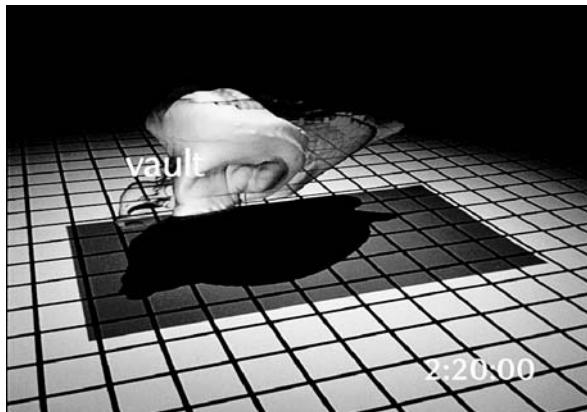
**FIGURA 1.29.** Visualización de superficies de corriente que fluyen a través de una lanzadera espacial. Proporcionado por Jeff Hultquist y Eric Raible, NASA Ames. (Cortesía de Sam Uselton, Nasa Ames Research Center.)



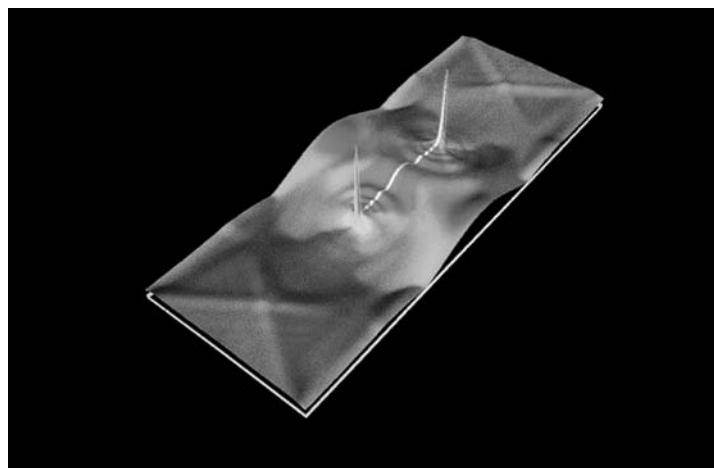
**FIGURA 1.30.** Modelo numérico de las corrientes de aire dentro de una tormenta. (Cortesía de Bob Wilhelmsom, Departamento de Ciencias Atmosféricas y del National Center for Supercomputing Applications, Universidad de Illinois, Urbana-Champaign.)



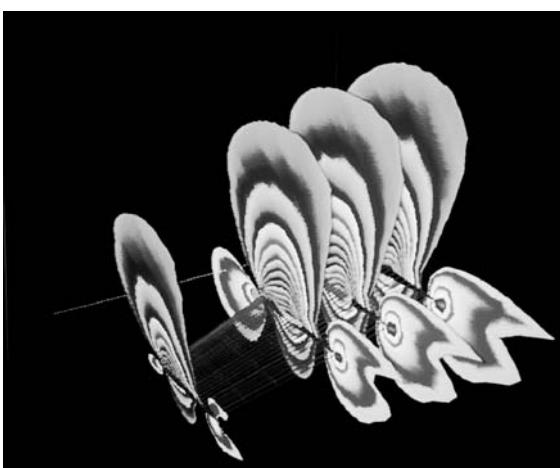
Desde la Figura 1.29 hasta la 1.42 se muestran diversas aplicaciones para la visualización. Estas figuras muestran: corrientes de aire fluyendo sobre la superficie de una lanzadera espacial, modelado numérico de una tormenta, un despliegue de los efectos de la propagación de fracturas en los metales, la representación en código de colores de la densidad de un fluido sobre un perfil aerodinámico, secciones cruzadas de un conjunto de datos, modelado de proteínas, visualización interactiva de estructuras moleculares dentro de un entorno de realidad virtual, un modelo del suelo del océano, una simulación de un incendio en un pozo petrolífero en Kuwait, un estudio de la contaminación del aire, un estudio del crecimiento del maíz, una reconstrucción de las ruinas del Cañón del Chaco en Arizona y un diagrama de las estadísticas de los accidentes de tráfico.



**FIGURA 1.31.** Modelo numérico de la superficie de una tormenta.(Cortesía de Bob Wilhelmson, Departamento de Ciencias Atmosféricas y del National Center for Supercomputing Applications, Universidad de Illinois, Urbana-Champaign.)



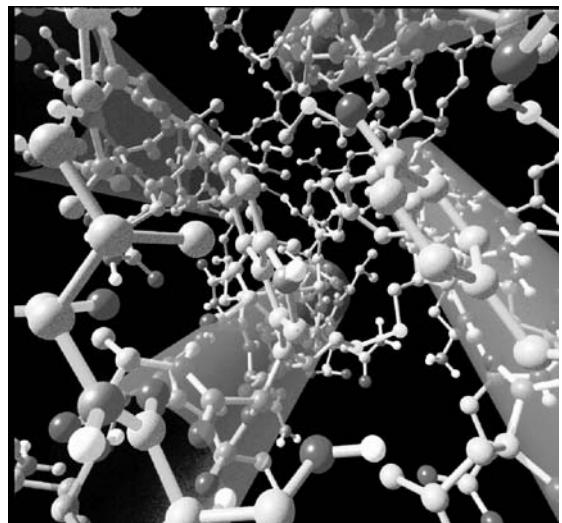
**FIGURA 1.32.** Visualización con código de colores de la densidad de energía de fatiga en el estudio de la propagación de una grieta en una plancha de metal. Modelado por Bob Haber. (Cortesía del National Center for Supercomputing Applications, Universidad de Illinois, Urbana-Champaign.)



**FIGURA 1.33.** Simulación de la dinámica de un fluido, mostrando la representación gráfica con código de colores de la densidad del fluido que se extiende sobre una malla de planos alrededor del ala de un avión, desarrollado por Lee-Hian Quek, John Eickemeyer y Jeffery Tan. (Cortesía de Information Technology Institute, República de Singapur.)



**FIGURA 1.34.** Software cortador de secciones, mostrando valores con código de colores sobre las secciones transversales de un conjunto de datos. (*Cortesía de Spyglass, Inc.*)



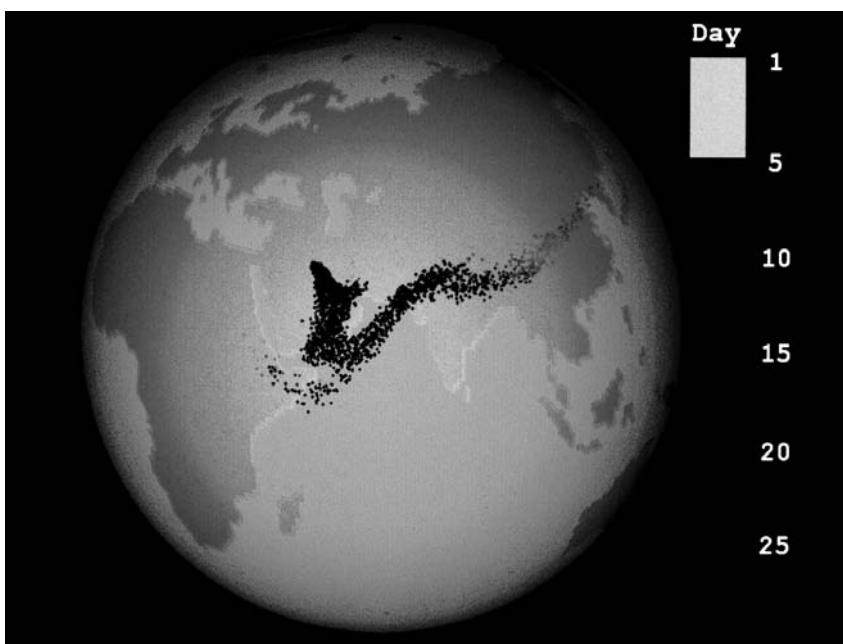
**FIGURA 1.35.** Visualización de la estructura de una proteína creada por Jay Siegel y Kim Baldridge, SDSC. (*Cortesía de Stephanie Sides, Supercomputer Center de San Diego.*)



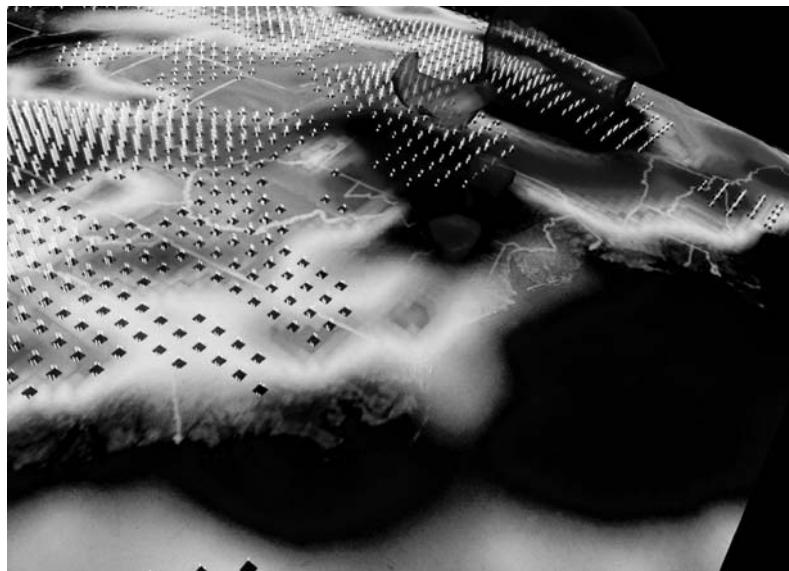
**FIGURA 1.36.** Científico interactuando con vistas esteroscópicas de estructuras moleculares dentro de un entorno de realidad virtual, llamado «CAVE», caverna. (*Cortesía de William Sherman y de National Center for Supercomputing Applications, Universidad de Illinois, Urbana-Champaign.*)



**FIGURA 1.37.** Imagen de un par esteroscópico, que muestra la visualización de suelo oceánico, obtenida de las imágenes de un satélite, creada por David Sandwell y Chris Small, Institución Scripps de la Oceanografía y Jim Mcleod, SDSC. (Cortesía de Stephanie Sides, San Diego Supercomputer Center.)



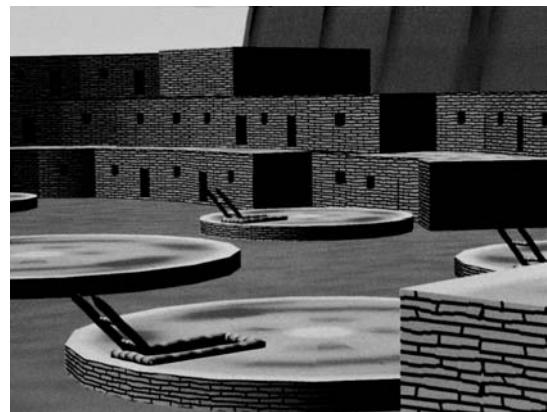
**FIGURA 1.38.** Simulación de los efectos de los incendios en los pozos petrolíferos de Kuwait, desarrollada por Gary Glatzmeier, Chuck Hanson y Paul Hinken. (Cortesía de Mike Krogh, Advanced Computing Laboratory en Los Alamos National Laboratory.)



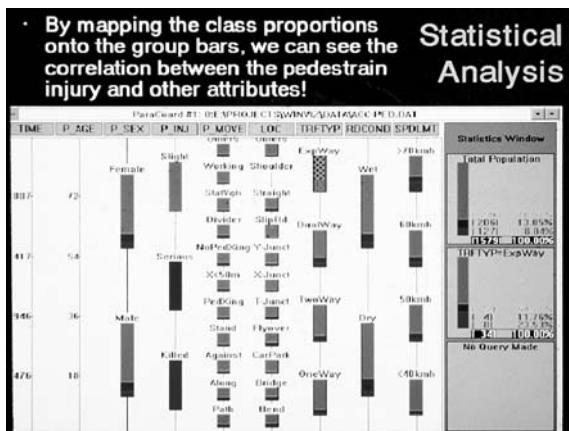
**FIGURA 1.39.** Visualización de la contaminación sobre la superficie de la tierra diseñada por Tom Palmer, Cray Research Inc. /NCSC, Chris Landreth, NCSC, and Dave Bock, NCSC. El contaminante  $\text{SO}_4$  se representa con color azul, la precipitación de la lluvia ácida es un color plano en el mapa de la superficie, y la concentración de lluvia se representa como cilindros huecos. (*Cortesía del Supercomputing Center/ MCNC de Carolina del Norte.*)



**FIGURA 1.40.** Un marco de la animación que muestra el crecimiento de una mazorca de maíz. (*Cortesía de National Center for Supercomputing Applications, Universidad de Illinois, Urbana-Champaign.*)



**FIGURA 1.41.** Visualización de la reconstrucción de las ruinas del Cañón del Chaco en Arizona. (*Cortesía de Melvin L. Prueitt, Los Alamos National Laboratory. Datos proporcionados por Stephen H. Lekson.*)

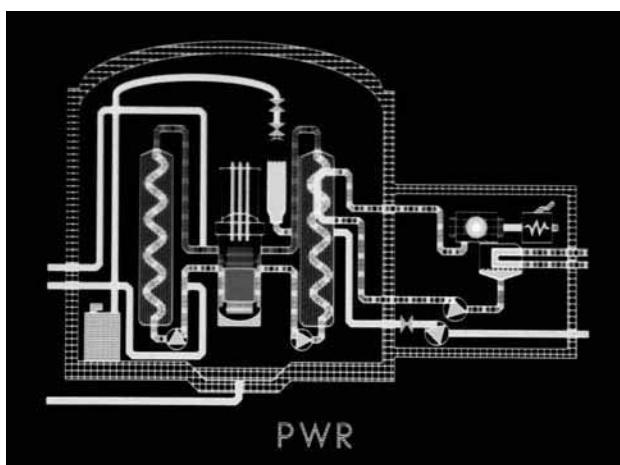


**FIGURA 1.42.** Prototipo para la visualización de tablas multidimensionales de datos, llamado WinViz y desarrollado por el equipo de visualización del Instituto de Tecnologías de la Información, República de Singapur. Se usa aquí para hacer correlaciones de información estadística de peatones implicados en accidentes de tráfico. (Cortesía de Lee-Hean Quek, Oracle Corporation, Redwood Shores, California.)

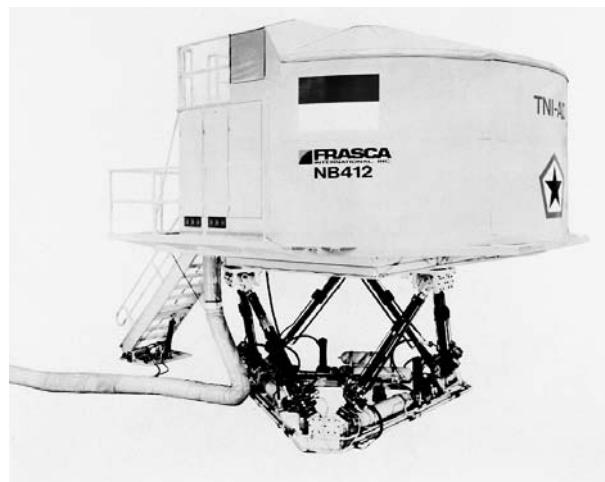
## 1.5 EDUCACIÓN Y FORMACIÓN

Los modelos generados por computadora de sistemas físicos, financieros, políticos, sociales, económicos y otros se usan frecuentemente como ayudas para la educación. Modelos de procesos físicos, psicológicos, tendencias de la población, equipamiento, como el diagrama codificado de colores (en escala de grises en la imagen) de la Figura 1.43, puede ayudar a los alumnos a comprender la operación de un sistema.

En algunas aplicaciones de formación se necesitan equipos especiales de hardware. Ejemplos de este tipo de sistemas son los simuladores para las sesiones prácticas de formación de capitanes de barco, pilotos de aeronaves, operadores de maquinaria pesada, personal del control de tráfico aéreo. Algunos simuladores no tienen pantallas de video, como por ejemplo, un simulador de vuelo con sólo un panel instrumental de vuelo. Pero la mayoría de los simuladores proporcionan pantallas para mostrar visualizaciones del entorno exterior. Dos ejemplos de grandes simuladores con sistemas internos de visualización se muestran en las Figuras 1.44 y 1.45. Otro tipo de sistemas de visualización lo tenemos en las Figuras 1.46(b) y(c). En este caso, se monta una pantalla de visualización enfrente del simulador y proyectores en color muestran la escena del vuelo en paneles de pantallas. La Figura 1.47 muestra la zona que puede estar situada detrás de la cabina del piloto del simulador de vuelo. El teclado lo usa el instructor para introducir parámetros que afectan al funcionamiento del avión o al entorno, así mismo se visualiza el camino de la aeronave y otros datos en los monitores durante la sesión de formación o prueba.



**FIGURA 1.43.** Diagrama con código de colores (escala de grises en la imagen) para explicar el funcionamiento de un reactor nuclear. (Cortesía de Los Alamos National Laboratory.)



**FIGURA 1.44.** Gran simulador de vuelo cerrado con sistema visual a todo color y seis grados de libertad en su movimiento. (*Cortesía de Frasca Internacional.*)

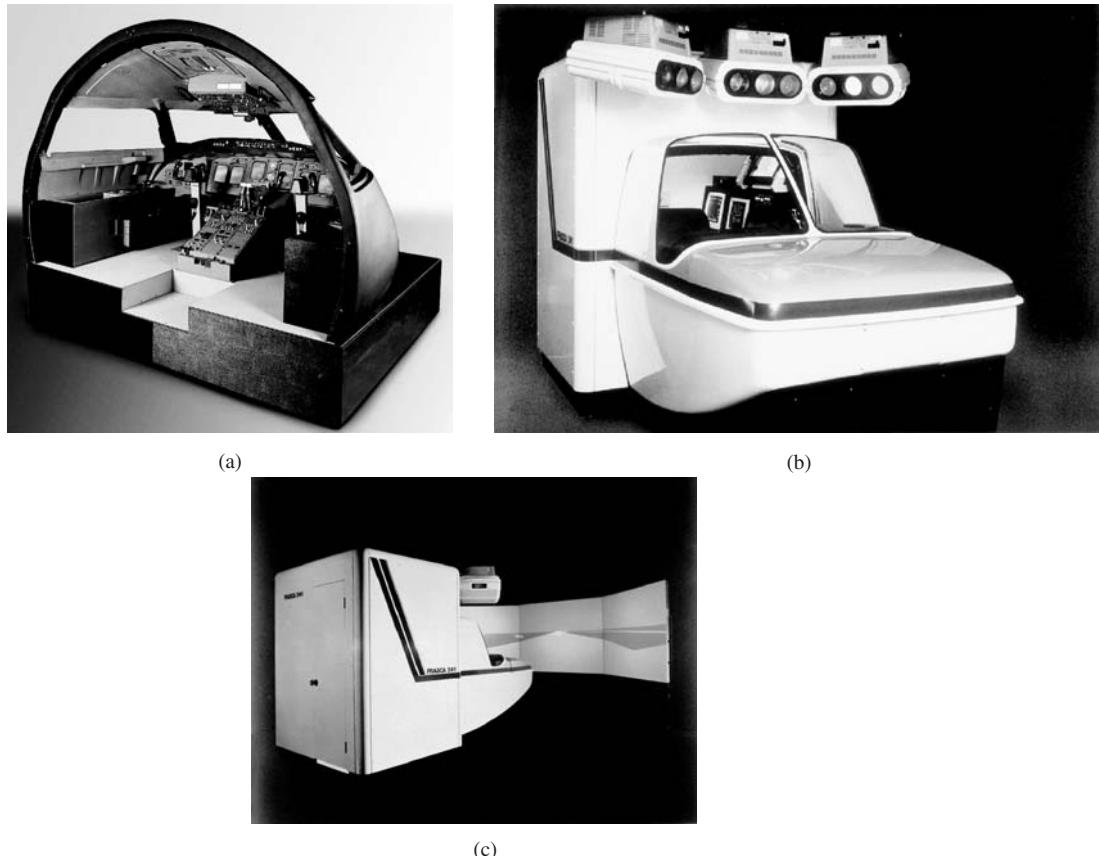


**FIGURA 1.45.** Simulador de tanque militar con sistema visual de imágenes. (*Cortesía de Mediatech y GE Aerospace.*)

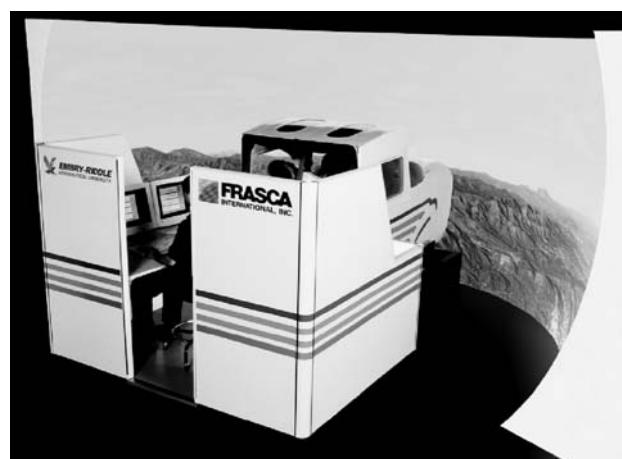
Escenas generadas para simuladores de aeronaves, barcos y naves espaciales se muestran en las Figuras 1.48 hasta la 1.50.

El simulador de un automóvil y sus imágenes asociadas se dan en la Figura 1.51. La parte (a) de esta figura muestra el interior del simulador y la pantalla de visualización visible a través del parabrisas. Una escena típica del tráfico en una calle se muestra en la Figura 1.51(b). A pesar de que los simuladores de automóviles

pueden ser usados como sistemas de entrenamiento, se utilizan comúnmente para estudiar el comportamiento de los conductores ante situaciones críticas. Las reacciones del conductor en diversas condiciones de tráfico pueden ser utilizadas como base para diseñar un vehículo maximizando la seguridad en el tráfico.



**FIGURA 1.46.** Interior de la cabina (a) de un simulador de vuelo con control dual y sistema visual exterior en color completo (b) y (c) para un simulador pequeño de vuelo. (*Cortesía de Frasca International.*)



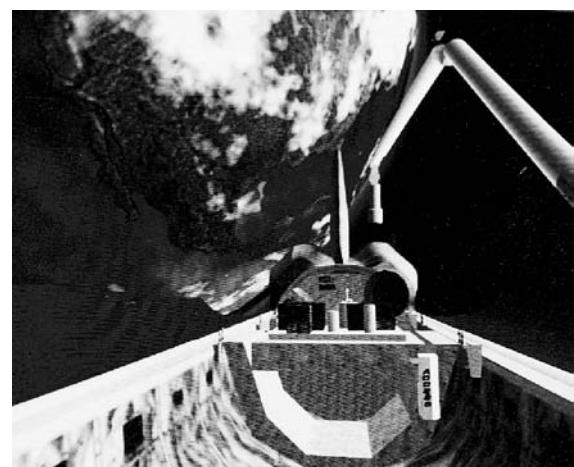
**FIGURA 1.47.** El área del instructor detrás de la cabina de un simulador pequeño de vuelo. El equipo permite al instructor monitorizar las condiciones de vuelo y establecer los parámetros del avión y del entorno. (*Cortesía de Frasca International.*)



**FIGURA 1.48.** Imágenes de un simulador de vuelo. (*Cortesía de Evans & Sutherland.*)



**FIGURA 1.49.** Imágenes generadas para un simulador naval. (*Cortesía de Evans & Sutherland.*)



**FIGURA 1.50.** Imágenes de una lanzadera espacial. (*Cortesía de Mediatech y GE Aerospace.*)



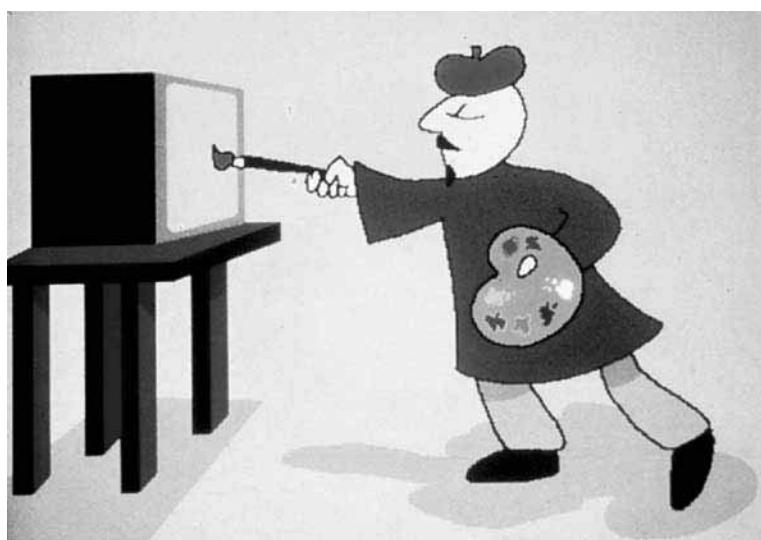
**FIGURA 1.51.** Interior del simulador de una automóvil (a), una vista de una escena de calle (b) puede presentarse al conductor. (Cortesía de Evans & Sutherland.)

## 1.6 ARTE POR COMPUTADORA

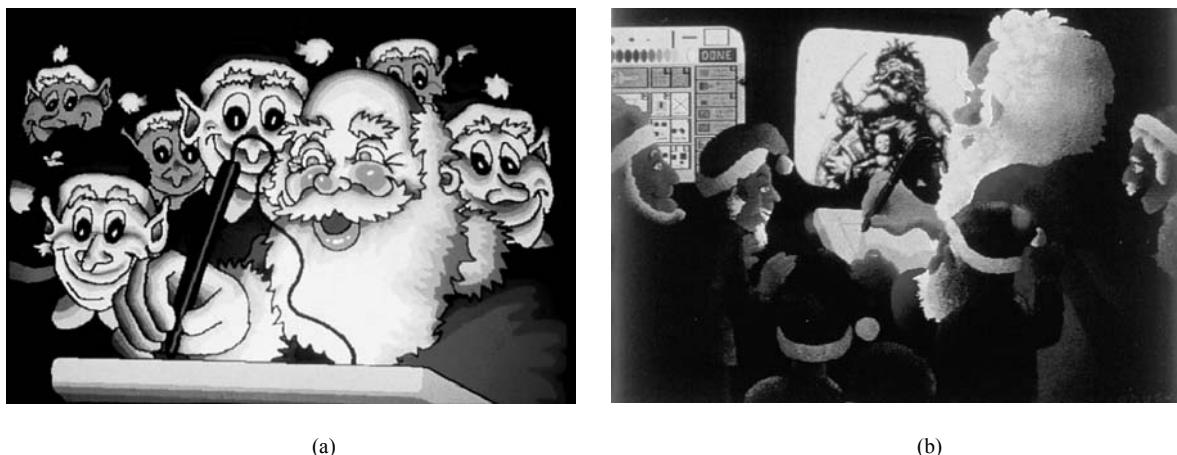
---

Tanto el arte puro como el comercial hacen uso de los métodos de los gráficos por computadora. Los artistas tienen disponibles una variedad de métodos y herramientas, incluyendo hardware especializado, paquetes de software comercial (como Lumena), programas de matemática simbólica (como Mathematica), paquetes CAD, software de escritorio para publicación y sistemas de animación que proporcionan capacidades para diseñar formas de objetos y especificar sus movimientos.

La Figura 1.52 proporciona una representación figurada del uso de un **programa de dibujo** (*paint brush*) que permite al artista «pintar» cuadros o dibujos en la pantalla de un monitor de vídeo. En realidad, el dibujo se pinta electrónicamente en una tableta gráfica digitalizadora utilizando una pluma, que puede simular diferentes golpes de pincel, grosores de pincel y colores. Utilizado un programa de dibujo, un diseñador de comics creó los personajes de la Figura 1.53 quienes parecen estar muy ocupados en la creación de si mismos.



**FIGURA 1.52.** Dibujo animado generado con un programa de dibujo (*paintbrush*), representa de manera simbólica cómo un artista trabaja en un monitor de vídeo. (Cortesía de Gould Inc, Imaging and Graphics Division y Aurora Imaging)



**FIGURA 1.53.** Demostraciones en comic de un «artista» creando un dibujo con un sistema *paintbrush*. En (a) el dibujo se hace en una tableta gráfica mientras ellos mismos observan el desarrollo de la imagen en la pantalla de video. En (b) el artista y ellos mismos están sobreimpuestos en la famosa pintura de Thomas Nast de San Nicolás, la cual fue introducida en el sistema a través de una cámara de video. (*Cortesía de Gould Inc., Imaging & Graphics Division y Aurora Imaging*).

Un sistema de dibujo, con un sistema Wacom inalámbrico y pluma sensible a la presión, se emplearon para producir la pintura electrónica de la Figura 1.54, que simula las pinceladas dadas por Van Gogh. La pluma transforma los cambios de presión en la mano a líneas de ancho variable, diferentes tamaños de pincel y gradaciones de color. La Figura 1.55 muestra una acuarela generada con este tipo de pluma electrónica que permite al artista crear no sólo acuarelas sino también, pastel, óleos, simular efectos de secado, humedad e incluso huellas. La Figura 1.56 proporciona un ejemplo de métodos de dibujo combinados con imágenes escaneadas.

Los creadores de bellas artes utilizan diversas técnicas para crear imágenes por computadora. Para crear dibujos como el que aparece en la Figura 1.57, los artistas utilizan un combinación de técnicas de paquetes de modelado tridimensional, mapeo de texturas, programas de dibujo y software CAD. En la Figura 1.58 tenemos una pintura generada con un trazador usando un software diseñado a medida que puede crear «arte automático», sin la intervención de ningún artista.

La Figura 1.59 muestra un ejemplo de arte «matemático». Este artista utiliza una combinación de funciones matemáticas y procedimientos fractales, el software Mathematica, impresoras de inyección de tinta y otros sistemas para crear diversos objetos tridimensionales y formas bidimensionales, así como pares de imágenes estereoscópicas. En la Figura 1.60 se muestra otro ejemplo de arte electrónico creado con la ayuda de rela-



**FIGURA 1.54.** Cuadro parecido a un Van Gogh creado por la artista gráfica Elizabeth O'Rourke con una pluma inalámbrica sensible a la presión. (*Cortesía de Wacom Technology Corporation*.)



**FIGURA 1.55.** Acuarela electrónica, pintada por John Derry de Time Arts, Inc. con pluma inalámbrica sensible a la presión y el software Lumena con pincel aguado. (Cortesía de Wacom Technology Corporation.)

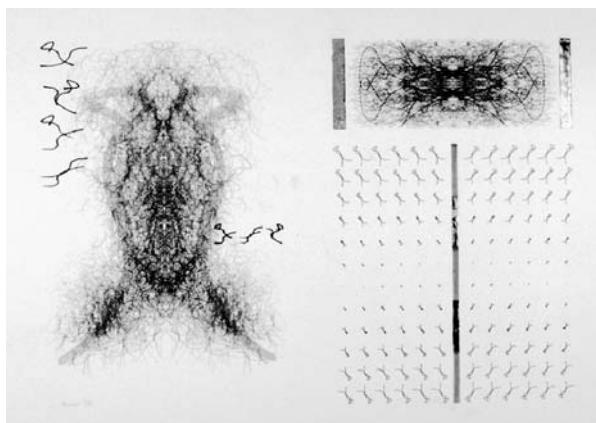


**FIGURA 1.56.** El creador de esta pintura, titulada *Avalancha Electrónica* realiza una afirmación sobre nuestra relación con la tecnología, utilizando una computadora personal con tableta gráfica y el software Lumena para combinar la generación de gráficos para hojas, pétalos de flor y componentes electrónicos con imágenes escaneadas. (Cortesía de la Williams Gallery. © 1991 Trukenbrod, The School of the Art Institute of Chicago.)



**FIGURA 1.57.** De una serie llamada «Esferas de influencia», esta pintura electrónica titulada *Whigmalaree* fue creada con una combinación de métodos utilizando una tableta gráfica, modelado tridimensional, mapeado de texturas y una serie de transformaciones geométricas. (Cortesía de la Williams Gallery. © 1992 Wynne Ragland, Jr.)

**FIGURA 1.58.** Producción de arte electrónico con un trazador y software diseñado específicamente para el artista para imitar su estilo. El trazador tiene múltiples plumas e instrumentos de pintura incluyendo pinceles chinos. (Cortesía de la Williams Gallery © Roman Verostko, Minneapolis College of Art & Design.)



**FIGURA 1.59.** Esta creación de Andrew Hanson está basada en una visualización del último teorema de Fermat,  $x^n + y^n = z^n$ ,  $n = 5$ . Departamento de Ciencias de la Computación de la Universidad de Indiana. La imagen fue renderizada con Mathematica y Software Wavefront. (Cortesía de la Williams Gallery. © 1991 Stewart Dickinson.)

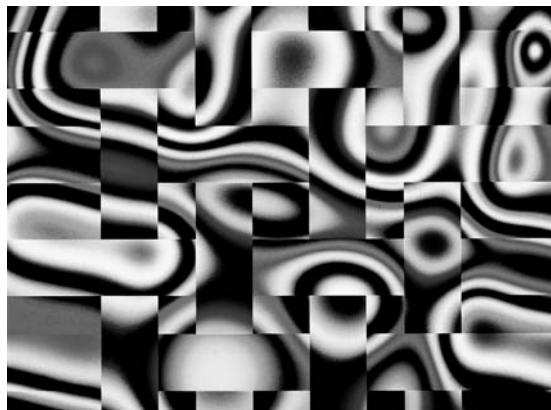


ciones matemáticas. La propuesta artística de este creador frecuentemente se diseña en relación con las variaciones de frecuencia y otros parámetros de una composición musical para producir vídeo que integra patrones visuales y auditivos.

Las artes gráficas también utilizan estas técnicas de «pintura» para generar logos y otros diseños, diseños de páginas combinando textos y gráficos, anuncios de televisión y otras aplicaciones. En la Figura 1.61 se muestra una estación de trabajo para diseñar páginas que combinan textos y gráficos.

Como en muchas otras aplicaciones de los gráficos por computadora, el arte comercial frecuentemente emplea técnicas fotorrealistas para presentar imágenes de un diseño, producto o escena. La Figura 1.62 muestra el ejemplo del diseño de un logotipo tridimensional, la Figura 1.63 presenta tres imágenes generadas por computadora para el anuncio de un producto.

Las animaciones generadas en una computadora se utilizan frecuentemente en la producción de los anuncios de televisión. Estos anuncios son generados, cuadro a cuadro, y cada cuadro se visualiza y almacena como un archivo de imagen. En cada sucesivo cuadro, las posiciones de cada objeto son ligeramente desplazadas para simular los movimientos que tienen lugar en la animación. Cuando todos los cuadros que participan en la secuencia de animación se han renderizado, se transfieren a una película o se almacenan en un búfer de video para su reproducción. Las películas de animación requieren 24 cuadros por cada segundo de la secuencia. Si la animación se va reproducir en un monitor de vídeo, por lo menos se necesitan 30 cuadros por segundo.



**FIGURA 1.60.** Usando funciones matemáticas, procedimientos fractales, supercomputadoras, este compositor-artista experimenta con varios diseños para sintetizar formas y colores con composición musical. (*Cortesía de Brian Evans, Vanderbilt University.*)

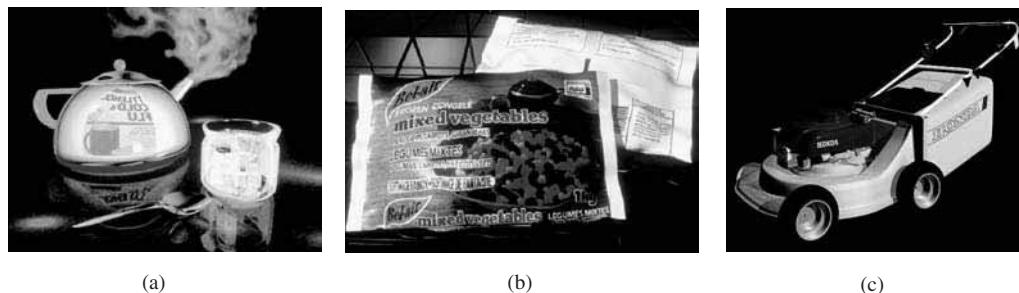


**FIGURA 1.61.** Estación de trabajo para composición de páginas. (*Cortesía de Visual Technology.*)



**FIGURA 1.62.** Presentación tridimensional de un logotipo. (*Cortesía de Vertigo Technology, Inc.*)

Un método gráfico muy empleado en muchos anuncios de televisión es el morfismo (*morfing*), donde un objeto se transforma (metamorfiza) en otro. Este método se utiliza en anuncios publicitarios para transformar aceite en un motor de automóvil, un automóvil en un tigre, un charco de agua en un neumático y la cara de una persona en la de otra. Un ejemplo de morfismo lo tenemos en la siguiente sección, Figura 1.69.



**FIGURA 1.63.** Publicidad de productos mediante imágenes generadas por computadora. (Cortesía de (a) Audrey Fleisher y (b) y (c) de SOFTIMAGE, Inc.)

## 1.7 ENTRETENIMIENTO

---

Las producciones de televisión, las películas de cine y los videos musicales usan de manera rutinaria los gráficos por computadora. Algunas veces estas imágenes se combinan con actores reales y escenas filmadas, a veces, toda la película está generada mediante renderización por computadora y técnicas de animación.

Muchas series de televisión utilizan métodos para producir efectos especiales basados en los gráficos por computadora, como en la Figura 1.64, de la serie «*Deep Space Nine*». Algunos programas de televisión utilizan técnicas de animación para combinar figuras de personas, animales, o personajes de dibujos animados, generados por computadora con actores reales, también se transforma la cara de un actor en otra forma. Muchos programas también utilizan los gráficos por computadora para generar edificios, rasgos del terreno u otros fondos de escenas. La Figura 1.65 muestra una vista muy realista generada por computadora del Dadu (hoy Pekín) en el siglo trece para una emisión japonesa de televisión.

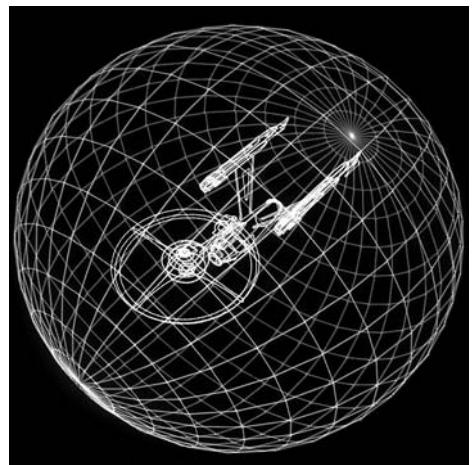
Los efectos especiales, las animaciones, personajes y escenas generadas por computadora están ampliamente extendidas en las películas de hoy en día. La Figura 1.66 ilustra la escena preliminar generada por computadora de la película «*Star Trek la furia del Khan*». Los métodos de renderizado se aplican a los modelos



**FIGURA 1.64.** Escena gráfica de la serie de TV *Deep Space Nine*. (Cortesía of Rhythm & Hues Studios.)



**FIGURA 1.65.** Imagen de una reconstrucción generada por computadora del Dadu (hoy Pekín) en el siglo trece para una emisión japonesa de televisión, de Taisei Corporation (Tokio, Japón) y renderizado con el software TDI. (Cortesía de Thomson Digital Image, Inc.)



**FIGURA 1.66.** Gráficos generados para la película de Paramount Pictures «Star Trek la furia del Khan». (Cortesía de Evans and Sutherland.)



(a)



(b)

**FIGURA 1.67.** Escenas de película generadas por computadora: (a) *El sueño de Red*, (Cortesía de Pixar) Copyright © Pixar 1987. (b) *Knickknack*, (Cortesía de Pixar. Copyright © Pixar 1989.)

alámbricos del planeta y de la nave espacial, para producir el aspecto final con el que aparecen los objetos en la película. Para producir las escenas de las dos películas, ganadoras de premios, que se muestran en la Figura 1.67 se utilizaron técnicas avanzadas de modelado y renderización de superficies. Otras películas emplean modelado, renderizado y animación para producir por completo personajes con aspecto humano. Para dar a los actores generados por computadora, tonos de piel humanos, rasgos realistas en las caras, e imperfecciones en la piel como, lunares, pecas o acné se emplean técnicas fotorrealistas. La Figura 1.68 muestra una escena de la película «*Final Fantasy: The Spirits Within*» en la que se emplean estas técnicas fotorrealistas para simular de manera cercana la apariencia de un actor humano.



**FIGURA 1.68.** Una escena de la película «*Final Fantasy: The Spirits Within*» mostrando tres de los personajes animados del reparto Dr. Aki Ross, Gray Edwards y Dr. Sid. (Cortesía de Square Pictures, Inc. © 2001 FFFP. Reservados todos los derechos.)



**FIGURA 1.69.** Ejemplos de morfismo en el video de David Byrne «*She is mad*». Cortesía de David Byrne, Index Video, y Pacific Data Images.

También se emplean métodos de gráficos por computadora para simular actores humanos. Utilizando archivos de los rasgos faciales de un actor y un programa de animación se pueden generar secuencias de película que contengan réplicas generadas por computadora de esa persona. En el hipotético caso de que el actor enferme o se accidente durante el rodaje, estos métodos de simulación se pueden utilizar para sustituirlo en las subsiguientes escenas.

Los videos musicales utilizan los gráficos por computadora de diversas formas. Se pueden combinar objetos gráficos con acción real, o emplearse gráficos y técnicas de procesamiento de imágenes para transformar

una persona en un objeto o viceversa (morfismo). Un ejemplo de morfismo lo tenemos en la secuencia de escenas de la Figura 1.69 producidas para el vídeo de David Byrne, «*She is mad*».

## 1.8 PROCESAMIENTO DE IMÁGENES

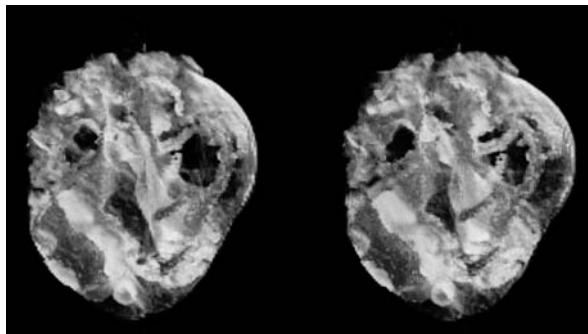
La modificación o interpretación de imágenes existentes, como fotografías o cámaras de TV es conocida como **procesamiento de imágenes**. Aunque los métodos empleados en los gráficos por computadora y el procesamiento de imágenes se solapan, las dos áreas están dedicadas a operaciones fundamentales diferentes. En los gráficos por computadora, la computadora se utiliza para crear una imagen. Por otra parte las técnicas de procesamiento de imágenes se utilizan para mejorar la calidad de un dibujo, analizar las imágenes o reconocer patrones visuales para aplicaciones robotizadas. Sin embargo, los métodos de procesamiento de imágenes se utilizan frecuentemente en los gráficos por computadora, y los métodos de los gráficos por computadora se aplican también en el procesamiento de imágenes.

Por lo general, antes de procesar una imagen o una fotografía, primero se almacena en un archivo de imagen. Entonces es cuando se pueden aplicar los métodos digitales de reorganización de las partes de la imagen, para resaltar separaciones de color, o mejorar la calidad del sombreado. En la Figura 1.70 se da un ejemplo de los métodos de procesamiento de imagen para el realce de la calidad de una imagen. Estas técnicas se usan de manera amplia en las aplicaciones de arte comercial que se implican en el retoque y reestructuración de secciones de fotografías y otras obras artísticas. Técnicas similares se utilizan para analizar las fotografías de la tierra tomadas por un satélite y las grabaciones de distribuciones de estrellas en las galaxias hechas por un telescopio.

Las aplicaciones médicas también hacen uso de las técnicas de procesamiento de imágenes para el realce de tomografías y simulaciones en operaciones quirúrgicas. La tomografía es un tipo de fotografía de rayos X que permite mostrar vistas de secciones transversales de sistemas fisiológicos. *Tomografía Computerizada por Rayos X* (TC), *Tomografía Emisiva de Posición* (TEP) y *Tomografía Axial Computerizada* (TAC) utilizan métodos computacionales para reconstruir secciones cruzadas a partir de datos digitales. Estas técnicas se utilizan para monitorizar funciones internas y mostrar secciones durante las intervenciones quirúrgicas. Otras técnicas de imágenes médicas incluyen ultrasonidos y escáneres médicos nucleares. Con los ultrasonidos, se usan ondas de sonido de alta frecuencia en lugar de rayos X para generar los datos digitales.



**FIGURA 1.70.** Una fotografía borrosa de una placa de matrícula de automóvil se transforma en legible tras la aplicación de técnicas de procesamiento de imágenes. (Cortesía de Los Alamos National Laboratory.)



**FIGURA 1.71.** Cuadro de una animación en la que se visualiza niveles de activación cardiaca dentro de regiones de volúmenes semitransparentes del corazón de un perro. Datos médicos proporcionados por William Smith, Ed Simpson, and G Allan Johnson, Duke University. Software de renderización de imágenes proporcionado por Tom Palmer, Cray Research, Inc./NCSC. (Cortesía de Dave Bock, Supercomputing Center/MCNC de Carolina del Norte.)

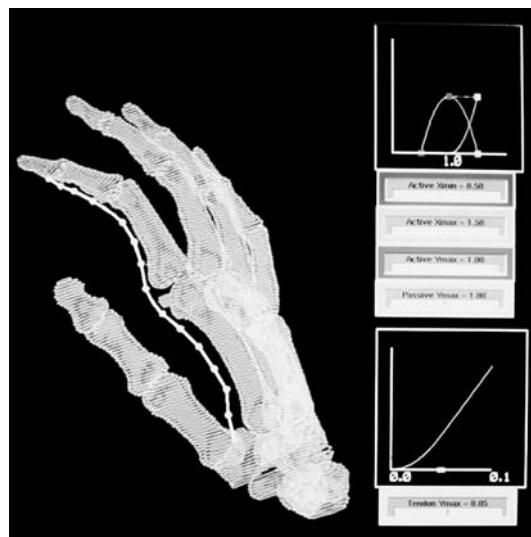
Los escáneres de medicina nuclear recopilan datos de la radiación emitida por radionucleoideos ingeridos y los datos son presentados como imágenes con código de colores.

El procesamiento de imágenes y los gráficos por computadora son frecuentemente utilizados en aplicaciones médicas para modelar y estudiar funciones físicas, para diseñar extremidades artificiales y para planificar y practicar técnicas quirúrgicas. Esta última aplicación es conocida como CAS (Computer Aid Surgery, cirugía asistida por computadora). Utilizando las técnicas de procesamiento de imágenes se pueden obtener secciones transversales en dos dimensiones del cuerpo. Estas secciones del cuerpo se manipulan utilizando modelos gráficos para simular procedimientos quirúrgicos reales e intentar diferentes cortes. En las Figuras 1.71 y 1.72 se muestran ejemplos de estas aplicaciones médicas.

## 1.9 INTERFAZ GRÁFICA DE USUARIO

Hoy día es muy común que las aplicaciones software se proporcionen con interfaces gráficas de usuario (GUI) (Grafic User Interface). Un componente principal en una interfaz gráfica es un gestor de ventanas que permite al usuario visualizar múltiples áreas rectangulares de la pantalla, llamadas ventanas de visualización. Cada área de visualización en la pantalla contiene un proceso diferente, mostrando información gráfica o de otro tipo pudiendo ser los métodos para activar una de estas ventanas variados. Si usamos un dispositivo apuntador interactivo, como un ratón, podemos en algunos sistemas, activar una ventana posicionando el cursor de la pantalla dentro del área mostrada por ésta presionando el botón izquierdo del ratón. Con otros sistemas tendremos que hacer clic con el ratón en la barra de título en la parte superior de la ventana.

Las interfaces también presentan menús e iconos para la selección de una ventana, una opción de proceso o el valor de un parámetro. Un ícono es un símbolo gráfico frecuentemente diseñado para sugerir la opción



**FIGURA 1.72.** Imagen de un par estereoscópico mostrando los huesos de una mano humana, renderizado por Inmo Yoon, D. E. Thompson y W. N. Waggoner, Jr., LSU, de un conjunto de datos obtenidos a partir de tomografías CT por Rehabilitation Research, GWLNHDC. Estas imágenes muestran un posible camino para la reconstrucción quirúrgica de un tendón. (Cortesía de IMRLAB, Mechanical Engineering, Universidad del Estado de Louisiana.)



**FIGURA 1.73.** Interfaz gráfica de usuario, mostrando múltiples ventanas, menús e iconos. (*Cortesía de Image-In Corporation.*)

que representa. La ventaja de los iconos es que necesitan menos espacio en la pantalla que la correspondiente descripción textual y puede ser entendido de una manera más rápida si se ha diseñado adecuadamente. Una ventana de visualización se puede convertir en o a partir de la representación de un ícono, los menús pueden contener listas tanto de descripciones textuales como de íconos.

La Figura 1.73 ilustra la típica interfaz gráfica de usuario, con múltiples ventanas, menús e iconos. En este ejemplo los menús permiten la selección de opciones de procesamiento, valores de color y parámetros gráficos. Los íconos representan opciones para pintar, dibujar, acercar, escribir cadenas de texto y otras operaciones relacionadas con la construcción de una pintura.

## 1.10 RESUMEN

---

Hemos hecho una prospección sobre muchas de las áreas en las que se aplican los gráficos por computadora, incluyendo la visualización de datos, CAD, realidad virtual, visualización científica, educación, arte, entretenimiento, procesamiento de imágenes e interfaces gráficas de usuario. Sin embargo, hay muchos otros campos que no hemos mencionado y con los que podríamos llenar este libro con ejemplos de aplicaciones. En los capítulos siguientes exploraremos los equipos y los métodos utilizados en las aplicaciones de este capítulo, así como otras muchas aplicaciones.

## REFERENCIAS

---

Aplicaciones de métodos gráficos en varias áreas, incluyendo arte, ciencia, matemáticas y tecnología son tratados en Bouquet (1978), Yessios (1979), Garner y Nelson (1983), Grotch (1983) ), Tufte (1983 y 1990), Wolfram (1984), Huitric y Nahas (1985), Glassner (1989), y Hearn and Baker (1991). Los métodos gráficos para la visualización de la música se dan en Mitroo, Herman, y Badler (1979). Disertaciones sobre diseño y fabricación asistidas por computadora (CAD/CAM) en varias industrias se presentan en Pao (1984). Las técnicas gráficas para simuladores de vuelo se presentan en Schachter (1983). Fu y Rosenfeld (1984) expone sobre la simulación de visión, y Weinberg (1978) da cuenta de simulación del transbordador espacial. Los iconos gráficos y los conceptos simbólicos se presentan en Lodding (1983) y en Loomis, *et al.* (1983). Para obtener información adicional sobre aplicaciones médicas véase Hawrylyshyn, Tasker y Organ (1977); Preston, Fagan, Huang y Pryor (1984); y Rhodes, *et al.* (1983).

# Introducción a los sistemas gráficos



Un sistema de representación de gráficos por computadora dotado de una pantalla panorámica y curvada, y su panel de control. (Cortesía de *Silicon Graphics, Inc* y *Tridimension Systems*. © 2003 SGI. Todos los derechos reservados.)

- |            |  |             |                       |
|------------|--|-------------|-----------------------|
| <b>2.1</b> | Dispositivos de visualización de vídeo                     | <b>2.6</b>  | Redes gráficas        |
| <b>2.2</b> | Sistemas de barrido de líneas                              | <b>2.7</b>  | Gráficos en Internet  |
| <b>2.3</b> | Estaciones de trabajo gráficas y sistemas de visualización | <b>2.8</b>  | Software gráfico      |
| <b>2.4</b> | Dispositivos de entrada                                    | <b>2.9</b>  | Introducción a OpenGL |
| <b>2.5</b> | Dispositivos de copia impresa                              | <b>2.10</b> | Resumen               |

La potencia y la utilidad de los gráficos por computador están ampliamente reconocidas, como lo demuestra la amplia gama de hardware gráfico y sistemas software disponibles en la actualidad para aplicaciones en casi todos los campos. Las capacidades gráficas tanto para aplicaciones bidimensionales como tridimensionales son comunes en computadores de propósito general como calculadoras de mano. Con los computadores personales, podemos utilizar una gran variedad de dispositivos de entrada interactivos y paquetes de software gráficos. Para aplicaciones que requieren una calidad superior, podemos escoger entre varios sistemas y tecnologías sofisticadas con hardware gráfico para propósitos especiales. En este capítulo, analizamos las características básicas de los componentes hardware y de los paquetes de software para gráficos.

## 2.1 DISPOSITIVOS DE VISUALIZACIÓN DE VÍDEO

---

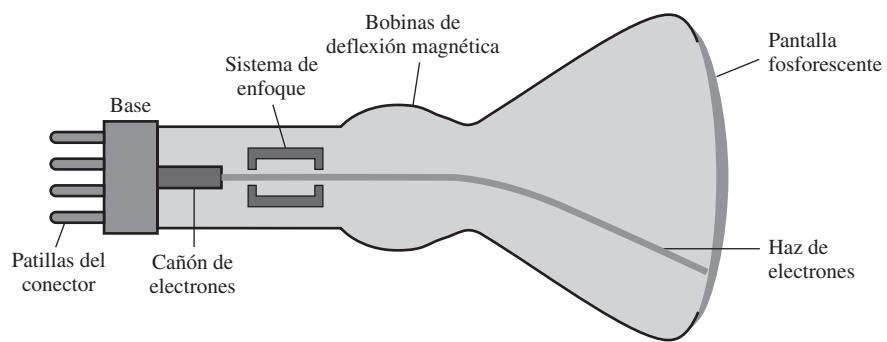
Por lo general, el dispositivo principal de salida en un sistema gráfico es un monitor de vídeo (Figura 2.1). El funcionamiento de la mayor parte de los monitores de vídeo se basa en el diseño estándar de **tubo de rayos catódicos** o TRC (**CRT, Cathode Ray Tube**), pero existen otras diversas tecnologías, por lo que con el tiempo pueden llegar a predominar los monitores de estado sólido.

### Tubos de refresco de rayos catódicos

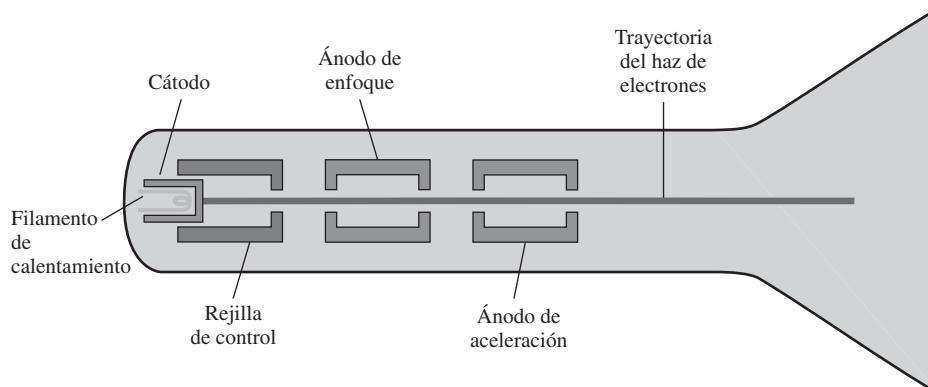
La Figura 2.2 ilustra el funcionamiento básico de un TRC. Un haz de electrones (rayos catódicos), emitido por un cañón de electrones, pasa a través de sistemas de enfoque y deflexión que dirigen el haz hacia posiciones específicas de la pantalla revestida de fósforo. Entonces el fósforo emite un pequeño punto de luz en cada



**FIGURA 2.1.** Una estación de trabajo para gráficos por computadora. (Cortesía de *Silicon Graphics, Inc., Why Not Films, and 525 Post Production. © 2003 SGI. Todos los derechos reservados.*)



**FIGURA 2.2.** Diseño básico de un TRC de deflexión magnética.



**FIGURA 2.3.** Funcionamiento de un cañón de electrones con ánodo de aceleración.

posición alcanzada por el haz de electrones. Puesto que la luz emitida por el fósforo se desvanece muy rápidamente, se requiere algún método para mantener la imagen en la pantalla. Una forma de hacer esto consiste en almacenar la información de la imagen en forma de distribución de carga dentro del TRC. Esta distribución de carga se puede utilizar para mantener el fósforo activado. Sin embargo, el método más utilizado en la actualidad para mantener el resplandor del fósforo es volver a dibujar la imagen redirigiendo rápidamente el haz de electrones de nuevo sobre los mismos puntos de la pantalla. Este tipo de pantalla se llama **TRC de refresco**. La frecuencia a la cual una imagen es redibujada en la pantalla se llama **velocidad de refresco**.

Los componentes principales de un cañón de electrones en un TRC son el cátodo de metal caliente y una rejilla de control (Figura 2.3). El calor se suministra al cátodo dirigiendo una corriente a través de una bobina de cable, llamada el filamento, dentro de la estructura cilíndrica del cátodo. Esto causa el «desprendimiento» de los electrones de la superficie del cátodo caliente. En el vacío, dentro de la cubierta del TRC, los electrones libres y cargados negativamente son acelerados hacia el recubrimiento de fósforo mediante una tensión altamente positiva. La tensión de aceleración se puede generar con un revestimiento de metal cargado positivamente dentro de la cubierta del TRC cerca de la pantalla de fósforo, o se puede utilizar un ánodo de aceleración, como el de la Figura 2.3, para suministrar la tensión positiva. A veces el cañón de electrones se diseña para que el ánodo de aceleración y el sistema de enfoque se encuentren dentro en la misma unidad.

La intensidad del haz de electrones se controla mediante la tensión de la rejilla de control, la cual es un cilindro de metal encajado sobre el cátodo. Una tensión muy negativa aplicada a la rejilla de control interrumpe el haz, al repeler e impedir que los electrones pasen a través de un pequeño agujero dispuesto al final de la estructura de la rejilla de control. Una tensión negativa menor en la rejilla de control simplemente reduce el número de electrones que pasan a través de ella. Ya que la cantidad de luz emitida por el revestimiento de

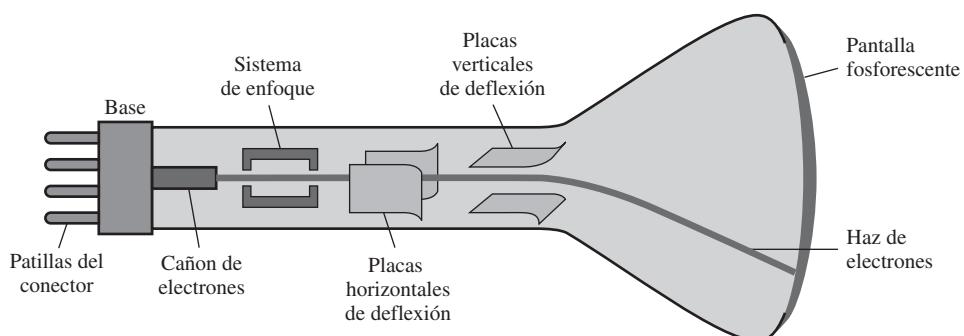
fósforo depende del número de electrones que chocan con la pantalla, el brillo de un punto de la pantalla se ajusta variando la tensión en la rejilla de control. Este brillo, o nivel de intensidad, se especifica para las posiciones individuales de la pantalla mediante instrucciones del software gráfico, como se estudia en el Capítulo 3.

El sistema de enfoque en un TRC obliga al haz de electrones a converger a una pequeña superficie cuando incide sobre el fósforo. De otro modo, los electrones se repelerían entre sí y el haz se expandiría a medida que se aproximase a la pantalla. El enfoque se consigue ya sea con campos eléctricos o campos magnéticos. Mediante enfoque electrostático, el haz de electrones se pasa a través de un cilindro metálico cargado positivamente para que los electrones se encuentren en una posición de equilibrio a lo largo del eje del cilindro. Esta disposición forma una lente electrostática, como se muestra en la Figura 2.3, y el haz de electrones se enfoca en el centro de la pantalla del mismo modo que una lente óptica enfoca un haz de luz a una distancia focal concreta. Se pueden conseguir efectos similares de enfoque de la lente con un campo magnético creado por una bobina montada alrededor del exterior de la cubierta del TRC. La lente magnética de enfoque habitualmente produce el punto en la pantalla de menor tamaño.

En sistemas de alta precisión se utiliza hardware adicional de enfoque para mantener el haz enfocado en todas las posiciones de la pantalla. La distancia que el haz de electrones debe viajar hasta los distintos puntos de la pantalla varía a causa de que el radio de curvatura en la mayoría de los TRC es mayor que la distancia del sistema de enfoque al centro de la pantalla. Por tanto, el haz de electrones se enfocará de manera adecuada sólo en el centro de la pantalla. Conforme el haz se mueve hacia los bordes externos de la pantalla, las imágenes mostradas se vuelven borrosas. Para compensar este efecto, el sistema puede ajustar el enfoque de acuerdo con la posición del haz en la pantalla.

Del mismo modo que ocurre con el enfoque, se puede controlar la deflexión del haz de electrones ya sea con campos eléctricos o con campos magnéticos. Los tubos de rayos catódicos se construyen habitualmente con bobinas de deflexión magnética, montadas sobre el exterior de la cubierta del TRC, como se ilustra en la Figura 2.2. Se emplean dos pares de bobinas para este propósito. Un par se monta en la parte superior y en la inferior del cuello del TRC, y el otro par se monta en los lados opuestos del cuello. El campo magnético producido por cada par de bobinas produce una fuerza transversal de deflexión que es perpendicular tanto a la dirección del campo magnético como a la dirección del trayecto del haz de electrones. La deflexión horizontal se logra con un par de bobinas, y la deflexión vertical con el otro par. Los niveles de deflexión apropiados se alcanzan al ajustar la corriente que atraviesa las bobinas. Cuando se emplea deflexión electrostática, se montan dos pares de placas paralelas dentro de la carcasa del TRC. Un par de placas se monta horizontalmente para controlar la deflexión vertical, y el otro par se monta verticalmente para controlar la deflexión horizontal (Figura 2.4).

Se producen puntos de luz en la pantalla al transferir la energía del haz del TRC al fósforo. Cuando los electrones del haz colisionan con el recubrimiento de fósforo, se detienen y su energía cinética es absorbida por el fósforo. Parte de la energía del haz se transforma por fricción en energía calorífica, y el resto provoca que los electrones de los átomos de fósforo se muevan hacia niveles cuánticos de energía superiores. Poco



**FIGURA 2.4.** Deflexión electrostática de un haz de electrones en un TRC.



**FIGURA 2.5.** Distribución de intensidad de un punto de fósforo iluminado en una pantalla de TRC.



**FIGURA 2.6.** Dos puntos de fósforo iluminado se pueden distinguir cuando su separación es mayor que el diámetro en que se reduce la intensidad un 60 por ciento del máximo.

tiempo después, los electrones de fósforo «excitados» comienzan a regresar a su estado estable y despiden su exceso de energía en forma de pequeños «cuantos» de energía luminosa llamados fotones. Lo que vemos en la pantalla es el efecto combinado de todas las emisiones de luz de los electrones: un punto resplandeciente que rápidamente se desvanece después de que todos los electrones excitados de fósforo hayan vuelto a su nivel estable de energía. La frecuencia (o color) de la luz emitida por el fósforo es proporcional a la diferencia de energía entre el estado cuántico excitado y el estado estable.

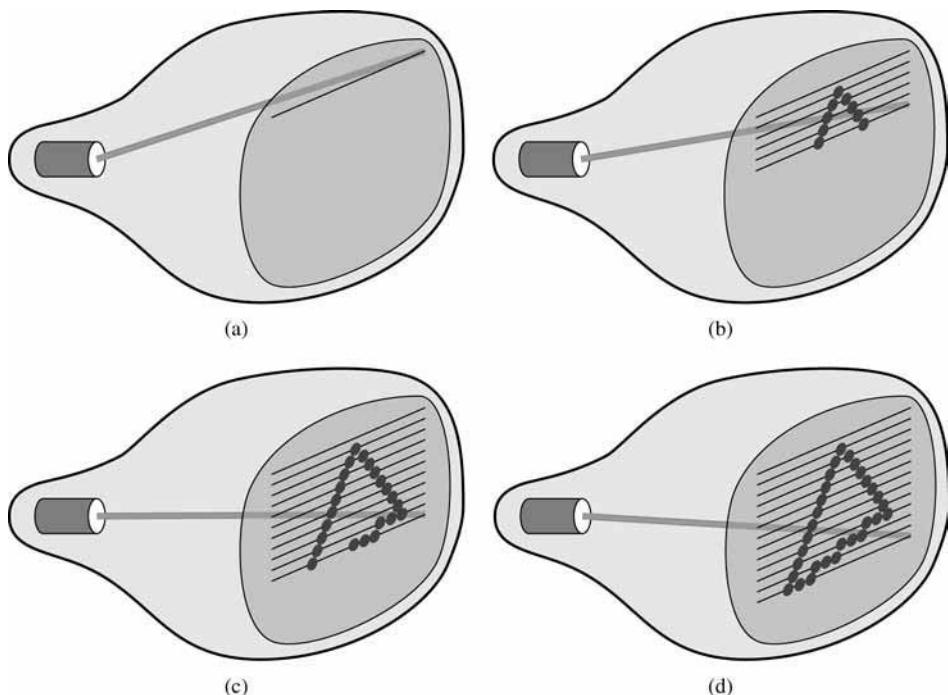
Hay disponibles diferentes clases de fósforo para su uso en tubos de rayos catódicos. Además del color, la diferencia mayor entre los fósforos es su **persistencia**: ¿cuánto tiempo permanecerán emitiendo luz (es decir, ¿cuánto tiempo pasará antes de que todos los electrones excitados hayan regresado a su estado estable?) después de que el haz del TRC se ha retirado? La persistencia se define como el tiempo que transcurre para que la luz emitida desde la pantalla disminuya a una décima parte de su intensidad original. Los fósforos de menor persistencia requieren velocidades de refresco mayores para mantener una imagen en la pantalla sin parpadeo. Un fósforo con baja persistencia puede ser útil en animación, mientras que los fósforos de alta persistencia son más adecuados para mostrar imágenes estáticas de alta complejidad. Aunque algunos fósforos poseen valores de persistencia mayores de 1 segundo, los monitores gráficos de propósito general se suelen construir con una persistencia del orden de 10 a 60 microsegundos.

La Figura 2.5 muestra la distribución de intensidad de un punto en la pantalla. La intensidad es mayor en el centro del punto y decrece según una distribución gaussiana hacia los bordes del punto. Esta distribución se corresponde con la distribución transversal de la densidad de electrones del haz del TRC.

El máximo número de puntos que se pueden mostrar sin que se solapen en un TRC se conoce como **resolución**. Una definición más precisa de resolución es el número de puntos por centímetro que se pueden dibujar horizontal y verticalmente, aunque a menudo sólo se expresa como el número total de puntos en cada dirección. La intensidad del punto posee una distribución gaussiana (Figura 2.5), de manera que dos puntos adyacentes parecerán distintos siempre y cuando su separación sea mayor que el diámetro en que cada punto tiene una intensidad aproximada del 60% de la del centro del punto. Esta superposición se presenta en la Figura 2.6. El tamaño del punto también depende de la intensidad. Cuantos más electrones por segundo se aceleren hacia el fósforo, mayor será el diámetro del haz del TRC y el punto iluminado será de mayor tamaño. Además, la energía de excitación aumentada tiende a extenderse hacia los átomos de fósforo vecinos que no se encuentran en el camino del haz, lo cual aumenta más aún el diámetro del punto. Por tanto, la resolución de un TRC depende del tipo de fósforo, la intensidad a mostrar y los sistemas de enfoque y deflexión. La resolución habitual en los sistemas de alta calidad es 1280 por 1024, aunque hay disponibles resoluciones mayores en muchos sistemas. Los sistemas de alta resolución se suelen denominar *sistemas de alta definición*. El tamaño físico de un monitor gráfico, por otra parte, se expresa como la longitud de la diagonal de la pantalla y varía desde aproximadamente 12 hasta 27 pulgadas o más. Un monitor TRC se puede conectar a múltiples computadoras, por lo que el número de puntos de pantalla que se pueden dibujar realmente también depende de las capacidades del sistema al cual está conectado.

## Pantallas por barrido de líneas

La clase más común de monitor gráfico que utiliza un TRC es la **pantalla por barrido de líneas** y está basada en la tecnología de la televisión. En un sistema de barrido de líneas, el haz de electrones recorre la panta-



**FIGURA 2.7.** Un sistema de barrido de líneas muestra un objeto como un conjunto de puntos discretos a lo largo de cada línea de barrido.

lla, una fila cada vez y de arriba hacia abajo. Cada fila se denomina **línea de barrido**. Ya que el haz de electrones se mueve a lo largo de una línea de barrido, la intensidad del haz se activa y se desactiva (o se establece a un nivel intermedio) para crear un patrón de puntos iluminados. La definición de la imagen se almacena en un área de la memoria llamada **búfer de refresco** o **búfer de imagen**, donde el término **imagen** hace referencia al área de la pantalla en su totalidad. Este área de memoria contiene el conjunto de niveles de color de los puntos de la pantalla. Estos niveles de color almacenados se obtienen del búfer de refresco y se utilizan para controlar la intensidad del haz de electrones a medida que se mueve de un punto a otro por la pantalla. De este modo, la imagen se «pinta» en la pantalla de línea a línea de barrido, como se muestra en la Figura 2.7. Cada punto de pantalla que se puede iluminar por el haz de electrones se denomina **píxel** o **pel** (formas abreviadas de **picture element**; elemento de imagen). Dado que el búfer de refresco se emplea para almacenar el conjunto de niveles de color de la pantalla, a veces también se denomina **búfer de imagen**. También, otras clases de información del píxel, además del color, se almacenan en posiciones del búfer, por lo que todas estas zonas diferentes del búfer a veces se denominan de forma colectiva «búfer de cuadro». La capacidad de un sistema de barrido de líneas para almacenar la información de color para cada punto de pantalla hace que éste sea muy adecuado para la representación realista de escenas que contienen sutiles sombreados y patrones de color. Los sistemas de televisión y las impresoras son ejemplos de otros sistemas que utilizan métodos de barrido de líneas.

Los sistemas de barrido se caracterizan comúnmente por su resolución, la cual es el número de píxeles que se pueden dibujar. Otra propiedad de los monitores de vídeo es su **relación de aspecto**, la cual se define a menudo como el número de columnas de píxeles dividido entre el número de líneas de barrido que se pueden mostrar en el sistema (a veces el término relación de aspecto se utiliza para hacer referencia al número de líneas de barrido dividido entre el número de columnas de píxeles). La relación de aspecto también se puede describir como el número de puntos horizontales frente al número de puntos verticales (o viceversa) necesario para generar líneas de igual longitud en ambas direcciones de la pantalla. Por tanto, una relación de aspecto

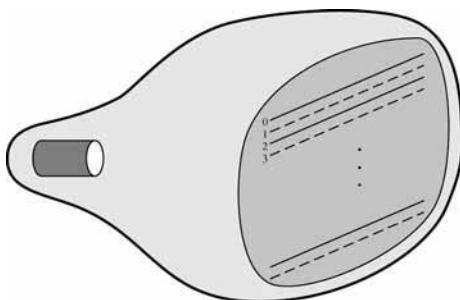
to de 4/3, por ejemplo, significa que una línea horizontal pintada con cuatro puntos posee la misma longitud que una línea vertical dibujada con tres puntos; la longitud de la línea se mide con alguna unidad física tal como el centímetro. Similarmente, la relación de aspecto de cualquier rectángulo (incluyendo el área total de la pantalla) se puede definir para que sea la anchura del rectángulo dividida entre su altura.

La gama de colores o de niveles de gris que se puede mostrar en un sistema de barrido depende tanto del tipo de fósforo utilizado en el TRC como del número de bits por píxel disponibles en el búfer de imagen. En sistemas de blanco y negro puros, cada punto de la pantalla o está activado o desactivado, por lo que sólo se necesita un bit por píxel para ajustar la intensidad de los puntos de pantalla. Un bit de valor 1, por ejemplo, indica que el haz de electrones se debe activar en aquella posición, y un valor de 0 desactiva el haz. Bits adicionales permiten que la intensidad del haz de electrones pueda variar dentro de una gama de valores entre «activado» y «desactivado». Los sistemas de alta calidad incluyen hasta 24 bits por píxel, por lo que pueden requerir varios megabytes de almacenamiento para el búfer de imagen, dependiendo de la resolución del sistema. Por ejemplo, un sistema con 24 bits por píxel y una resolución de pantalla de 1024 por 1024 requieren 3 megabytes de almacenamiento para el búfer de refresco. El número de bits por píxel de un búfer de imagen se denomina a veces **profundidad** del área de búfer o el número de **planos de bit**. También, el búfer de un bit por píxel se denomina comúnmente **bitmap** y un búfer de imagen con múltiples bits por pixel es un  **pixmap**. Pero los términos bitmap y pixmap se emplean también para describir otras matrices rectangulares, en las cuales un bitmap es cualquier patrón de valores binarios y un pixmap es un patrón multicolor.

Cada vez que se refresca la pantalla, tendemos a ver cada cuadro o imagen como una continuación suave de los patrones del cuadro anterior, siempre y cuando la velocidad de refresco no sea demasiado baja. Por debajo de, aproximadamente, 24 cuadros por segundo, podemos habitualmente percibir una separación entre las sucesivas imágenes de pantalla, y cada imagen parece parpadear. Las antiguas películas de cine mudo, por ejemplo, muestran este efecto, ya que gran parte de ellas fueron fotografiadas a una velocidad de 16 cuadros por segundo. Cuando se desarrollaron en los años veinte los sistemas sonoros, la velocidad de las imágenes en movimiento se incrementó a 24 cuadros por segundo, lo cual eliminó el parpadeo y los movimientos a tiros de los actores. Los primeros sistemas de barrido de líneas para computadora se diseñaron con una velocidad de refresco de, aproximadamente, 30 cuadros por segundo. Esto produce resultados suficientemente buenos, pero la calidad de la imagen se ha mejorado, en cierta manera, con las velocidades más elevadas de refresco de los monitores de vídeo, porque la tecnología de representación en un monitor es básicamente diferente de la de una película. Un proyector de películas puede mantener la representación continua de un cuadro de la película hasta que se muestra el siguiente cuadro. Pero en un monitor de vídeo, un punto de fósforo comienza a desvanecerse tan pronto como se ilumina. Por tanto, las pantallas de barrido de líneas habituales refrescan la pantalla a una velocidad de 60 a 80 cuadros por segundo, aunque algunos sistemas actuales poseen velocidades de refresco de hasta 120 cuadros por segundo. Y algunos sistemas gráficos han sido diseñados con una velocidad de refresco variable. Por ejemplo, una velocidad de refresco más elevada se podría utilizar en una aplicación estereoscópica para que las dos vistas (una para cada posición del ojo) de una escena se puedan mostrar alternativamente sin parpadeo. Pero, habitualmente, se utilizan otros métodos tales como múltiples búferes de imagen para tales aplicaciones.

A veces, la velocidades de refresco se indican en unidades de ciclos por segundo, o hercios (Hz), donde un ciclo se corresponde con un cuadro. Empleando estas unidades, indicaríamos una velocidad de refresco de 60 cuadros por segundo como simplemente 60 Hz. Al final de cada línea de barrido, el haz de electrones vuelve al lado izquierdo de la pantalla para comenzar a mostrar la siguiente línea de barrido. La vuelta a la parte izquierda de la pantalla, después de refrescar cada línea de barrido, se denomina **retrazado horizontal** del haz de electrones. Y al final de cada cuadro (mostrado en  $\frac{1}{80}$  a  $\frac{1}{60}$  segundos), el haz de electrones vuelve a la esquina superior izquierda de la pantalla (**retrazado vertical**) para comenzar el siguiente cuadro.

En algunos sistemas de barrido de líneas y televisiones, cada cuadro se muestra en dos pasos mediante un procedimiento de refresco *entrelazado*. En la primera pasada, el haz barre una de cada dos líneas de arriba hacia abajo. Después del retrazado vertical, el haz barre el resto de las líneas (Figura 2.8). El entrelazado de las líneas de rastreo de este modo nos permite mostrar la pantalla completa en la mitad de tiempo que

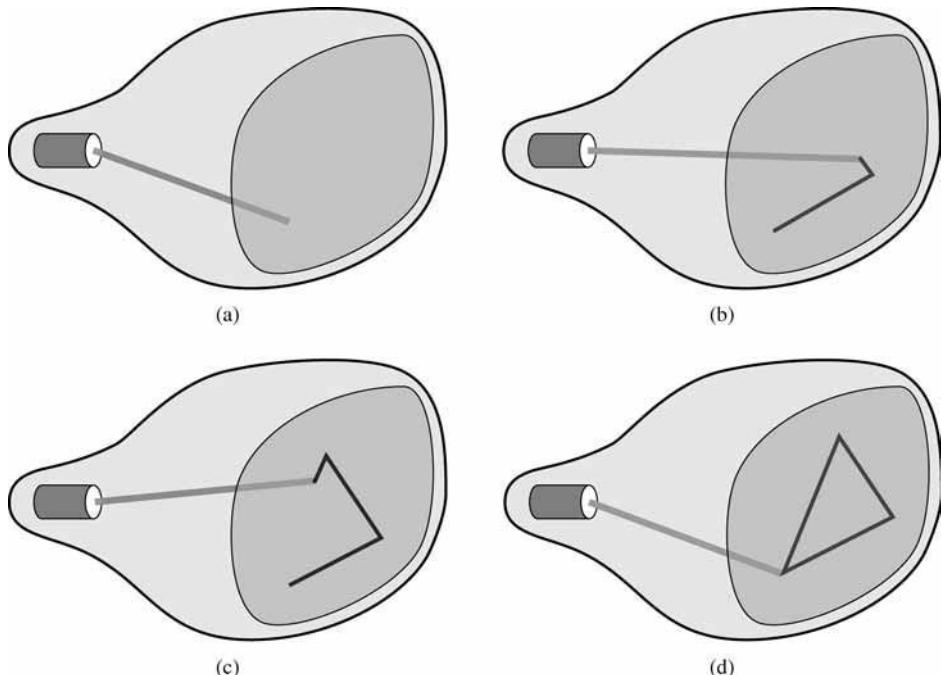


**FIGURA 2.8.** Entrelazado de las líneas de barrido en un monitor de barrido. En primer lugar, se muestran los puntos de las líneas de barrido con numeración par (línea continua); y después se muestran todos los puntos de las líneas con numeración impar (línea discontinua).

transcurriría si barriésemos todas las líneas una vez, de arriba hacia abajo. Esta técnica se utiliza fundamentalmente en el caso de velocidades de refresco bajas. Por ejemplo, en un monitor de 30 cuadros por segundo no entrelazado se observa algún parpadeo. Sin embargo, con entrelazado, cada una de las dos pasadas se puede realizar en  $\frac{1}{60}$  segundos, lo cual hace que la velocidad de refresco se aproxime a 60 cuadros por segundo. Esta es una técnica efectiva para evitar parpadeos, siempre que las líneas de barrido adyacentes contengan información de representación similar.

### Pantallas de barrido aleatorio

Cuando un monitor de TRC funciona como **pantalla de barrido aleatorio**, dispone de un haz de electrones dirigido exclusivamente a las partes de la pantalla donde se muestra una imagen. Las imágenes se generan a base de líneas; el haz de electrones traza las líneas componentes una tras otra. Por este motivo, los monitores de barrido aleatorio se denominan también **pantallas vectoriales** (o **pantallas de escritura de impacto** o **pantallas caligráficas**). Las líneas componentes de una imagen se pueden dibujar y refrescar en los monitores



**FIGURA 2.9.** Un sistema de barrido aleatorio dibuja las líneas componentes de un objeto específico en cualquier orden que se especifique.

de barrido aleatorio en cualquier orden (Figura 2.9). Una plumilla de un trazador funciona de un modo similar y constituye un ejemplo de dispositivo de impresión de barrido aleatorio.

La velocidad de refresco en un sistema de barrido aleatorio depende del número de líneas que deba mostrar. La definición de la imagen se almacena como un conjunto de órdenes de dibujo de líneas en una zona de memoria denominada **lista de visualización, archivo de refresco de visualización, archivo vectorial, o programa de visualización**. Para mostrar una imagen concreta, el sistema recorre el conjunto de comandos del archivo de visualización, dibujando una línea componente cada vez. Después de que todos los comandos de dibujo de líneas se han procesado, el sistema vuelve al comando de la primera línea de la lista. Las pantallas de barrido aleatorio se diseñan para dibujar todas las líneas componentes de una imagen de 30 a 60 veces por segundo, con hasta 100.000 líneas «cortas» en la lista de visualización. Cuando se debe mostrar un pequeño conjunto de líneas, cada ciclo de refresco se retrasa para evitar frecuencias de refresco muy elevadas, las cuales podrían quemar el fósforo.

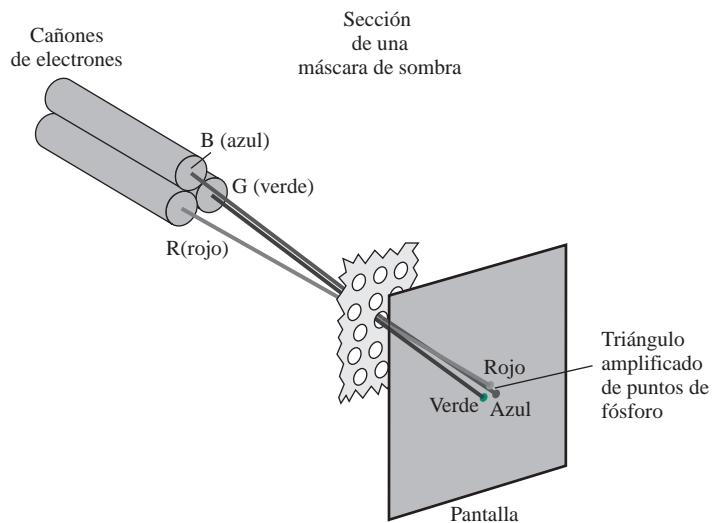
Los sistemas de barrido aleatorio se diseñaron para aplicaciones de dibujo de líneas, tales como diseños de arquitectura e ingeniería, y no son capaces de mostrar escenas con matices realistas. Ya que la definición de una imagen se almacena como un conjunto de instrucciones de dibujo de líneas, en lugar de como un conjunto de niveles de intensidad para todos los puntos de pantalla, las pantallas vectoriales disponen generalmente de resoluciones más altas que los sistemas de barrido. Del mismo modo, las pantallas vectoriales producen dibujos de líneas más suaves, ya que el haz del TRC sigue la trayectoria de la línea. En cambio, un sistema de barrido produce líneas dentadas que se muestran como conjuntos de puntos discretos. Sin embargo, la mayor flexibilidad y las capacidades mejoradas de dibujo de líneas de los sistemas de barrido han provocado el abandono de la tecnología vectorial.

## Monitores TRC de color

Un monitor TRC muestra imágenes en color empleando una combinación de fósforos que emiten luz de diferentes colores. Las luces emitidas por los diferentes fósforos se funden para formar un único color percibido, el cual depende del conjunto particular de fósforos que se hayan excitado.

Un método para representar imágenes en color en la pantalla consiste en cubrir la pantalla con capas de fósforos de diferente color. El color emitido depende de la penetración del haz de electrones en las capas de fósforo. Esta técnica, llamada **método de penetración del haz**, se suele utilizar sólo con dos capas de fósforo: roja y verde. Un haz de electrones lentos excita sólo la capa roja exterior, sin embargo, un haz de electrones rápidos penetra a través de la capa roja y excita la capa interior verde. Con velocidades intermedias del haz se emiten combinaciones de luz roja y verde para mostrar dos colores adicionales como naranja y amarillo. La velocidad de los electrones y, por tanto, el color de la pantalla en cualquier punto se ajusta mediante la tensión de aceleración del haz. La penetración del haz fue un método barato para producir color, pero sólo permite un número limitado de colores, y la calidad de la imagen no es tan buena como con otros métodos.

Los métodos de la **máscara de sombra** se utilizan habitualmente en sistemas de barrido con rastreo (televisores en color incluidos), ya que producen una más amplia gama de colores que el método de penetración del haz. Esta técnica se basa en el modo con el que parece que percibimos los colores como combinaciones de las componentes roja, verde y azul, y se llama **modelo de color RGB**. Por tanto, un TRC de máscara de sombra emplea tres puntos de color de fósforo en cada píxel. Un punto de fósforo emite luz roja, otro emite luz verde y el tercero emite luz azul. Este tipo de TRC posee tres cañones de electrones, uno para cada punto de color y una rejilla de máscara justo detrás de la pantalla recubierta de fósforo. La luz emitida desde los tres fósforos se convierte en un pequeño punto de color en cada píxel, ya que nuestros ojos tienden a fundir la luz emitida desde los tres puntos en un color compuesto. La Figura 2.10 muestra el método *delta-delta* de la máscara de sombra, que se utiliza habitualmente en los sistemas de TRC de color. Los tres haces de electrones se deflectan y enfocan como un grupo en la máscara de sombra, la cual contiene una serie de agujeros alineados con los patrones de puntos de fósforo. Cuando los tres haces pasan a través de un agujero de la máscara de sombra, activan un triángulo de puntos, el cual aparece como un pequeño punto de color en la pantalla. Los puntos de fósforo en los triángulos se disponen de modo que cada haz de electrones pueda



**FIGURA 2.10.** Funcionamiento de un TRC delta-delta de máscara de sombra. Tres cañones de electrones, alineados con los patrones triangulares de puntos de color de la pantalla, se dirigen hacia cada triángulo de puntos mediante una máscara de sombra.

activar sólo su punto de color correspondiente cuando pasa a través de la máscara de sombra. Otra configuración para los tres cañones de electrones es una disposición en línea en la cual los tres cañones de electrones, y los correspondientes puntos de color rojo-verde-azul, se encuentran alineados a lo largo de una línea de barrido en lugar de en un patrón triangular. Esta disposición en línea de los cañones de electrones es más fácil de mantener alineada y, habitualmente, se emplea en los TRC de color de alta resolución.

Mediante la variación de los niveles de intensidad de los tres haces de electrones podemos obtener combinaciones de color en la máscara de sombra del TRC. Si se apagan dos de los tres cañones, obtenemos sólo el color procedente del único fósforo activado (rojo, verde, o azul). Al activar los tres puntos con iguales intensidades de los haces, vemos el color blanco. El color amarillo se produce con iguales intensidades en los puntos verde y rojo solamente, el color magenta se produce con iguales intensidades en los puntos azul y rojo, y el color cian cuando los puntos azul y verde se activan del mismo modo. En un sistema de bajo coste, cada uno de los tres haces de electrones puede que únicamente permita estar activado o desactivado, de modo que sólo puede mostrar ocho colores. Sistemas más sofisticados pueden permitir que se establezcan niveles de intensidad intermedios en los haces de electrones, por lo que son capaces de mostrar varios millones de colores.

Los sistemas gráficos en color se pueden usar con varias clases de pantallas de TRC. Algunas computadoras de bajo coste para el hogar y videojuegos se han diseñado para utilizarlos con una televisión en color y un modulador de RF (radio-frecuencia). El propósito del modulador RF es simular la señal procedente de una emisora de televisión. Esto quiere decir que la información del color y la intensidad de la imagen se debe combinar y superponer a la señal portadora de la frecuencia de multidifusión que la televisión requiere como entrada. Entonces la circuitería del televisor toma esta señal procedente del modulador de RF, extrae la información de la imagen y la pinta en la pantalla. Como era de esperar, está manipulación adicional de la información de la imagen por parte del modulador de RF y de la circuitería de la televisión decrementa la calidad de las imágenes mostradas.

Los **monitores compuestos** son adaptaciones de los televisores que permiten evitar la circuitería de multidifusión. Estos dispositivos de pantalla aún requieren que la información de la imagen se combine, pero no se necesita señal portadora. Dado que la información de la imagen se combina en una señal compuesta y, a continuación el monitor la separa, la calidad de la imagen resultante no es todavía la mejor que se puede alcanzar.

Los TRC de color para sistemas gráficos se diseñan como los **monitores RGB**. Estos monitores emplean métodos de máscara de sombra y toman el nivel de intensidad de cada cañón de electrones (rojo, verde, y azul) directamente de la computadora sin ningún procesamiento intermedio. Los sistemas gráficos de barrido de alta calidad poseen 24 bits por píxel en el búfer de imagen, permitiendo 256 configuraciones de tensión para cada cañón de electrones y cerca de 17 millones de posibles colores para cada píxel. Un sistema en color RGB con 24 bits de almacenamiento por píxel se le denomina, generalmente, **sistema de color completo** o **sistema de color real**.

## Pantallas planas

Aunque la mayor parte de los monitores gráficos se construyen aún con los TRC, están surgiendo otras tecnologías que podrían pronto reemplazar a los monitores de TRC. El término **pantalla plana** se refiere a la clase de dispositivos de video que han reducido su volumen, peso y requisitos de potencia comparados con un TRC. Una característica significativa de las pantallas planas es que son más finas que los TRC y podemos colgarlas en la pared o llevarlas en la muñeca. Dado que es posible escribir en algunas pantallas planas, se encuentran también disponibles como bloc de notas de bolsillo. Algunos usos adicionales de las pantallas planas son los siguientes: pequeños televisores, pantallas de calculadoras, pantallas de videojuegos de bolsillo, pantallas de computadoras portátiles, las pantallas disponibles en los brazos de los asientos de los aviones para ver películas, paneles para anuncios en los ascensores y pantallas gráficas para aplicaciones que requieren monitores altamente portables.

Podemos clasificar las pantallas planas en dos categorías: **pantallas emisivas** y **pantallas no emisivas**. Las pantallas emisivas (o **emisores**) son dispositivos que transforman la energía eléctrica en luz. Como ejemplos de pantallas emisivas se pueden citar las pantallas de plasma, las pantallas electroluminiscentes de película fina y los diodos que emiten luz. También se han ideado TRC planos, en los que los haces de electrones se aceleran paralelos a la pantalla y se deflectan 90° sobre la pantalla. Pero no se ha demostrado que los TRC planos sean tan exitosos como otros dispositivos emisivos. Las pantallas no emisivas (o **no emisores**) emplean efectos ópticos para transformar la luz del sol o la luz de alguna otra fuente en patrones gráficos. El ejemplo más importante de una pantalla plana no emisiva es el dispositivo de cristal líquido.

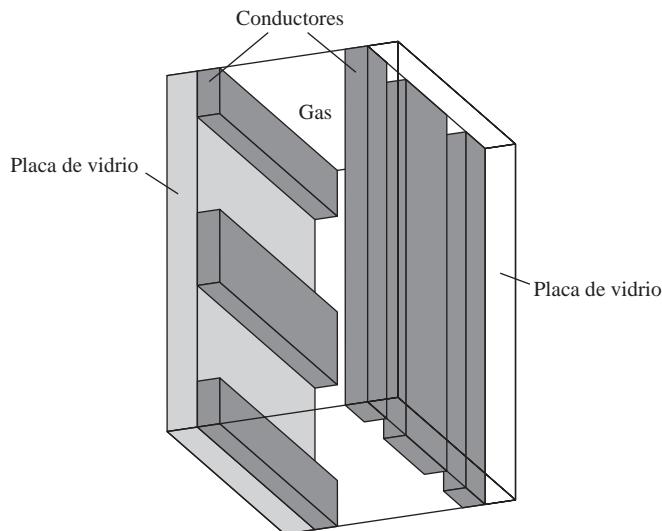
Las **pantallas de plasma**, también llamadas **pantallas de descarga de gas**, se construyen llenando el espacio entre dos placas de cristal con una mezcla de gases que habitualmente incluye el neón. Una serie de tiras de material conductor se colocan de forma vertical en un panel de cristal, y en el otro panel de cristal se sitúan de forma horizontal (Figura 2.11). Si se aplican tensiones de disparo a un par de intersección de conductores verticales y horizontales se provoca que el gas en la intersección de los dos conductores se descomponga en un plasma de electrones e iones que emiten luz. La definición de la imagen se almacena en un búfer de refresco y las tensiones de disparo se aplican para refrescar los píxeles (en las intersecciones de los conductores) 60 veces por segundo. Se emplean métodos de corriente alterna para proporcionar la aplicación más rápida de tensiones de disparo y, por tanto, generar pantallas más brillantes. La separación entre los píxeles se debe al campo eléctrico de los conductores. La Figura 2.12 muestra una pantalla de plasma de alta definición. Una desventaja de las pantallas de plasma fue que eran estrictamente dispositivos monocromos, pero en la actualidad existen pantallas de plasma con capacidades multicolor.

Las **pantallas electroluminiscentes de película fina** se construyen de forma similar a las pantallas de plasma. La diferencia consiste en que el espacio entre las placas de cristal se llena con fósforo, tal como sulfato de zinc dopado con manganeso, en lugar de con gas (Figura 2.13). Cuando se aplica una tensión suficientemente alta a un par de electrodos que se cruzan, el fósforo se transforma en conductor en el área de la intersección de los dos electrodos. La energía eléctrica es absorbida por los átomos de manganeso, los cuales liberan entonces la energía como un punto luminoso similar al efecto del plasma luminiscente en la pantalla de plasma. Las pantallas electroluminiscentes requieren más potencia que las pantallas de plasma, y en color son más difíciles de conseguir.

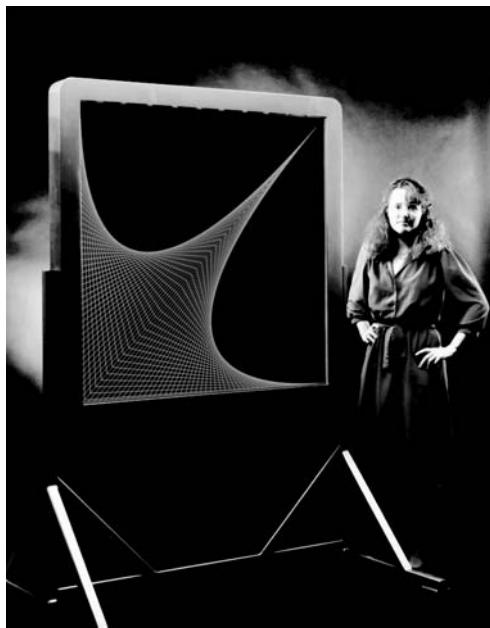
La tercera clase de dispositivos emisivos es el **diodo emisor de luz (LED)**. Los píxeles de la pantalla se crean mediante una matriz de diodos, y la resolución de la imagen se almacena en el búfer de refresco. Como

en el caso del refresco de la línea de barrido de un TRC, la información se lee del búfer de refresco y se transforma en niveles de tensión que se aplican a los diodos para producir patrones de luz en la pantalla.

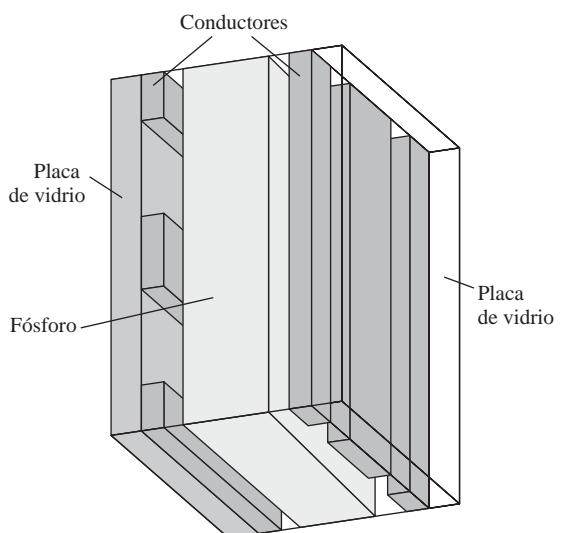
Las **pantallas de cristal líquido (LCD)** se utilizan habitualmente en sistemas pequeños, tales como computadoras portátiles y calculadoras (Figura 2.14). Estos sistemas no emisivos producen una imagen



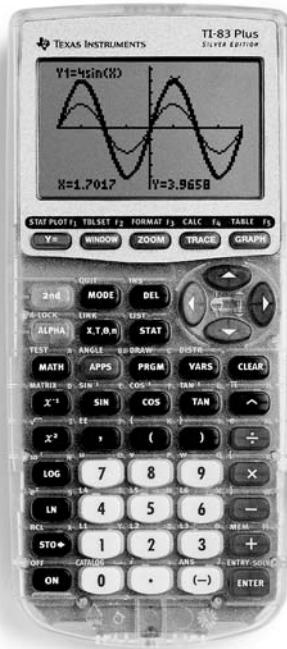
**FIGURA 2.11.** Diseño básico de un dispositivo de pantalla de plasma.



**FIGURA 2.12.** Una pantalla de plasma con una resolución de 2048 por 2048 y una diagonal de pantalla de 1,5 m. (Cortesía de Photonics Systems.)



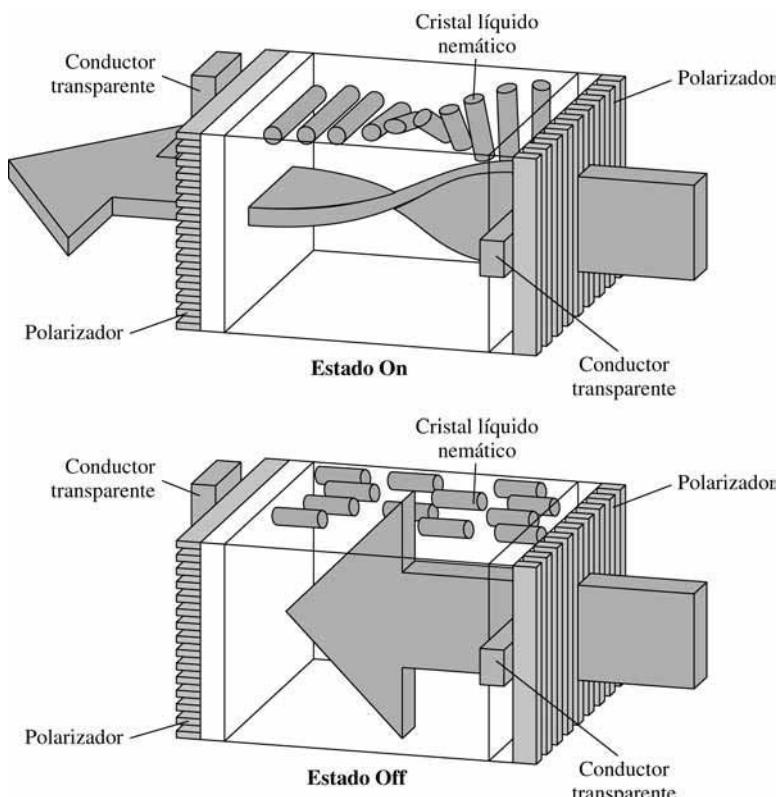
**FIGURA 2.13.** Diseño básico de un dispositivo de pantalla electroluminiscente de película fina.



**FIGURA 2.14.** Una calculadora de mano con una pantalla LCD. (Cortesía de Texas Instruments.)

mediante el paso de luz polarizada desde las cercanías o desde una fuente interna a través de un material de cristal líquido que se puede alinear para bloquear o trasmitir la luz.

El término *cristal líquido* hace referencia al hecho de que en estos compuestos las moléculas están dispuestas según una estructura cristalina, a pesar de que fluyen como en un líquido. Las pantallas planas utilizan habitualmente compuestos de cristal líquido nemáticos (con aspecto de hebras) y que tienden a mantener los largos ejes de moléculas con forma de cuerda alineados. Una pantalla plana se puede construir con un cristal líquido nemático, como se demuestra en la Figura 2.15. Dos placas de cristal, cada una de las cuales contiene un polarizador de luz que está alineado en ángulo recto con la otra placa, encierran el material de cristal líquido. En una placa de cristal se han dispuesto conductores transparentes en filas horizontales, y en la otra placa se han dispuesto dichos conductores en columnas verticales. Cada intersección de dos conductores determina un píxel. Normalmente, las moléculas se alinean como se muestra en el «estado encendido» de la Figura 2.15. La luz polarizada que pasa a través de material se retuerce para que pase a través del polarizador opuesto. La luz entonces se refleja de vuelta hacia el visor. Para apagar el píxel, aplicamos una tensión a los dos conductores que se intersectan, para alinear las moléculas para que la luz no se retuerza. Esta clase de dispositivos de pantalla plana se denominan LCD de **matriz pasiva**. La definición de la imagen se almacena en un búfer de refresco, y la pantalla se refresca a una velocidad de 60 cuadros por segundo, como en los dispositivos emisivos. La retroiluminación también se aplica habitualmente en los dispositivos electrónicos de estado sólido, para que el sistema no dependa completamente de las fuentes de luz externas. Los colores se pueden mostrar mediante el empleo de diferentes materiales o tintes y mediante la colocación de una triada



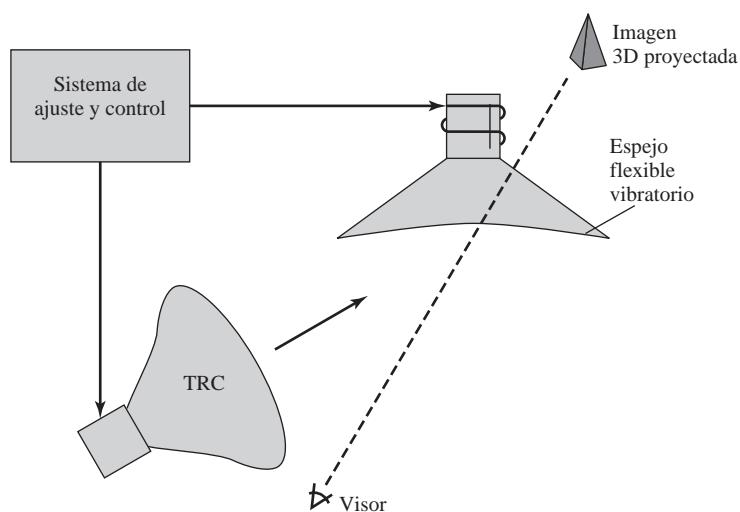
**FIGURA 2.15.** El efecto de luz retorcida se emplea en el diseño de la mayor parte de los dispositivos de pantalla de cristal líquido.

de píxeles de color en cada posición de la pantalla. Otro método para construir los LCD consiste en colocar un transistor en cada píxel, empleando tecnología de transistores de película fina. Los transistores se emplean para ajustar la tensión de los píxeles y para prevenir que la carga se fugue de manera gradual de las celdas de cristal líquido. Estos dispositivos se denominan pantallas de **matriz activa**.

## Dispositivos de visualización tridimensional

Se han ideado monitores gráficos para mostrar escenas tridimensionales que emplean una técnica que refleja una imagen de un monitor de TRC desde un espejo flexible y que vibra (Figura 2.16). Conforme el espejo varifocal vibra, cambia su distancia focal. Estas vibraciones están sincronizadas con la representación de un objeto en un monitor de TRC, para que cada punto del objeto sea reflejado desde el espejo hacia la posición espacial correspondiente a la distancia de ese punto, desde una localización de visionado específica. Esto permite caminar alrededor de un objeto o una escena y verlo desde diferentes lados.

La Figura 2.17 muestra el sistema SpaceGraph de Genisco, el cual emplea una espejo vibrante para proyectar objetos tridimensionales en un volumen de 25 cm por 25 cm por 25 cm. Este sistema también es capaz de mostrar «rebanadas» bidimensionales transversales de objetos seleccionados a diferentes profundidades. Tales sistemas se han utilizado en aplicaciones para analizar los datos procedentes de ultrasonografía y dispositivos de rastreo CAT, en aplicaciones geológicas para analizar datos topológicos y sísmicos, en aplicaciones de diseño en las que están involucrados objetos sólidos y en simulaciones tridimensionales de sistemas tales como moléculas y de terreno.



**FIGURA 2.16.** Funcionamiento de un sistema de pantalla tridimensional que emplea un espejo vibrante que cambia su distancia focal para ajustarse a las profundidades de los puntos de una escena.

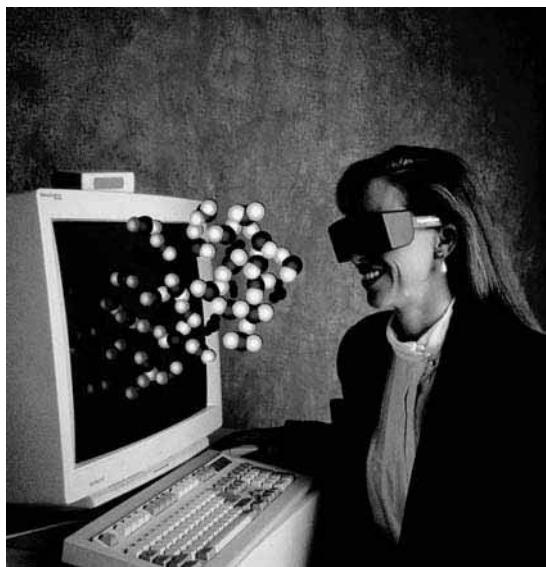
## Sistemas estereoscópicos y de realidad virtual

Mostrar vistas estereoscópicas de un objeto es otra técnica para representar un objeto tridimensional. Este método no produce imágenes realmente tridimensionales, pero proporciona un efecto tridimensional mediante la presentación de una vista diferente a cada ojo de un observador, para que parezca que las escenas poseen profundidad (Figura 2.18).

Para obtener una proyección estereoscópica, debemos obtener dos vistas de una escena generadas con las direcciones de visualización desde la posición de cada ojo (izquierda y derecha) hacia la escena. Podemos construir las dos vistas como escenas generadas por computadora con diferentes puntos de vista, o podemos utilizar un par de cámaras estéreo para fotografiar un objeto o una escena. Cuando miramos simultáneamente a la vista izquierda con el ojo izquierdo y a la vista derecha con el ojo derecho, las dos imágenes se funden en una única imagen y percibimos una escena con profundidad. La Figura 2.19 muestra dos vistas de una esce-



**FIGURA 2.17.** El SpaceGraph, sistema gráfico interactivo que muestra objetos en tres dimensiones empleando un espejo flexible y vibrante. (Cortesía de Genisco Computers Corporation.)



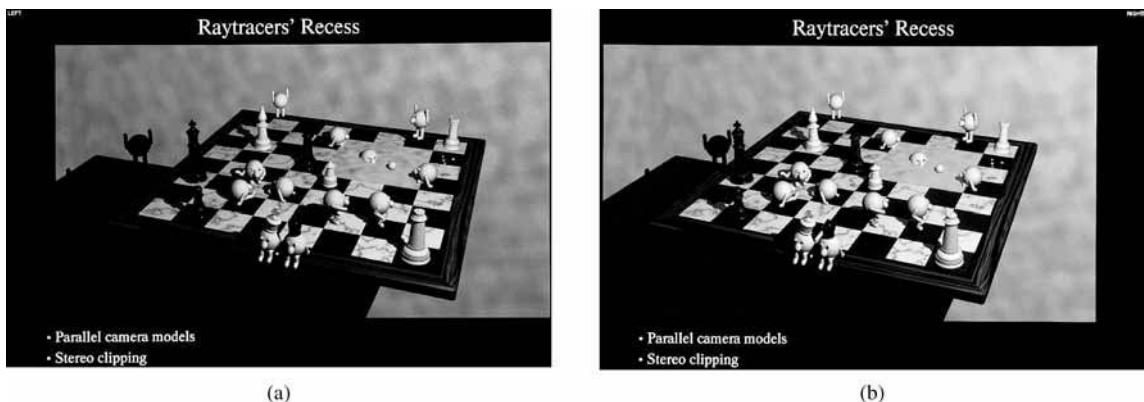
**FIGURA 2.18.** Visionado simulado de una proyección estereoscópica. (Cortesía de StereoGraphics Corporation.)

na generada por computadora para proyección estereoscópica. Para incrementar la comodidad en la visualización, se han eliminado las zonas de los bordes derecho e izquierdo de una escena que son visibles sólo por un ojo.

Una forma de producir un efecto estereoscópico en sistemas de barrido consiste en mostrar cada una de las dos vistas en ciclos alternos de refresco. La pantalla se ve a través de unas gafas, con cada lente diseñada

para actuar como un obturador rápido, el cual está sincronizado para impedir la visión de una de las imágenes. La Figura 2.20 muestra un par de gafas estereoscópicas construidas con obturadores de cristal líquido y un emisor de infrarrojos que sincroniza las gafas con las vistas de la pantalla.

La visualización estereoscópica también forma parte de los sistemas de realidad virtual, en los que los usuarios pueden introducirse en una escena e interactuar con el entorno. Un casco (Figura 2.21) que contiene



**FIGURA 2.19.** Un par de imágenes estereoscópicas.



**FIGURA 2.20.** Gafas para visualización de una escena estereoscópica y un emisor para sincronización por infrarrojos.



**FIGURA 2.21.** Casco utilizado en sistemas de realidad virtual. (*Cortesía de Virtual Research.*)



**FIGURA 2.22.** Un biólogo molecular analizando estructuras moleculares dentro de un sistema de realidad virtual llamado Trimension ReACTor. Los «guantes para pellizcar en el falso espacio» habilitan al científico para agarrar y redistribuir los objetos virtuales en una escena proyectada. (Cortesía de Silicon Graphics, Inc. and Trimension Systems ReACTor. © 2003 SGI. Todos los derechos reservados.)



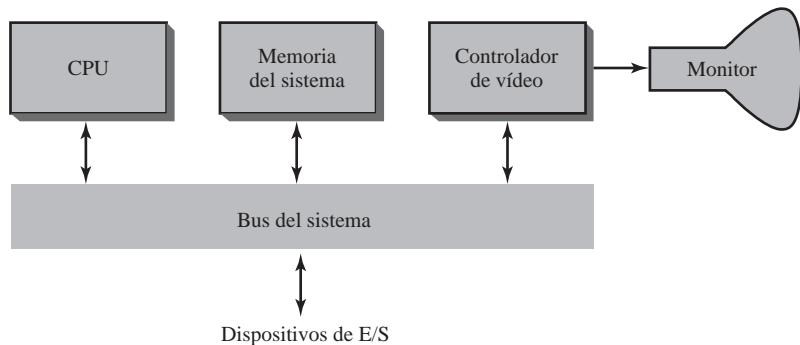
**FIGURA 2.23.** Un sistema de seguimiento por ultrasonidos empleado en gafas estereoscópicas para registrar los cambios en la posición de la cabeza del observador. (Cortesía de StereoGraphics Corporation.)

un sistema óptico para generar las vistas estereoscópicas se puede utilizar en conjunción con dispositivos de entrada interactivos para localizar y manipular los objetos en la escena. Un sistema de sensores dispuesto en el casco registra la posición del observador, para que se puedan ver las partes frontal y trasera de los objetos como si el observador «caminara alrededor» e interactuase con la pantalla. Otro método para crear entornos de realidad virtual consiste en utilizar proyectores para generar una escena dentro de una distribución de paredes, como en la Figura 2.22, donde el observador interactúa con la pantalla empleando unas gafas estereoscópicas y unos guantes de datos (Sección 2.4).

Se pueden organizar entornos de realidad virtual interactivos y de bajo coste empleando un monitor gráfico, gafas estereoscópicas y un dispositivo de seguimiento montado en la cabeza. La Figura 2.23 muestra un dispositivo de seguimiento por ultrasonidos con seis grados de libertad. El dispositivo de seguimiento está situado encima del monitor de vídeo y se emplea para registrar los movimientos de la cabeza, con el fin de que la posición del observador de la escena se cambie según los movimientos de la cabeza.

## 2.2 SISTEMAS DE BARRIDO DE LÍNEAS

Los sistemas interactivos gráficos de barrido emplean habitualmente varias unidades de procesamiento. Además de la unidad central de procesamiento, o UCP, un procesador de propósito específico, llamado **controlador de vídeo** o **controlador de pantalla**, se emplea para controlar el funcionamiento del dispositivo de pantalla. En la Figura 2.24 se muestra la organización de un sistema de barrido simple. En éste, el búfer de



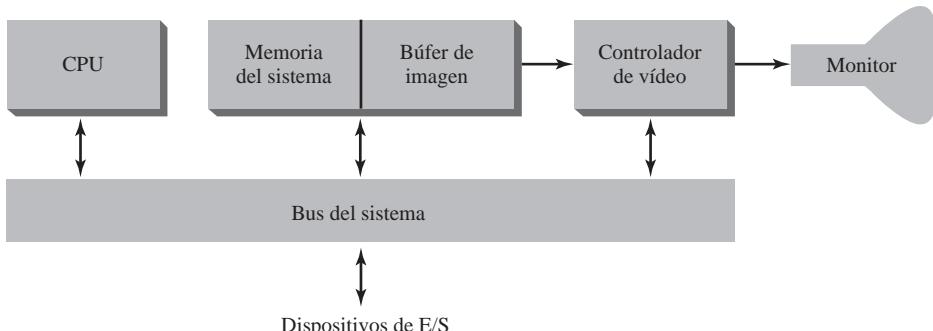
**FIGURA 2.24.** Arquitectura de un sistema gráfico simple de barrido.

imagen se puede colocar en cualquier parte del sistema de memoria, y el controlador de vídeo accede al búfer de imagen para refrescar la pantalla. Además del controlador de vídeo, los sistemas de barrido más sofisticados emplean otros procesadores tales como procesadores y aceleradores para implementar diversas operaciones gráficas.

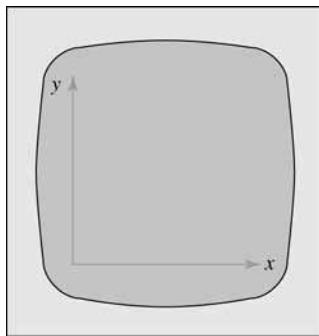
### Controlador de vídeo

La Figura 2.25 muestra la organización utilizada habitualmente en los sistemas de barrido. Una zona fija del sistema de memoria se reserva para el búfer de imagen y se permite que el controlador de vídeo acceda directamente a la memoria del búfer de imagen.

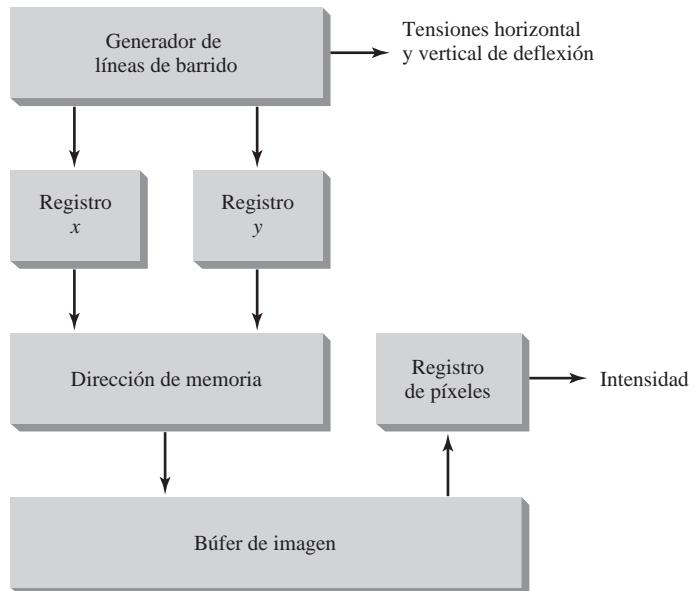
Las posiciones del búfer de imagen y sus correspondientes posiciones de pantalla se refieren en coordenadas cartesianas. En un programa de aplicación, utilizamos los comandos de un paquete de software gráfico, para establecer las coordenadas de posición de los objetos mostrados, relativas al origen del sistema de ejes cartesianos de referencia. A menudo, el origen de coordenadas se fija en la esquina inferior izquierda de la pantalla mediante los comandos del software, aunque podemos situar el origen en cualquier posición para una aplicación concreta. En la Figura 2.26 se muestra un sistema de coordenadas cartesianas de referencia con su origen en la esquina inferior izquierda de la pantalla. La superficie de la pantalla se representa como el primer cuadrante de un sistema de dos dimensiones, con los valores positivos del eje  $x$  que crecen de izquierda a derecha y con los valores positivos del eje  $y$  creciendo de abajo hacia arriba. A las posiciones de los píxeles se les asignan valores de  $x$  que varían desde 0 a  $x_{\max}$  a lo largo de la pantalla, de izquierda a derecha, y valores enteros de  $y$  que varían desde 0 hasta  $y_{\max}$ , de abajo hacia arriba. Sin embargo, el hardware realiza el refresco de la pantalla, así como algunos sistemas de software, utilizando como referencia para las posiciones de los píxeles la esquina superior izquierda de la pantalla.



**FIGURA 2.25.** Arquitectura de un sistema de barrido con una porción fija del sistema de memoria reservada para el búfer de imagen.



**FIGURA 2.26.** Ejes cartesianos de referencia con origen en la esquina inferior izquierda de un monitor de vídeo.



**FIGURA 2.27.** Funciones de refresco de un controlador básico de vídeo.

En la Figura 2.27, se representan mediante un diagrama las funciones básicas de refresco de un controlador de vídeo. Se utilizan dos registros para almacenar los valores de las coordenadas de los píxeles de la pantalla. Inicialmente, se establece el valor del registro  $x$  en 0 y el registro  $y$  con el valor de la línea superior de barrido. Se obtiene el contenido del búfer de imagen en esta posición de píxel y se utiliza para establecer la intensidad del haz del TRC. Entonces el registro  $x$  se incrementa en una unidad, y el proceso se repite para el siguiente píxel en la línea superior de barrido. Este procedimiento se repite para cada píxel a lo largo de la línea superior de barrido. Después de que se ha procesado el último píxel de las líneas superiores de barrido, se establece el valor del registro  $x$  en 0 y el registro  $y$  con el valor de la siguiente línea de barrido, situada debajo de la línea superior de barrido de la pantalla. Entonces se procesan los píxeles a lo largo de esta línea de barrido y el procedimiento se repite para cada sucesiva línea de barrido. Después de procesar todos los píxeles a lo largo de la línea inferior de barrido, el controlador de vídeo restablece los registros con la primera posición de píxel de la línea superior de barrido y el proceso de refresco comienza de nuevo.

Ya que la pantalla se debe refrescar a una velocidad de al menos 60 cuadros por segundo, puede que el procedimiento mostrado en la Figura 2.27 no pueda ser realizado por los chips de RAM habituales, si su tiempo de ciclo es demasiado lento. Para acelerar el procesamiento de los píxeles, los controladores de vídeo pueden obtener múltiples valores de píxel del búfer de refresco en cada pasada. Las múltiples intensidades de los píxeles se almacenan en registros separados y se utilizan para controlar la intensidad del haz del TRC para un grupo de píxeles adyacentes. Cuando se ha procesado este grupo de píxeles, se obtiene del búfer de imagen el siguiente bloque de valores de píxel.

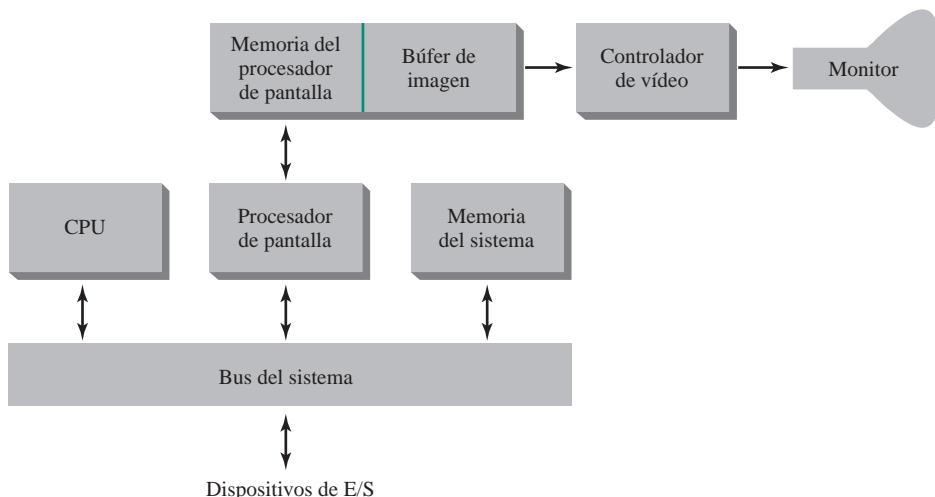
Un controlador de vídeo se puede diseñar para realizar otras funciones. El controlador de vídeo puede obtener los valores de los píxeles desde diferentes zonas de memoria en diferentes ciclos de refresco para varias aplicaciones. En algunos sistemas, por ejemplo, hay disponibles múltiples búferes de imagen para que un búfer se pueda utilizar para refresco y mientras los valores de los píxeles se están cargando en los otros búferes. Entonces el búfer de refresco actual puede intercambiar su función con otro de los búferes. Esto proporciona un mecanismo rápido para la generación de animaciones en tiempo real, por ejemplo, ya que se pue-

den cargar sucesivamente diferentes vistas del movimiento de los objetos en un búfer sin necesidad de interrumpir el ciclo de refresco. Otra tarea de los controladores de vídeo consiste en la transformación de bloques de píxeles, para que las zonas de pantalla se puedan ampliar, reducir, o mover de una posición a otra durante los ciclos de refresco. Además, el controlador de vídeo a menudo contiene una tabla de búsqueda, de modo que los valores de los píxeles en el búfer de imagen se utilizan para acceder a la tabla de búsqueda en lugar de para controlar la intensidad del haz del TRC directamente. Esto proporciona un rápido mecanismo para cambiar los valores de intensidad de la pantalla. Las tablas de búsqueda se estudian con más detalle en el Capítulo 4. Finalmente, algunos sistemas se diseñan para permitir al controlador de vídeo mezclar la imagen del búfer de imagen con una imagen procedente de una cámara de televisión o de otro dispositivo de entrada.

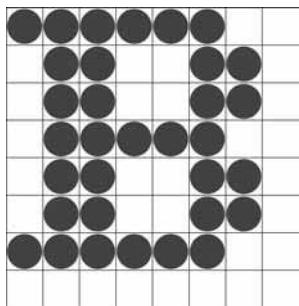
## Procesador de pantalla de líneas de barrido

La Figura 2.28 muestra un modo de organizar los componentes de un sistema de barrido que contiene un **procesador de pantalla** independiente, denominado a veces **controlador gráfico** o **coprocesador de pantalla**. El propósito del procesador de pantalla es liberar a la UCP de las tareas gráficas. Además de la memoria del sistema, se puede disponer de una zona de memoria independiente para el procesador de pantalla.

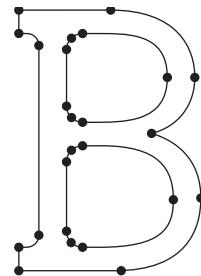
Una tarea importante del procesador de pantalla consiste en digitalizar una imagen, procedente de un programa de aplicación para transformarla en un conjunto de valores de píxeles para almacenarla en el búfer de cuadro. Este proceso de digitalización se conoce como **conversión de barrido**. Las órdenes gráficas que especifican líneas rectas y otros objetos geométricos sufren una conversión de barrido en un conjunto de puntos discretos que se corresponde con las posiciones de los píxeles de la pantalla. La conversión de barrido de un segmento de línea recta, por ejemplo, significa que tenemos que localizar las posiciones de los píxeles más cercanos a la trayectoria de la línea y almacenar el color para cada posición en el búfer de imagen. Se utilizan otros métodos similares para convertir mediante barrido otros objetos de una imagen. Los caracteres se pueden definir con rejillas rectangulares de píxeles, como en la Figura 2.29, o se pueden definir mediante sus líneas de contorno, como en la Figura 2.30. El tamaño de la matriz utilizada en la cuadrícula de los caracteres puede variar desde, aproximadamente, 5 por 7 hasta 9 por 12 o más en el caso de pantallas de calidad superior. Una cuadrícula de un carácter se muestra mediante la superposición del patrón de la rejilla rectangular en el búfer de imagen en unas coordenadas específicas. Cuando los caracteres se definen mediante sus contornos se realiza la conversión de barrido de las formas en el búfer de imagen mediante la localización de las posiciones de los píxeles más cercanos al contorno.



**FIGURA 2.28.** Arquitectura de un sistema gráfico de barrido con un procesador de pantalla.



**FIGURA 2.29.** Un carácter indefinido como una rejilla rectangular de las posiciones de sus píxeles.



**FIGURA 2.30.** Un carácter definido como un contorno.

Los procesadores de pantalla también se diseñan para realizar otras funciones adicionales. Dentro de estas funciones se incluye la generación de varios estilos de línea (discontinua, punteada o continua), mostrar áreas de color y aplicar transformaciones a los objetos de una escena. También, los procesadores de pantalla se diseñan habitualmente para actuar como interfaz con dispositivos de entrada interactivos, tales como un ratón.

Con el propósito de reducir los requisitos de memoria de los sistemas de barrido, se han ideado métodos para organizar el búfer de imagen como una lista enlazada y codificar la información del color. Un esquema de organización consiste en almacenar cada línea de barrido como un conjunto de pares de números. El primer número de cada par puede ser una referencia al valor del color y el segundo número puede especificar el número de píxeles adyacentes en la línea de barrido que se deben mostrar en ese color. Esta técnica, llamada **codificación de longitud de recorrido**, puede proporcionar un considerable ahorro de espacio de almacenamiento si la imagen está constituida principalmente por largos recorridos de un único color cada uno. Se puede seguir un planteamiento similar cuando los colores de los píxeles cambian de forma lineal. Otro planteamiento consiste en codificar el barrido como un conjunto de zonas rectangulares (**codificación de celdas**). Las desventajas de la codificación de recorridos son que los cambios de color son difíciles de registrar y los requisitos de almacenamiento aumentan a medida que las longitudes de los recorridos decrecen. Además, es difícil para el controlador de pantalla procesar el barrido cuando están involucrados muchos recorridos cortos. Por otra parte, el tamaño de búfer de imagen ya no es importante, puesto que el coste de la memoria se ha reducido considerablemente. No obstante, los métodos de codificación pueden ser útiles en el almacenamiento digital y en la transmisión de la información de la imagen.

## 2.3 ESTACIONES DE TRABAJO GRÁFICAS Y SISTEMAS DE VISUALIZACIÓN

La mayor parte de los monitores gráficos en la actualidad funcionan como pantallas de barrido de líneas y se utilizan habitualmente tanto los sistemas de TRC como las pantallas planas. Estaciones de trabajo gráficas en el rango desde pequeñas computadoras de propósito general hasta instalaciones con varios monitores, a menudo disponen de pantallas de visualización muy grandes. En las computadoras personales, las resoluciones de las pantallas varían desde, aproximadamente, 640 por 480 hasta 1280 por 1024, y la longitud de la diagonal de pantalla desde 12 pulgadas hasta más de 21 pulgadas. La mayoría de los sistemas de propósito general de hoy día poseen considerables capacidades de color, y muchos son sistemas de color total. En las estaciones de trabajo diseñadas específicamente para aplicaciones gráficas, la resolución de la pantalla puede variar desde 1280 por 1024 hasta, aproximadamente, 1600 por 1200, con una diagonal habitual de pantalla de 18 pulgadas o más. También se pueden obtener estaciones de trabajo comerciales con una gran variedad de dispositivos para aplicaciones específicas. La Figura 2.31 muestra las características de un tipo de estación de trabajo para un artista.

Los sistemas gráficos de alta definición, con resoluciones de hasta 2560 por 2048, se utilizan habitualmente para la generación de imágenes médicas, el control de tráfico aéreo, la simulación y el diseño CAD. En la Figura 2.32 se muestra una pantalla plana de 2048 por 2048.

Muchas estaciones de trabajo gráficas también incluyen amplias pantallas de visualización, a menudo con características especializadas. La Figura 2.33 muestra un sistema de pantalla amplia para visualización estereoscópica, y la Figura 2.34 es un sistema de pantalla panorámica multicanal.

Las pantallas multipanel se utilizan en una gran variedad de aplicaciones que no requieren áreas de visualización «del tamaño de una pared». Estos sistemas se diseñan para presentar gráficos en reuniones, conferencias, convenciones, demostraciones comerciales, almacenes al detalle, museos, también las hay en terminales de pasajeros. Una pantalla multipanel se puede utilizar para mostrar una gran vista de una única escena o varias imágenes individuales. Cada panel del sistema muestra una porción de la imagen total, como se muestra en la Figura 2.35. Las grandes pantallas gráficas también se pueden encontrar en forma de pantallas de



**FIGURA 2.31.** Una estación de trabajo de un artista, constituida por un monitor, un teclado, una tableta gráfica con cursor de mano, y una mesa ligera, además de dispositivos para el almacenamiento de datos y telecomunicaciones. (*Cortesía de DICOMED Corporation.*)



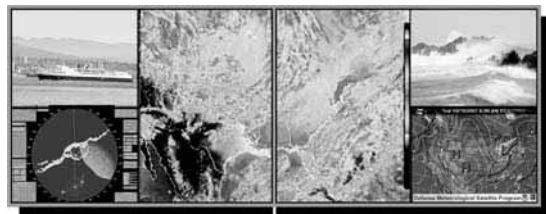
**FIGURA 2.32.** Un monitor gráfico de alta resolución (2048 por 2048). (*Cortesía de BarcoView.*)



**FIGURA 2.33.** El SGI Reality Center 200D, constituido por un ImmersaDesk R2 que muestra en una gran pantalla estereoscópica una vista de los contornos de presión de un torrente sanguíneo vascular simulado y superpuesto sobre un conjunto de datos anatómicos y dibujados con volumen. (*Cortesía de Silicon Graphics, Inc. y el catedrático Charles Taylor de la Universidad de Stanford. © 2003 SGI. Todos los derechos reservados.*)



**FIGURA 2.34.** Una vista en pantalla panorámica de un sistema molecular mostrado en un SGI Reality Center 3300W de tres canales. (Cortesía de Silicon Graphics, Inc. y Molecular Simulations. © 2003 SGI. Todos los derechos reservados.)



**FIGURA 2.35.** Un sistema de pantalla multipanel llamado «Super Wall (supermuro)». (Cortesía de RGB Spectrum.)



**FIGURA 2.36.** Un estudio de seguridad de un estado mostrado en un sistema de gran pantalla de visualización curvada. (Cortesía de Silicon Graphics, Inc. © 2003. Todos los derechos reservados.)

visualización curvadas, tales como el sistema de la Figura 2.36. Un sistema de pantalla curvada grande puede ser útil para la visualización por un grupo de personas que estudian una aplicación gráfica concreta, en las Figuras 2.37 y 2.38 se muestran ejemplos de tales pantallas. Un centro de control, constituido por una batería de monitores estándar, permite a un operador visualizar partes en la pantalla grande y controlar el audio, el vídeo, la iluminación y los sistemas de proyección mediante el empleo de un menú de pantalla táctil. Los sistemas con proyectores proporcionan una pantalla multicanal y sin junturas que incluye la fusión de los bor-

des, la corrección de la distorsión y el balance del color. Un sistema de sonido envolvente se utiliza para proporcionar el entorno de audio. La Figura 2.39 muestra un sistema de visualización panelado de 360° del simulador de la torre de control de la NASA, el cual se emplea para el entrenamiento y para probar modos de resolver problemas de tráfico aéreo y de rodadura en los aeropuertos.



**FIGURA 2.37.** Un sistema gráfico de pantalla curvada que muestra un paseo interactivo por una planta de gas natural. (Cortesía de Silicon Graphics, Inc., Trimension Systems, y el CadCentre, Cortaillod, Suiza. © 2003 SGI. Todos los derechos reservados.)



**FIGURA 2.38.** Una visualización geofísica presentada en una pantalla semicircular de 25 pies, que proporciona un campo de visión de 160° en horizontal y 40° en vertical. (Cortesía de Silicon Graphics, Inc., Landmark Graphics Corporation, y Trimension Systems. © 2003 SGI. Todos los derechos reservados.)



**FIGURA 2.39.** La pantalla de visualización de 360° del simulador de la torre de control del aeropuerto de la NASA, llamado el FutureFlight Central Facility. (Cortesía de Silicon Graphics, Inc. y NASA. © 2003 SGI. Todos los derechos reservados.)

## 2.4 DISPOSITIVOS DE ENTRADA

---

Las estaciones de trabajo gráficas pueden utilizar diversos dispositivos para la entrada de datos. La mayoría de los sistemas disponen de un teclado y uno o más dispositivos adicionales diseñados específicamente para poder realizar entradas interactivas. Entre estos dispositivos se incluyen el ratón, *trackball*, *spaceball* y las palancas de mando (*joystick*). Otros dispositivos de entrada, que se utilizan en aplicaciones particulares son los digitalizadores, las ruedas de selección (*dials*), las cajas de botones, los guantes de datos, los paneles táctiles, los escáneres de imagen y los sistemas de voz.

### Teclados, cajas de botones y ruedas de selección

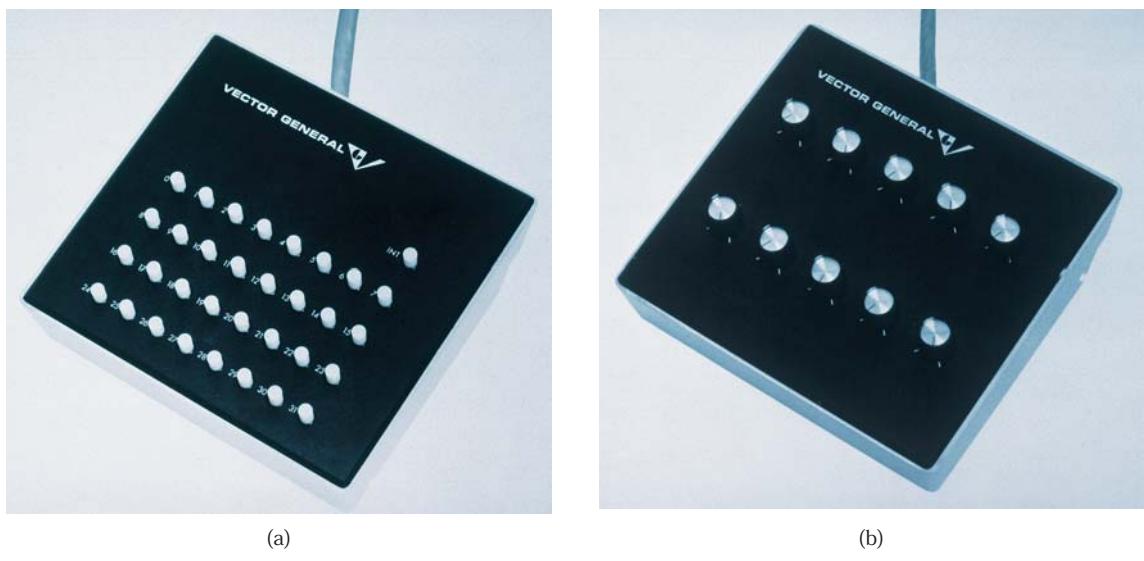
Un **teclado** alfanumérico de un sistema gráfico se utiliza primordialmente como dispositivo de entrada de cadenas de texto, para ejecutar determinados comandos y seleccionar opciones de menú. El teclado es un dispositivo eficiente para introducir datos no gráficos como etiquetas de imágenes asociadas a una pantalla gráfica. Los teclados también pueden estar provistos de características que faciliten la entrada de coordenadas de pantalla, selecciones de menús o funciones gráficas.

Habitualmente, en los teclados de propósito general hay disponibles teclas de cursor y teclas de función. Las teclas de función permiten a los usuarios seleccionar operaciones de uso habitual con la pulsación de una única tecla y las teclas del cursor permiten seleccionar un objeto o una posición situando el cursor de pantalla. Un teclado puede también contener otras clases de dispositivos de posicionamiento del cursor, tales como un *trackball* o una palanca de mando, junto con un pequeño teclado numérico para introducir rápidamente datos numéricos. Además de estas características, algunas teclados se han diseñado ergonómicamente (Figura 2.40) con el fin de proporcionar elementos que alivien la fatiga del usuario.

En tareas especializadas, la entrada en aplicaciones gráficas puede proceder de un conjunto de botones, botones de selección, o interruptores que seleccionan valores u operaciones gráficas personalizadas. La Figura 2.41 muestra un ejemplo de una **caja de botones** y un conjunto de **ruedas de selección**. Los botones y los interruptores se utilizan habitualmente para funciones de entrada predefinidas, y las ruedas de selección para introducir valores escalares. Los valores numéricos que se encuentran dentro de unos límites definidos se seleccionan mediante rotaciones de las ruedas de selección. Para medir la rotación de una rueda de selección se utiliza un potenciómetro, el cual se convierte en el correspondiente valor numérico.



**FIGURA 2.40.** Teclado diseñado ergonómicamente y dotado de reposamuñecas separable. La inclinación de cada parte del teclado se puede ajustar de forma separada. Al lado del teclado se muestra un ratón de un único botón, que dispone de un cable para conectarlo a la computadora. (*Cortesía de Apple Computer, Inc.*)



**FIGURA 2.41.** Una caja de botones (a) y un conjunto de ruedas de selección (b). (*Cortesía de Vector General.*)

## Ratones

La Figura 2.40 presenta el típico diseño de un **ratón** de un solo botón, el cual es un pequeño dispositivo de mano, que se mueve habitualmente sobre una superficie plana para posicionar el cursor de la pantalla. Para registrar la cantidad y la dirección del movimiento se utilizan ruedas o rodamientos en la parte inferior del ratón. Otro método para detectar el movimiento del ratón es la utilización de un sensor óptico. En algunos sistemas ópticos, el ratón se mueve sobre una alfombrilla especial que contiene una retícula de líneas verticales y horizontales. El sensor óptico detecta los movimientos sobre las líneas de la retícula. Otros ratones ópticos pueden funcionar sobre cualquier superficie. Y algunos son inalámbricos y se comunican con el procesador de la computadora empleando tecnología de radio digital.

Dado que un ratón se puede pulsar y liberar en otra posición sin cambiar el movimiento del cursor, se utiliza para realizar cambios relativos en la posición del cursor en la pantalla. En la parte superior del ratón se

dispone de uno, dos, tres o cuatro botones para señalar la ejecución de funciones, tales como registrar la posición del cursor o invocar una función. La mayoría de los sistemas gráficos de propósito general incluyen en la actualidad un ratón y un teclado como dispositivos de entrada principal.

Se pueden incluir características adicionales en el diseño básico de un ratón para incrementar el número de parámetros de entrada disponibles. El ratón Z de la Figura 2.42 dispone de tres botones, una rueda de pulgar en su lateral, un *trackball* en su parte superior y una bola de ratón estándar por debajo. Este diseño proporciona seis grados de libertad para seleccionar posiciones en el espacio, rotaciones y otros parámetros. Con el ratón Z, podemos seleccionar un objeto mostrado en un monitor de vídeo, rotarlo y moverlo en cualquier dirección. También podríamos utilizar el ratón Z para modificar la posición de visualización y la orientación en una escena tridimensional. Entre las aplicaciones del ratón Z se incluyen la realidad virtual, los sistemas CAD y la animación.

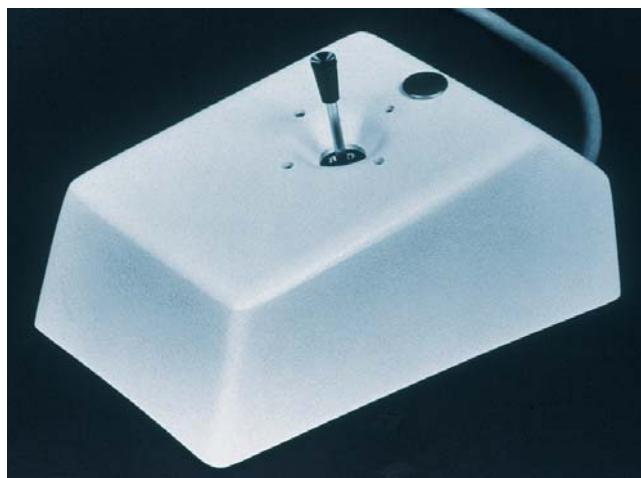
### *Trackballs y spaceballs*

Un *trackball* es un dispositivo de bola que se puede girar con los dedos o la palma de la mano para producir el movimiento del cursor de pantalla. Unos potenciómetros, conectados a la bola, miden lo que hay que girar y en qué dirección. Los teclados de las computadoras portátiles están a menudo equipados con un *trackball* para eliminar el espacio adicional que requiere un ratón. Un *trackball* se puede montar también en otros dispositivos, tales como el ratón Z mostrado en la Figura 2.42, o se puede obtener como un accesorio independiente que contiene dos o tres botones de control.

El *spaceball* se puede considerar como una ampliación del *trackball* de dos dimensiones, que proporciona seis grados de libertad. A diferencia del *trackball*, no se mueve realmente. Unas galgas extensiometrías



**FIGURA 2.42.** El ratón Z dispone de tres botones, una bola de ratón por debajo, una rueda de pulgar en el lateral y un *trackball* en su parte superior. (Cortesía de Multi-point Technology Corporation.)



**FIGURA 2.43.** Un joystick móvil. (Cortesía de CalComp Group, Sanders Associates, Inc.)

miden la cantidad de presión aplicada al *spaceball* para proporcionar los datos de entrada para definir el posicionamiento espacial y la orientación a medida que se empuja y tira de la bola en varias direcciones. Los *spaceballs* se utilizan para el posicionamiento tridimensional y en las operaciones de selección en sistemas de realidad virtual, modelado, animación, CAD y otras aplicaciones.

## Joysticks

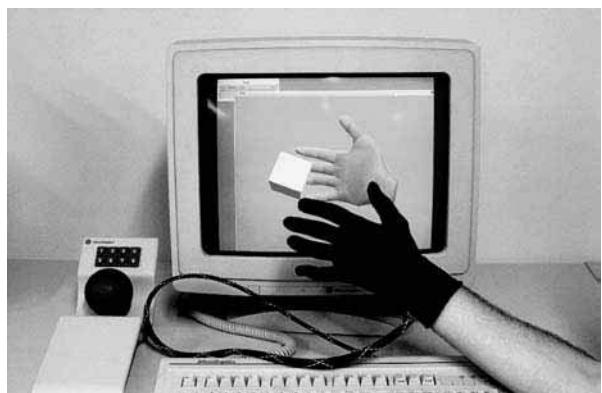
Un *joystick* o **palanca de mando** es otro dispositivo de posicionamiento, el cual consiste en una palanca vertical y pequeña (llamada *stick*) montada sobre una base. El *joystick* se usa para dirigir el cursor por la pantalla. La mayoría de los *joysticks*, tales como el mostrado en la Figura 2.43, seleccionan posiciones en la pantalla con movimientos verdaderos de la palanca; y otros responden a la presión sobre la palanca. Algunos se montan sobre un teclado y otros se diseñan como dispositivos independientes.

La distancia que se mueve la palanca en cualquier dirección respecto de su posición de equilibrio se corresponde con el movimiento relativo del cursor de la pantalla en dicha dirección. Unos potenciómetros montados en la base de la palanca miden la cantidad de movimiento, y unos muelles devuelven la palanca a su posición en central cuando ésta se libera. Estos dispositivos disponen de uno o más botones que se pueden programar para que funcionen como interruptores de entrada para invocar acciones que se deben ejecutar una vez que se ha seleccionado una posición en la pantalla.

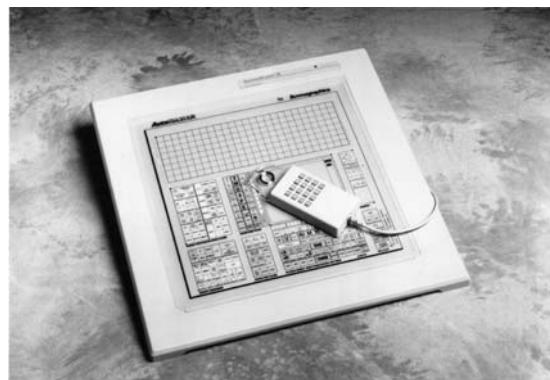
En otros tipos de *joysticks* móviles, la palanca se utiliza para activar interruptores que provocan el movimiento del cursor de la pantalla a una velocidad constante en la dirección seleccionada. A veces, se dispone de ocho interruptores, dispuestos formando un círculo, para que la palanca pueda seleccionar una cualquiera de las ocho direcciones para el movimiento del cursor. Las palancas de mando sensibles a la presión, también llamados *joystick isométricos*, disponen de una palanca que no se puede mover. En estos dispositivos se mide un empujón o un tirón de la palanca mediante el empleo de galgas extensiométricas y se convierte en movimiento del cursor de la pantalla y en la dirección en que se aplica la presión.

## Guantes de datos

La Figura 2.44 muestra un **guante de datos** que se puede utilizar para agarrar un «objeto virtual». El guante dispone de una serie de sensores que detecta los movimientos de la mano y de los dedos. Para proporcionar información acerca de la posición y de la orientación de la mano se utiliza un acoplamiento electromagnético entre antenas de transmisión y antenas de recepción. Cada una de las antenas de transmisión y recepción se puede considerar constituida por un conjunto de tres bobinas mutuamente perpendiculares, que forman un sistema de referencia cartesiano tridimensional. Los datos de entrada procedentes del guante se utilizan para posicionar o manipular objetos en una escena virtual. La proyección bidimensional de la escena se puede visualizar en un monitor de vídeo, o su proyección tridimensional se puede visualizar con un casco.



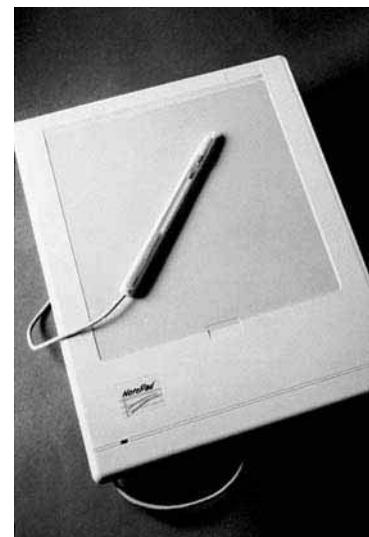
**FIGURA 2.44.** Una escena de realidad virtual mostrada en un monitor de vídeo bidimensional, con entrada de datos procedentes de un guante de datos y un *spaceball*. (Cortesía de The Computer Graphics Center, Darmstadt, Alemania.)



**FIGURA 2.45.** La tableta de escritorio SummaSketch III con dieciséis botones y cursor manual. (*Cortesía de Summagraphics Corporation.*)



**FIGURA 2.46.** La tableta Microgrid III con dieciséis botones y cursor manual, diseñada para digitalizar grandes dibujos. (*Cortesía de Summagraphics Corporation.*)



**FIGURA 2.47.** La tableta de escritorio NotePad con pluma. (*Cortesía de CalComp Digitizer Division, una división de CalComp, Inc.*)

## Digitalizadores

Un **digitalizador** es un dispositivo que se utiliza habitualmente para dibujar, pintar o seleccionar posiciones de forma interactiva. Estos dispositivos se pueden diseñar para introducir coordenadas de espacios bidimensionales o tridimensionales. En aplicaciones de ingeniería o de arquitectura, un digitalizador se utiliza a menudo para escanear un dibujo o un objeto e introducir un conjunto discreto de coordenadas. Las posiciones introducidas se unen entonces con segmentos para generar una aproximación de una curva o una superficie.

La **tableta gráfica** (también denominada *tableta de datos*) es un tipo de digitalizador, que se utiliza para introducir coordenadas bidimensionales mediante la activación de un cursor manual o una pluma en posiciones concretas de una superficie plana. Un cursor manual dispone de una cruz para señalizar posiciones, mientras una pluma es un dispositivo con forma de lapicero que se apunta en posiciones de la tableta. Las Figuras 2.45 y 2.46 muestran ejemplos de tabletas de escritorio y para apoyar en el suelo, que utilizan cursosres manua-

les y que disponen de dos, cuatro o dieciséis botones. En las Figuras 2.47 y 2.48 se muestran ejemplos de tabletas con plumas para la entrada de datos. El sistema de digitalización para artistas de la Figura 2.48 utiliza la resonancia electromagnética para detectar la posición tridimensional de la pluma. Esto permite a un artista producir diferentes trazos de pincel mediante la aplicación de diferentes presiones sobre la superficie de la tableta. El tamaño de la tableta varía desde 12 por 12 pulgadas en los modelos de escritorio hasta 44 por 60 pulgadas o más en los modelos que se apoyan en el suelo. Las tabletas gráficas proporcionan un método altamente preciso para seleccionar coordenadas, con una precisión que varía desde cerca de 0,2 mm en modelos de escritorio hasta cerca de 0,05 mm o menos en modelos mayores.

Muchas de las tabletas gráficas se construyen con una cuadrícula rectangular de cables embebidos en la superficie de la tableta. Se generan pulsos electromagnéticos en secuencia a lo largo de los cables, y se induce una señal eléctrica en una bobina de cable en la pluma activada o en el cursor para registrar la posición en la tableta. Dependiendo de la tecnología, se puede utilizar la amplitud de la señal, pulsos codificados o cambios de fase para determinar la posición en la tableta.

Una *tableta acústica* (o *sónica*) utiliza las ondas sonoras para detectar la posición de la pluma. Se pueden utilizar tiras de micrófonos o micrófonos puntuales para detectar el sonido emitido por una chispa eléctrica desde la pluma. La posición de la pluma se calcula midiendo el tiempo que tarda en llegar el sonido generado a las diferentes posiciones de los micrófonos. Una ventaja de las tabletas acústicas bidimensionales es que los micrófonos se pueden situar en cualquier superficie para formar el área de trabajo de la tableta. Por ejemplo, los micrófonos se podrían situar en la página de un libro mientras se digitaliza una figura en esta página.

Los digitalizadores tridimensionales emplean transmisiones sónicas o electromagnéticas para registrar las posiciones. Un método de transmisión electromagnética es similar al empleado en los guantes de datos: un acoplamiento entre el transmisor y el receptor se utiliza para calcular las posiciones de una pluma a medida que ésta se mueve sobre la superficie de un objeto. La Figura 2.49 muestra un digitalizador registrando en las posiciones de la superficie de un objeto tridimensional. A medida que se seleccionan los puntos de un objeto no metálico, se muestra en la pantalla de una computadora el contorno alámbrico de su superficie. Una vez que se ha construido la superficie exterior del objeto, ésta se puede representar empleando efectos de iluminación para crear una imagen realista del objeto. La resolución de este sistema varía desde 0,8 mm hasta 0,08 mm, dependiendo del modelo.



**FIGURA 2.48.** Un sistema digitalizador para artistas, con una pluma inalámbrica sensible a la presión. (Cortesía de Wacom Technology Corporation.)



**FIGURA 2.49.** Un sistema de digitalización tridimensional para su uso con computadoras de Apple Macintosh. (Cortesía de Mira Imaging.)



**FIGURA 2.50.** Un escáner de mano que se puede utilizar para introducir texto o imágenes gráficas. (Cortesía de Thunderware, Inc.)

## Escáneres de imagen

Se pueden almacenar dibujos, gráficos, fotografías o texto para procesarlos con una computadora empleando un **escáner de imagen**. La información se almacena al pasar un mecanismo de exploración óptico sobre éstos. Los niveles de escala de grises o los colores se registran y se almacenan en una matriz. Una vez que se dispone de una representación interna de una imagen, podemos aplicar transformaciones para rotar, cambiar de escala o recortar la imagen en una zona particular de la pantalla. También podemos aplicar diversos métodos de procesamiento de imágenes para modificar la matriz de representación de la imagen. Después de introducir texto mediante un escáner, se pueden realizar varias operaciones de edición sobre los documentos almacenados. Los escáneres están disponibles en una gran variedad de tamaños y capacidades. En la Figura 2.50 se muestra un pequeño escáner de mano, mientras en las Figuras 2.51 y 2.52 se muestran modelos más grandes.

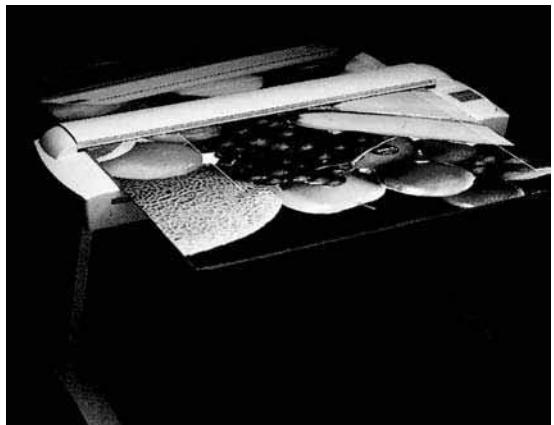
## Paneles táctiles

Como su propio nombre indica, los **paneles táctiles** permiten visualizar objetos o posiciones de pantalla que se pueden seleccionar con el toque de un dedo. Una aplicación habitual de los paneles táctiles es la selección de opciones de procesamiento que se representan como un menú de iconos gráficos. Algunos monitores, como las pantallas de plasma de la Figura 2.53, se diseñan con pantallas táctiles. Otros sistemas se pueden adaptar a la entrada táctil mediante la colocación de un dispositivo transparente (Figura 2.54), que contiene un mecanismo sensible al tacto, sobre un monitor de video, y los datos de entrada táctiles se pueden registrar mediante el empleo de métodos ópticos, eléctricos, o acústicos.

Los paneles táctiles ópticos emplean diodos emisores de luz (LED) infrarroja dispuestos en línea a lo largo del borde vertical y el borde horizontal del cuadro. Estos detectores se utilizan para registrar qué haces se interrumpen cuando el panel se toca. Los dos haces que se cruzan y que son interrumpidos identifican las



**FIGURA 2.51.** Escáneres de escritorio: (a) escáner de tambor y (b) escáner de superficie plana. (Cortesía de Aztek, Inc., Lago Forest, California.)

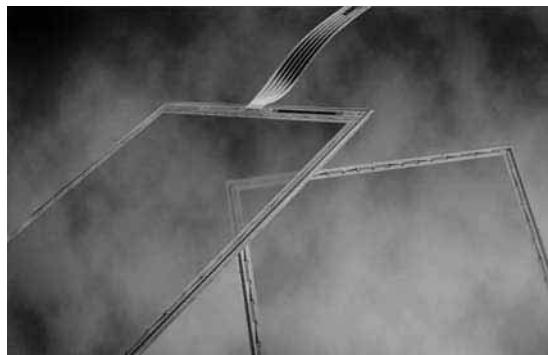


**FIGURA 2.52.** Un escáner de gran formato. (Cortesía de Aztek, Inc., Lago Forest, California.)

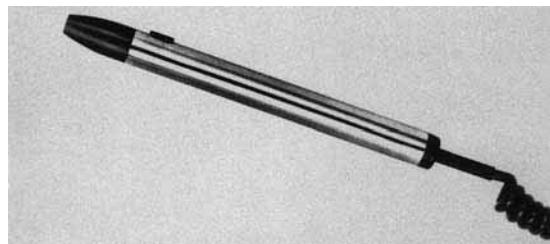


**FIGURA 2.53.** Paneles de plasma con pantallas táctiles. (Cortesía de Photonics Systems.)

coordenadas horizontal y vertical de la posición seleccionada. Las posiciones se pueden seleccionar con una precisión de, aproximadamente, 1/4 de pulgada. Cuando los LED están espaciados una distancia muy



**FIGURA 2.54.** Paneles táctiles resistivos de superposición. (Cortesía de Elo TouchSystems, Inc.)



**FIGURA 2.55.** Un lápiz óptico con un botón. (Cortesía de Interactive Computer Products.)

pequeña, es posible interrumpir dos haces verticales y dos horizontales de forma simultánea. En este caso, se registra la posición media entre los dos haces interrumpidos. La frecuencia a la que funcionan los LED corresponde al infrarrojo, con el fin de que la luz no sea visible por el usuario.

Un panel táctil eléctrico se construye con dos placas transparentes separadas una distancia pequeña. Una de las placas está recubierta con un material conductor y la otra placa está recubierta con un material resistivo. Cuando se toca la placa exterior, ésta se ve forzada a hacer contacto con la placa interior. Este contacto crea una tensión a lo largo de la placa resistiva que se transforma en las coordenadas de la posición seleccionada de la pantalla.

En los paneles táctiles acústicos se generan ondas de sonido de alta frecuencia en direcciones horizontales y verticales a lo largo de una placa de cristal. El toque de la pantalla provoca que una parte de cada onda se refleje desde el dedo hacia los emisores. La posición de la pantalla en el punto de contacto se calcula a partir de la medición del intervalo de tiempo que transcurre entre la emisión de cada onda y su reflexión hacia el emisor.

## Lapiceros ópticos

La Figura 2.55 muestra el diseño de un tipo de **lápiz óptico**. Estos dispositivos con forma de lapicero se utilizan para seleccionar posiciones de la pantalla mediante la detección de la luz procedente de los puntos de la pantalla de TRC. Son sensibles a la pequeña ráfaga de luz emitida por el recubrimiento de fósforo en el instante que el haz de electrones choca en un punto concreto. Otras fuentes de luz, tales como la luz ambiental de la habitación, no se detectan habitualmente por un lápiz óptico. Un lápiz óptico activado, que está situado en un punto de la pantalla cuando el haz de electrones lo está iluminando, genera un pulso eléctrico que provoca que las coordenadas del haz de electrones se registren. Como en los demás dispositivos de posicionamiento del cursor, las coordenadas registradas por un lapicero óptico se pueden usar para posicionar un objeto o seleccionar una opción de procesamiento.

Aunque todavía se utilizan los lapiceros ópticos, no son tan populares como lo fueron hace tiempo, ya que presentan algunas desventajas comparados con otros dispositivos de entrada que se han desarrollado. Por ejemplo, cuando se sitúa un lápiz óptico en un punto de la pantalla, parte de la imagen de la pantalla queda oscurecida por la mano y el lapicero. Además, un uso prolongado de un lápiz óptico puede causar fatiga en el brazo. También, los lapiceros ópticos requieren implementaciones especiales de algunas aplicaciones, ya que no puede detectar posiciones dentro de zonas negras. Para que un lapicero óptico permita seleccionar posiciones en cualquier zona de la pantalla, la intensidad de la luz emitida por cada píxel debe ser distinta de cero dentro de dicha zona. Además, a veces los lapiceros ópticos producen falsas lecturas debido a la iluminación ambiental de la habitación.



**FIGURA 2.56.** Un sistema de reconocimiento de voz (Cortesía de Threshold Technology, Inc.)

## Sistemas de voz

Los reconocedores de voz se utilizan en algunas estaciones de trabajo gráficas como dispositivos de entrada para órdenes de voz. La entrada procedente de un **sistema de voz** se puede utilizar para iniciar la ejecución de operaciones gráficas o para la introducción de datos. Estos sistemas funcionan mediante la comparación de la entrada frente a un diccionario predefinido de palabras y frases.

Un diccionario se crea pronunciando varias veces las palabras correspondientes a los comandos. A continuación, el sistema analiza cada palabra y establece un diccionario de patrones de frecuencia de las palabras, junto con las correspondientes funciones que se deben llevar a cabo. Después, cuando se da una orden de voz, el sistema busca una coincidencia en el patrón de frecuencia. Para cada usuario del sistema se necesita un diccionario independiente. En un sistema de voz la entrada se realiza habitualmente hablando a un micrófono montado en unos cascos, como el mostrado en la Figura 2.56, y éste se diseña para minimizar el ruido ambiental. Los sistemas de voz presentan ventajas frente a otros dispositivos de entrada, ya que el usuario no necesita cambiar su atención de un dispositivo a otro para introducir un comando.

## 2.5 DISPOSITIVOS DE COPIA IMPRESA

---

Podemos obtener una copia impresa de nuestras imágenes en varios formatos. Para realizar presentaciones o archivar, podemos enviar archivos de imagen a dispositivos a una empresa con servicio de reprografía que producirá transparencias, diapositivas de 35 mm, o películas. También podemos pasar nuestras imágenes a papel enviando la salida gráfica a una impresora o un trazador.

La calidad de las imágenes obtenidas mediante un dispositivo de salida depende del tamaño del punto y del número de puntos por pulgada, o líneas por pulgada, que se pueden mostrar. Para producir patrones más suaves, las impresoras de mayor calidad cambian las posiciones de los puntos para que los puntos adyacentes se superpongan.

La salida en las impresoras se produce o por métodos de impacto o sin él. Las impresoras de *impacto* presionan los caracteres formados contra una cinta impregnada con tinta sobre un papel. Una impresora de línea es un ejemplo de dispositivo de impacto, en la que los caracteres están montados sobre bandas, cadenas, tambores, o ruedas. Las impresoras y los trazadores *que no son de impacto* utilizan técnicas basadas en láser, inyectores de tinta, métodos electrostáticos y métodos electrotérmicos para obtener imágenes sobre el papel.

Las impresoras de impacto de caracteres, habitualmente, disponen de una cabeza de impresión de *matriz de puntos* que contiene una matriz rectangular de agujas de alambre que sobresalen, en las que el número de agujas depende de la calidad de la impresora. Los caracteres individuales o los patrones gráficos se obtienen al retraer ciertas agujas para que las agujas restantes formen el patrón que se tiene que imprimir. La Figura 2.57 muestra una imagen impresa con una impresora de matriz de puntos.

En un dispositivo *láser*, un haz láser crea una distribución de carga sobre un tambor rotante recubierto con un material fotoeléctrico, como el selenio. El tóner se aplica al tambor y luego se transfiere al papel. Los métodos de *inyección de tinta* producen la salida mediante el empleo de chorros a presión de tinta, dispuestos en

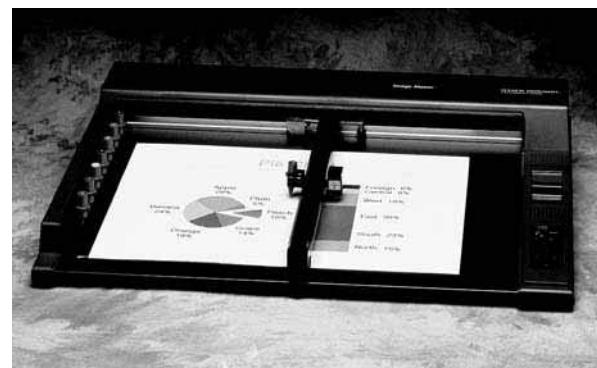
filas horizontales a lo largo del rollo de papel enrollado en un tambor. El chorro de tinta cargada eléctricamente se desvía mediante un campo eléctrico para producir patrones de matriz de puntos. Y un dispositivo *electrostático* coloca una carga negativa sobre el papel, según una línea completa cada vez a lo largo de la hoja de papel. Entonces se expone el papel a un tóner cargado positivamente. Esto provoca que el tóner sea atraído hacia las zonas cargadas negativamente, donde se adhiere para producir la salida especificada. Otra tecnología de salida es la impresora *electrotérmica*. En estos dispositivos, se aplica calor a la cabeza de impresión de matriz de puntos para producir la salida de patrones en un papel sensible al calor.

Podemos obtener una salida en colores limitados en impresoras de impacto mediante la utilización de cintas de diferentes colores. Los dispositivos que no son de impacto utilizan varias técnicas para combinar tres pigmentos de color diferentes (cian, magenta y amarillo), para producir una gama de patrones de color. Los dispositivos láser y electrostáticos depositan los tres pigmentos en pasadas independientes; y los métodos de inyección ponen los tres colores simultáneamente en una única pasada a lo largo de cada línea de impresión.

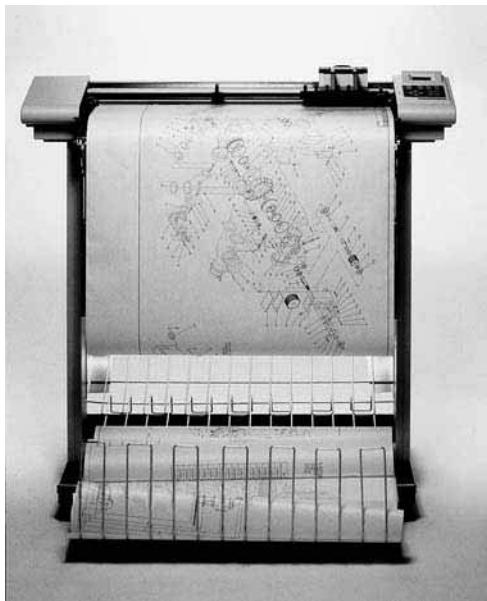
Los borradores de diseños y otros dibujos se generan habitualmente con trazadores de inyección de plumillas. Un trazador de plumillas dispone de una o más plumillas montadas sobre un carro, que abarca una hoja de papel. Las plumillas de diferentes colores y grosor se utilizan para producir una gran variedad de rellenos y estilos de línea. Las plumillas que se pueden utilizar con un trazador de plumillas son de tinta húmeda y de punta de fieltro. El papel del trazador puede estar desenrollado o enrollado sobre un tambor o cinta. Los rodillos pueden ser móviles o estacionarios, mientras la plumilla se mueve hacia adelante y hacia atrás a lo largo de la barra. El papel se mantiene en su posición utilizando dispositivos mecánicos, vacío o una carga electrostática. En la Figura 2.58 se muestra un ejemplo de trazador de plumillas de sobremesa y superficie plana. En la Figura 2.59 se muestra un trazador de plumillas de mayor tamaño y con alimentador de rodillo.



**FIGURA 2.57.** Una imagen generada con una impresora de matriz de puntos, que ilustra cómo la densidad de patrones de puntos se puede variar para producir zonas claras y zonas oscuras. (Cortesía de Apple Computer, Inc.)



**FIGURA 2.58.** Un trazador de escritorio con plumillas con una resolución de 0,025 mm. (Cortesía de Summagraphics Corporation.)



**FIGURA 2.59.** Trazador de plumillas con alimentador de rodillo y de gran tamaño que dispone de un cargador automático de 8 plumillas multicolores y de una resolución de 0,0127 mm. (Cortesía de Summagraphics Corporation.)

## 2.6 REDES GRÁFICAS

---

Hasta el momento, hemos considerado principalmente aplicaciones gráficas en sistemas aislados y con un único usuario. Sin embargo, los entornos multiusuario y las redes de computadores son en la actualidad elementos comunes en muchas aplicaciones gráficas. Varios recursos, tales como procesadores, impresoras, trazadores y archivos de datos se pueden distribuir en la red y se pueden compartir entre múltiples usuarios.

Un monitor gráfico de una red se denomina generalmente **servidor gráfico**, o simplemente **servidor**. El monitor a menudo incluye dispositivos de entrada tales como un teclado y un ratón o un *trackball*. En este caso, el sistema puede proporcionar entrada, así como puede funcionar como servidor de salida. La computadora de la red que ejecuta un programa de aplicación de gráficos se llama el **cliente** y la salida del programa se muestra en el servidor. Una estación de trabajo que incluye procesadores, así como un monitor y dispositivos de entrada, puede funcionar como servidor y como cliente.

Cuando el funcionamiento es en red, una computadora cliente transmite las instrucciones para mostrar una imagen al monitor (servidor). Habitualmente, esta labor se lleva a cabo acumulando las instrucciones en paquetes antes de transmitirlas, en lugar de enviar las instrucciones gráficas individuales una a una por la red. Por tanto, los paquetes de software gráfico a menudo también disponen de comandos relacionados con la transmisión de paquetes, así como comandos para la creación de imágenes.

## 2.7 GRÁFICOS EN INTERNET

---

Una gran cantidad del desarrollo gráfico se está realizando en la actualidad en **Internet**, que es una red global de redes de computadoras. Las computadoras en Internet se comunican mediante el protocolo TCP/IP (*Transmission Control Protocol/Internet Protocol*). Además, la **World Wide Web** proporciona un sistema de hipertexto que permite a los usuarios localizar y ver documentos que pueden contener texto, gráficos y sonidos. Los recursos, tales como los archivos gráficos, se identifican mediante un localizador de recursos uniforme (*Uniform Resource Locator*, URL). Cada URL, denominado a veces localizador de recursos universal, contiene dos partes: (1) el protocolo de transferencia del documento y (2) el servidor que contiene el docu-

mento y, opcionalmente, la localización (directorio) en el servidor. Por ejemplo, el URL <http://www.siggraph.org> indica un documento que se debe transferir utilizando el protocolo de transferencia de hipertexto (*hypertext transfer protocol; http*) y que el servidor es [www.siggraph.org](http://www.siggraph.org), el cual es la página principal del grupo SIGGRAPH (Special Interest Group in Graphics) de la Association for Computing Machinery. Otro tipo habitual de URL comienza por *ftp://*. Este tipo identifica un «sitio ftp», de donde se pueden descargar programas u otra clase de archivos mediante el uso del protocolo de transferencia de archivos FTP (*File Transfer Protocol*).

Los documentos de Internet se pueden construir con el lenguaje HTML (*Hypertext Markup Language*). El desarrollo del HTML ha proporcionado un método simple de descripción de documentos que contienen texto, gráficos y referencias (*hyperlinks*; hipervínculos) a otros documentos. Aunque se pudo hacer que los documentos estuvieran disponibles mediante el empleo de HTML y el direccionamiento URL, y era difícil inicialmente encontrar información en Internet. Posteriormente, el centro National Center for Supercomputing Applications (NCSA) desarrolló un «navegador» llamado Mosaic que facilitó a los usuarios la búsqueda de recursos en Internet. Después el navegador Mosaic evolucionó para convertirse en el navegador Netscape.

El lenguaje HTML proporciona un método simple para desarrollar gráficos en Internet, pero dispone de capacidades limitadas. Por tanto, se han desarrollado otros lenguajes para aplicaciones gráficas para Internet. Estos lenguajes los estudiaremos en la Sección 2.8.

## 2.8 SOFTWARE GRÁFICO

---

Existen dos grandes grupos de software para gráficos por computadora: paquetes de propósito específico y paquetes de programación general. Los paquetes de propósito específico se diseñan para quienes no saben programar y quieran generar imágenes, gráficas o diagramas en algún área de aplicación sin preocuparse por los procedimientos gráficos necesarios para producir las imágenes. La interfaz de un paquete de propósito específico es habitualmente un conjunto de menús, permite a los usuarios comunicarse con los programas con sus propios términos. Como ejemplos de tales aplicaciones se pueden citar los programas de pintura para artistas y varios sistemas CAD (Computer Aided Design; diseño asistido por computador) para arquitectura, empresas, médicos e ingeniería. En cambio, un paquete de programación general proporciona una biblioteca de funciones gráficas que se pueden utilizar en lenguajes de programación tales como C, C++, Java o Fortran. Entre las funciones básicas de una biblioteca gráfica típica se incluyen aquéllas para especificar componentes de la imagen (líneas rectas, polígonos, esferas y otros objetos), establecer el color, seleccionar vistas de una escena y aplicar rotaciones u otras transformaciones. Algunos ejemplos de paquetes de programación gráfica general son los siguientes: GL (Graphics Library; biblioteca gráfica), OpenGL, VRML (Virtual-Reality Modelling Language; lenguaje para el modelado de realidad virtual), Java 2D y Java 3D. Un conjunto de funciones gráficas se denomina a menudo interfaz de programación de aplicaciones para gráficos por computadora (**computer-graphics application programming interface; CG API**), porque la biblioteca proporciona una interfaz software entre un lenguaje de programación (tal como C++) y el hardware. Por tanto, cuando escribimos un programa de aplicación en C++, las subrutinas gráficas permiten construir y mostrar una imagen en un dispositivo de salida.

### Representaciones con coordenadas

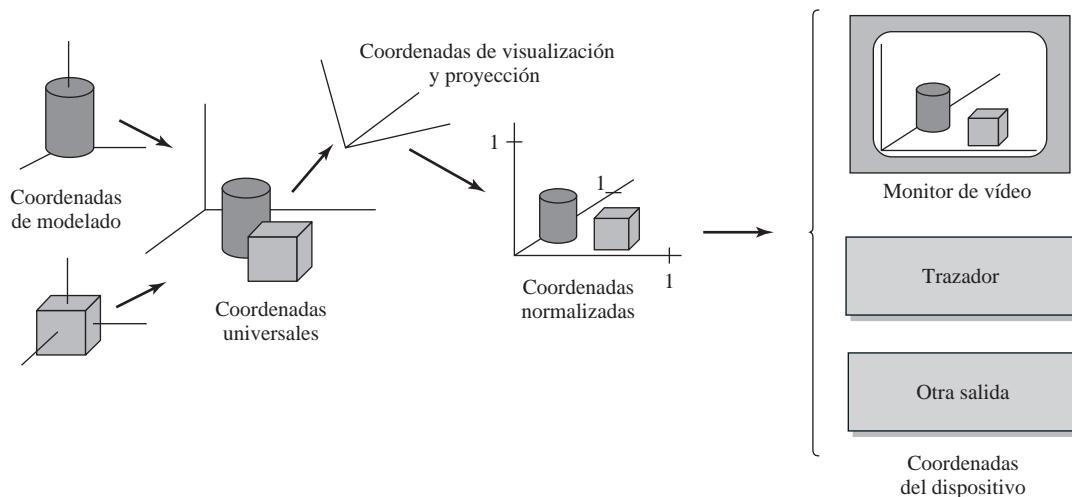
Para generar una imagen empleando un paquete de programación, lo primero que necesitamos es proporcionar las descripciones geométricas de los objetos que se han de mostrar. Estas descripciones determinan las posiciones y las formas de los objetos. Por ejemplo, una caja se especifica mediante las posiciones de sus esquinas (vértices), y una esfera se define con la posición de su centro y su radio. Con pocas excepciones, los paquetes para gráficos generales requieren que las descripciones geométricas se especifiquen en un sistema de referencia de coordenadas cartesianas estándar (Apéndice A). Si los valores de las coordenadas de una imagen se proporcionan en cualquier otro sistema de referencia (esférico, hiperbólico, etc.), se deben convertir en

coordenadas cartesianas antes de que se puedan utilizar como entrada del paquete de gráficos. Algunos paquetes diseñados para aplicaciones específicas pueden permitir el uso de otros sistemas de coordenadas que son apropiados para tales aplicaciones.

Por lo general, se utilizan varios sistemas de referencia cartesianos en la construcción y representación de una escena. En primer lugar, podemos definir las formas de los objetos individuales, tales como árboles o mobiliario, con un sistema de referencia de coordenadas independiente para cada objeto. Las coordenadas en estos sistemas de referencia se denominan **coordenadas de modelado** o, a veces, **coordenadas locales** o **coordenadas maestras**. Una vez que se ha especificado la forma individual de los objetos, podemos construir («modelar») una escena mediante la colocación de los objetos en sus posiciones apropiadas en un sistema de referencia de **coordenadas universales**. Este paso implica la transformación de las coordenadas en los sistemas de coordenadas de modelado individuales en posiciones y orientaciones especificadas en las coordenadas del sistema de coordenadas universales. A modo de ejemplo, podríamos construir una bicicleta mediante la definición de cada una de sus partes (ruedas, cuadro, asiento, manillar, ruedas dentadas, cadena, pedales) con un sistema de coordenadas de modelado independiente. Después, estas partes se unen en el sistema de coordenadas universales. Si ambas ruedas son del mismo tamaño, sólo necesitamos describir una de ellas en el sistema de coordenadas local. A continuación, la descripción de la rueda se introduce en la descripción de la bicicleta en coordenadas universales en dos lugares. En escenas no demasiado complicadas, los objetos componentes se pueden definir directamente en la estructura del objeto en coordenadas universales, sin realizar los pasos del sistema de coordenadas de modelado y la transformación del modelado. Las descripciones geométricas en el sistema de coordenadas de modelado y en el sistema de coordenadas universales se pueden expresar con valores cualesquiera en punto flotante o entero, sin tener en cuenta las restricciones de un dispositivo de salida concreto. En algunas escenas, podríamos querer especificar la geometría de los objetos en fracciones de pie, mientras que en otras aplicaciones podríamos querer emplear milímetros, kilómetros, o años luz.

Después de haber especificado todas las partes de una escena, la descripción del conjunto en coordenadas universales se procesa mediante varias subrutinas para su visualización en los sistemas de referencia de uno o más dispositivos de salida. Este proceso se denomina **pipeline de visualización**. En primer lugar, las posiciones en coordenadas universales se convierten a las *coordenadas de visualización* correspondientes a la vista que deseamos de la escena, que utilizan como base la posición y la orientación de una hipotética cámara. A continuación, las posiciones de objeto se transforman según una proyección bidimensional de la escena que se corresponde con lo que veremos en el dispositivo de salida. Entonces, la escena se almacena en **coordenadas normalizadas**, en las que cada coordenada se encuentra dentro de un rango de valores comprendido entre  $-1$  y  $1$  o dentro del rango de valores desde  $0$  a  $1$ , dependiendo del sistema. Las coordenadas normalizadas también se denominan *coordenadas de dispositivo normalizadas*, ya que la utilización de esta representación hace que el paquete gráfico sea independiente del rango de coordenadas del dispositivo de salida. También necesitamos identificar las superficies visibles y eliminar las partes de la imagen fuera de los límites de la vista que queremos mostrar en el dispositivo de visualización. Finalmente, la imagen se explora para almacenarla en el búfer de refresco de un sistema de rasterización para su representación. Los sistemas de coordenadas de los dispositivos de representación se denominan generalmente **coordenadas de dispositivo** o **coordenadas de pantalla** en el caso de un monitor de vídeo. A menudo, tanto las coordenadas normalizadas como las coordenadas de pantalla se especifican en un sistema de referencia de coordenadas a izquierdas para que un incremento positivo de las distancias desde el plano  $xy$  (la pantalla o plano de visualización) se pueda interpretar como un alejamiento de la posición de visualización.

La Figura 2.60 muestra de forma resumida la secuencia de transformaciones de coordenadas desde las coordenadas de modelado a las coordenadas de dispositivo, para una pantalla que debe contener la vista de los dos objetos tridimensionales. En esta figura, una posición inicial en coordenadas de modelado ( $x_{mc}$ ,  $y_{mc}$ ,  $z_{mc}$ ) se transforma en coordenadas universales, después en coordenadas de visualización y proyección, luego en coordenadas normalizadas a izquierdas y, finalmente, en la posición en coordenadas del dispositivo mediante la secuencia:



**FIGURA 2.60.** Secuencia de transformaciones desde las coordenadas de modelado hasta las coordenadas del dispositivo para una escena tridimensional. Las formas de los objetos se pueden definir individualmente en los sistemas de referencia de coordenadas de modelado. Después, las formas se sitúan dentro de la escena en coordenadas universales. A continuación, las especificaciones en coordenadas universales se transforman mediante la *pipeline* de visualización en coordenadas de visualización y proyección, y después en coordenadas normalizadas. Como último paso, los controladores de dispositivo individuales transfieren la representación de la escena en coordenadas normalizadas a los dispositivos de salida para su representación.

$$\begin{aligned}
 (x_{mc}, y_{mc}, z_{mc}) &\rightarrow (x_{wc}, y_{wc}, z_{wc}) \rightarrow (x_{vc}, y_{vc}, z_{vc}) \rightarrow (x_{pc}, y_{pc}, z_{pc}) \\
 &\rightarrow (x_{nc}, y_{nc}, z_{nc}) \rightarrow (x_{dc}, y_{dc})
 \end{aligned}$$

Las coordenadas de dispositivo  $(x_{dc}, y_{dc})$  son enteros dentro del rango  $(0,0)$  a  $(x_{\max}, y_{\max})$  para un dispositivo concreto. Además de las posiciones bidimensionales  $(x_{dc}, y_{dc})$  de la superficie de visualización, la información de profundidad para cada posición en coordenadas de dispositivo se almacena para su uso en varios algoritmos de visibilidad y procesamiento de superficies.

## Funciones gráficas

Un paquete gráfico de propósito general proporciona a los usuarios una gran variedad de funciones para la creación y manipulación de imágenes. Estas subrutinas se pueden clasificar según multitud de criterios, tales como estar relacionadas con la salida de gráficos, la entrada, los atributos, las transformaciones, visualización, subdivisión de imágenes, o control general.

Las partes básicas para la construcción de imágenes lo constituyen lo que se denomina **primitivas de salida gráfica**. Dentro de ellas están incluidas las cadenas de caracteres y las entidades geométricas, tales como los puntos, las líneas rectas, las líneas curvas, los rellenos con color (habitualmente polígonos), y las formas que se definen mediante matrices de puntos con color. Algunos paquetes gráficos proporcionan adicionalmente funciones para mostrar formas más complejas tales como esferas, conos y cilindros. Las subrutinas para la generación de primitivas de salida proporcionan las herramientas básicas para la construcción de imágenes.

Los **atributos** son propiedades de las primitivas de salida; es decir, un atributo describe cómo se mostrará una primitiva concreta. Dentro de éstos se incluyen las especificaciones de color, de estilos de línea, de estilos de texto y de patrones de relleno de áreas.

Podemos cambiar el tamaño, la posición, o la orientación de un objeto dentro de una escena empleando las **transformaciones geométricas**. Algunos paquetes gráficos proporcionan un conjunto adicional de funciones para realizar **transformaciones de modelado**, las cuales se utilizan para construir una escena en la que

las descripciones de los objetos individuales se expresan en coordenadas locales. Habitualmente, dichos paquetes proporcionan un mecanismo para la descripción de objetos complejos (tales como un circuito eléctrico o una bicicleta) con una estructura de árbol (jerárquica). Otros paquetes simplemente proporcionan las subrutinas de transformación geométrica y dejan los detalles de modelado al programador.

Después de que una escena se ha construido, empleando las subrutinas para la especificación de las formas de los objetos y de sus atributos, un paquete gráfico proyecta una vista de la imagen en un dispositivo de salida. Las **transformaciones de visualización** se utilizan para seleccionar una vista de la escena, la clase de proyección que se va a utilizar y la posición en un monitor de vídeo donde la vista se debe mostrar. Hay disponibles otras subrutinas para la gestión del área de visualización de pantalla mediante la especificación de su posición, tamaño y estructura. En escenas tridimensionales, los objetos visibles se identifican y se aplican las condiciones de iluminación.

Las aplicaciones gráficas interactivas utilizan varias clases de dispositivos de entrada, entre los que se incluyen un ratón, una tableta o un *joystick*. Las **funciones de entrada** se utilizan para controlar y procesar el flujo de datos procedente de estos dispositivos interactivos.

Algunos paquetes gráficos también proporcionan subrutinas para la subdivisión de la descripción de una imagen en un conjunto de partes componentes con un nombre. Se puede disponer de otras subrutinas para la manipulación de varias formas de estas componentes de la imagen.

Finalmente, un paquete gráfico contiene unas tareas de mantenimiento, tales como borrar una zona de la pantalla con un color seleccionado e inicializar parámetros. Podemos agrupar las funciones para llevar a cabo estas tareas de apoyo con el nombre de **operaciones de control**.

## Estándares de software

El objetivo principal del software gráfico estandarizado es la portabilidad. Cuando los paquetes se diseñan con funciones gráficas estándar, el software se puede trasladar fácilmente de un sistema hardware a otro y usar en diferentes implementaciones y aplicaciones. Sin estándares, los programas diseñados para un sistema hardware a menudo no se pueden trasladar a otro sistema sin realizar una amplia reescritura de los programas.

En muchos países las organizaciones nacionales e internacionales de planificación de estándares han cooperado en un esfuerzo para desarrollar un estándar de gráficos por computadora que sea aceptado de forma generalizada. Después de un considerable esfuerzo, este trabajo sobre los estándares condujo al desarrollo del sistema **GKS (Graphical Kernel System)** en 1984. Este sistema se adoptó como el primer estándar de software gráfico por la organización ISO (International Standards Organization) y por varias organizaciones nacionales de estándares, entre las que se incluye el instituto ANSI (American National Standards Institute). Aunque GKS se diseñó originalmente como un paquete de gráficos bidimensionales, pronto se desarrolló una extensión tridimensional de GKS. El segundo estándar de software que se desarrolló y aprobó por las organizaciones de estándares fue el sistema **PHIGS (Programmer's Hierarchical Interactive Graphics Standard)**, el cual es una extensión de GKS. PHIGS proporciona mayores capacidades para el modelado de objetos jerárquico, especificaciones de color, representación de superficies y manipulación de imágenes. Más tarde, se desarrolló una extensión de PHIGS, llamada PHIGS+, para proporcionar capacidades de representación de superficies tridimensionales no disponibles en PHIGS.

Cuando se desarrollaron los paquetes GKS y PHIGS, las estaciones de trabajo gráficas de Silicon Graphics, Inc. (SGI) llegaron a ser cada vez más populares. Estas estaciones de trabajo estaban dotadas de un conjunto de rutinas llamado **GL (Graphics Library)**, que muy pronto llegó a ser un paquete ampliamente utilizado en la comunidad gráfica. Por tanto, GL se convirtió en un estándar gráfico de facto. Las subrutinas de GL se diseñaron para la representación rápida y en tiempo real, y pronto este paquete se fue extendiendo a otros sistemas hardware. Como consecuencia, a comienzos de los años 90 se desarrolló OpenGL, una versión de GL independiente del hardware. En la actualidad, la organización **OpenGL Architecture Review Board** mantiene y actualiza este paquete. Esta organización es un consorcio de representantes de muchas compañías y organizaciones gráficas. OpenGL está diseñado especialmente para el procesamiento deficiente

de aplicaciones tridimensionales, pero también puede manipular descripciones de escenas bidimensionales como un caso particular del caso tridimensional donde todas las coordenadas  $z$  son 0.

Las funciones gráficas de cualquier paquete se definen habitualmente como un conjunto de especificaciones que son independientes de cualquier lenguaje de programación. Después se define una **correspondencia de lenguaje** a un lenguaje de programación de alto nivel particular. Esta correspondencia proporciona la sintaxis para acceder a las distintas funciones gráficas desde dicho lenguaje. Cada correspondencia de lenguaje se define para hacer el mejor uso de las capacidades correspondientes del lenguaje y para manejar diversos elementos sintácticos, tales como tipos de datos, paso de parámetros y errores. La organización internacional de estándares ISO establece las especificaciones de implementación de un paquete gráfico en un lenguaje determinado. Las correspondencias de OpenGL para los lenguajes C y C++ son iguales. También hay disponibles otras enlaces de OpenGL, tales como las de Ada y Fortran.

En los capítulos siguientes, utilizaremos el enlace C/C++ para OpenGL como marco de trabajo para el estudio de los conceptos gráficos básicos y el diseño y aplicación de los paquetes gráficos. Ejemplos de programas en C++ ilustran las aplicaciones de OpenGL y los algoritmos generales para la implementación de funciones gráficas.

## Otros paquetes gráficos

Se han desarrollado otras muchas bibliotecas de programación para gráficos por computadora. Algunas proporcionan subrutinas gráficas generales, y otras tienen como objetivo aplicaciones específicas o aspectos particulares de los gráficos por computadora, tales como la animación, la realidad virtual o los gráficos en Internet.

El paquete *Open Inventor* proporciona un conjunto de subrutinas orientadas a objetos para describir una escena que se debe mostrar mediante llamadas a OpenGL. El lenguaje VRML (*Virtual-Reality Modeling Language*), que comenzó como un subconjunto de Open Inventor, permite establecer modelos tridimensionales de mundos virtuales en Internet. También podemos construir imágenes en Internet utilizando bibliotecas gráficas desarrolladas para el lenguaje Java. Con *Java 2D*, podemos crear escenas bidimensionales dentro de applets de Java, por ejemplo. Mientras que con *Java 3D* podemos generar imágenes tridimensionales en Internet. Con *Renderman Interface* de Pixar Corporation, podemos generar escenas empleando una gran variedad de modelos de iluminación. Finalmente, las bibliotecas gráficas se proporcionan a menudo en otros tipos de sistemas, tales como Mathematica, MatLab y Maple.

## 2.9 INTRODUCCIÓN A OpenGL

---

En OpenGL se proporciona una biblioteca básica de funciones para especificar primitivas gráficas, atributos, transformaciones geométricas, transformaciones de visualización y muchas otras operaciones. Como indicamos en la sección anterior, está diseñada para ser independiente del hardware, por tanto, muchas operaciones, tales como las subrutinas de entrada y salida, no están incluidas en la biblioteca básica. Sin embargo, las subrutinas de entrada y salida y muchas funciones adicionales están disponibles en bibliotecas auxiliares que se han desarrollado para programas OpenGL.

### Sintaxis básica de OpenGL

Los nombres de las funciones de la **biblioteca básica de OpenGL** (también llamada de **biblioteca del núcleo de OpenGL**) utilizan como prefijo `gl`, y cada palabra que forma parte del nombre de una función tiene su primera letra en mayúscula. Los siguientes ejemplos ilustran este convenio de denominación.

```
glBegin,    glClear,    glCopyPixels,    glPolygonMode
```

Algunas funciones requieren que a uno (o más) de sus argumentos se les asigne una constante simbólica al especificar, por ejemplo, un nombre de parámetro, un valor para un parámetro, o un modo particular. Todas

estas constantes comienzan con las letras GL en mayúsculas. Además, las palabras que forman parte de una constante con nombres se escriben en mayúsculas, y el guión bajo (\_) se utiliza como separador entre todas estas palabras del nombre. Los siguientes son unos pocos ejemplos de los varios cientos de constantes simbólicas disponibles para uso con las funciones de OpenGL.

```
GL_2D, GL_RGB, GL_CCW, GL_POLYGON, GL_AMBIENT_AND_DIFFUSE
```

Las funciones de OpenGL también esperan tipos de datos específicos. Por ejemplo, un parámetro de una función de OpenGL podría esperar un valor que se especifica como un entero de 32 bits. Pero el tamaño de la especificación de un entero puede ser diferente en las distintas máquinas. OpenGL tiene incorporada una serie de nombres para tipos de datos, para indicar tipos de datos específicos tales como

```
GLbyte, GLshort, GLint, GLfloat, GLdouble, GLboolean
```

Cada nombre de tipo de datos comienza con las letras mayúsculas GL y, a continuación, un identificador de tipo de datos estándar, escrito con letras minúsculas.

A algunos argumentos de funciones de OpenGL se les puede asignar valores empleando una matriz que enumera un conjunto de valores de datos. Esta opción se utiliza para especificar una lista de valores como un puntero a una matriz, en lugar de especificar cada elemento de la lista explícitamente como un argumento. Un ejemplo típico del uso de esta opción es la especificación de los valores de las coordenadas xyz.

## Bibliotecas relacionadas

Existe un gran número de bibliotecas relacionadas para la realización de operaciones especiales, además de la biblioteca básica de OpenGL. La utilidad **GLU (OpenGL Utility)** proporciona subrutinas para la configuración de las matrices de visualización y proyección, descripción de objetos complejos mediante líneas y aproximaciones poligonales, visualización de cuádricas y *splines* B empleando aproximaciones lineales, procesamiento de operaciones de representación de superficies y otras tareas complejas. Toda implementación de OpenGL incluye la biblioteca GLU. Todos los nombres de las funciones de GLU comienzan con el prefijo glu. También existe un conjunto de herramientas orientadas a objetos basado en OpenGL, llamado **Open Inventor** que proporciona subrutinas y formas de objetos predefinidos para su uso en aplicaciones tridimensionales interactivas. Este conjunto de herramientas está escrito en C++.

Para crear gráficos utilizando OpenGL, necesitamos en primer lugar configurar una **ventana de visualización** en nuestra pantalla de vídeo. Se trata simplemente de la zona rectangular de la pantalla en la que nuestra imagen se mostrará. No podemos crear directamente la ventana de visualización con las funciones de OpenGL básicas, ya que esta biblioteca contiene únicamente funciones gráficas independientes del dispositivo, y las operaciones de gestión de ventanas dependen de la computadora que estemos utilizando. Sin embargo, existen varias bibliotecas de sistema de ventanas que soportan las funciones de OpenGL en una gran variedad de máquinas. La ampliación de OpenGL al sistema de ventanas X (**GLX**, OpenGL Extension to the X Window System) proporciona un conjunto de subrutinas que utilizan como prefijo las letras glx. Los sistemas Apple pueden utilizar la interfaz para operaciones de gestión de ventanas **Apple GL (AGL)**. Los nombres de las funciones en esta biblioteca utilizan como prefijo agl. En los sistemas que utilizan Microsoft Windows, las subrutinas de **WGL** proporcionan una interfaz de **Windows a OpenGL**. Estas subrutinas utilizan como prefijo las letras wgl. El gestor **PGL (Presentation Manager to OpenGL)** es una interfaz para el sistema operativo OS/2 de IBM, que utiliza el prefijo pg1 en las subrutinas de la biblioteca. Y el kit de herramientas **GLUT (OpenGL Utility Toolkit)** proporciona una biblioteca de funciones para interactuar con cualquier sistema de ventanas. Las funciones de la biblioteca GLUT utilizan como prefijo glut. Esta biblioteca también contiene métodos para describir y representar superficies y curvas cuádricas.

Ya que GLUT es una interfaz con otros sistemas de ventanas dependientes del dispositivo, podemos utilizar GLUT para que nuestros programas sean independientes del dispositivo. La información relacionada con la última versión de GLUT y los procedimientos de descarga de su código fuente están disponibles en la dirección web:

<http://reality.sgi.com/opengl/glut3/glut3.html>

## Archivos de cabecera

En todos nuestros programas gráficos, necesitaremos incluir el archivo de cabecera para la biblioteca de núcleo OpenGL. En la mayoría de las aplicaciones también necesitaremos GLU. Y necesitaremos incluir el archivo de cabecera para el sistema de ventanas. Por ejemplo, en Microsoft Windows, el archivo de cabecera para acceder a las subrutinas de WGL es `windows.h`. Este archivo de cabecera se debe indicar antes de los archivos de cabecera de OpenGL y GLU, ya que contiene macros que se necesitan en la versión de la biblioteca de OpenGL para Microsoft Windows. Por tanto, el archivo fuente en este caso debería comenzar con

```
#include <windows.h>
#include <GL/gl.h>
#include <GL/glu.h>
```

Sin embargo, si utilizamos GLUT para gestionar las operaciones de gestión de ventanas, no necesitaremos incluir `gl.h` y `glu.h` porque GLUT garantiza que estos archivos se incluirán correctamente. Por tanto, podemos reemplazar los archivos de cabecera de OpenGL y GLU por

```
#include <GL/glut.h>
```

También podríamos incluir `gl.h` y `glu.h`, pero al hacerlo seríamos redundantes y se podría ver afectada la portabilidad del programa.

Además, a menudo necesitaremos incluir archivos de cabecera que el código C++ requiere. Por ejemplo,

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
```

Con el nuevo estándar ISO/ANSI para C++, estos archivos de cabecera se denominan `cstudio`, `cstdlib` y `cmath`.

## Gestión de la ventana de visualización empleando GLUT

Para comenzar, podemos considerar un número mínimo y simplificado de operaciones para mostrar una imagen. Ya que estamos empleando la utilidad GLUT (*OpenGL Utility Toolkit*), nuestro primer paso consiste en inicializar GLUT. Esta función de inicialización podría también procesar argumentos cualesquiera de la línea de comandos, pero no necesitaremos utilizar estos parámetros en nuestro primer ejemplo de programa. Realizamos la inicialización de GLUT con la siguiente línea

```
glutInit (&argc, argv);
```

A continuación, podemos indicar que se cree una ventana de visualización en la pantalla con un título en su barra de título. Esto se realiza con la función

```
glutCreateWindow ("Un programa de ejemplo con OpenGL");
```

donde el único argumento de esta función puede ser cualquier cadena de caracteres que queramos utilizar como título de la ventana de visualización.

Ahora hay que especificar qué va a contener la ventana de visualización. Para ello, creamos una imagen empleando las funciones de OpenGL y se pasa la definición de la imagen a la subrutina llamada `glutDisplayFunc`, que asigna nuestra imagen a la ventana de visualización. Como ejemplo, suponga que disponemos del código de OpenGL para describir un segmento en un procedimiento llamado `lineSegment`. Entonces la siguiente llamada a función pasa la descripción del segmento a la ventana de visualización.

```
glutDisplayFunc (lineSegment);
```

Pero la ventana de visualización no está aún en la pantalla. Necesita una función más de GLUT para completar las operaciones de procesamiento de ventana. Después de la ejecución de la siguiente línea, todas las ventanas de visualización que hayamos creado, incluyendo su contenido gráfico, se activarán.

```
glutMainLoop ( );
```

Esta función debe ser la última en nuestro programa. Ésta muestra los gráficos iniciales y pone el programa en un bucle infinito que comprueba la entrada procedente de dispositivos, tales como un ratón o un teclado. Nuestro primer ejemplo no será interactivo, por lo que el programa únicamente continuará mostrando la imagen hasta que cerremos la ventana de visualización. En los capítulos siguientes, tendremos en consideración cómo podemos modificar nuestros programas en OpenGL para gestionar la entrada interactiva.

Aunque la ventana de visualización que creamos tendrá una posición y un tamaño predeterminados, podemos establecer estos parámetros empleando funciones adicionales del kit de herramientas GLUT. Utilizamos la función `glutInitWindowPosition` para proporcionar una posición inicial para la esquina superior izquierda de la ventana de visualización. Esta posición se especifica en coordenadas enteras de pantalla, cuyo origen está en la esquina superior izquierda de la pantalla. Por ejemplo, la siguiente línea especifica que la esquina superior izquierda de la ventana de visualización se debería colocar 50 píxeles a la derecha del borde izquierdo de la pantalla y 100 píxeles hacia abajo desde el borde superior de la pantalla.

```
glutInitWindowPosition (50, 100);
```

Análogamente, la función `glutInitWindowSize` se utiliza para establecer la anchura y la altura en píxeles de la ventana de visualización. Por tanto, especificamos una ventana de visualización con una anchura inicial de 400 píxeles y una altura de 300 píxeles (Figura 2.61) con la línea siguiente

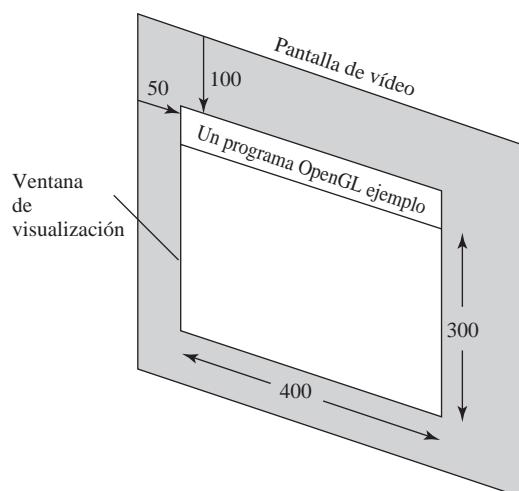
```
glutInitWindowSize (400, 300);
```

Después de que la ventana de visualización esté en la pantalla, podemos volver a cambiar tanto su posición como su tamaño.

También podemos establecer otras opciones de la ventana de visualización, tales como los búferes y una opción de los modos de color, con la función `glutInitDisplayMode`. Los argumentos de ésta subrutina se asignan mediante constante simbólicas de GLUT. Por ejemplo, la siguiente orden especifica que se utilice un único búfer de refresco en la ventana de visualización y que se utilice el modo de color RGB (rojo, verde, azul) para seleccionar los valores de los colores.

```
glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
```

Los valores de las constantes que se pasan a esta función se combinan empleando la operación lógica *or*. En realidad, el búfer único y el modo de color RGB son opciones predeterminadas. Pero usaremos esta fun-



**FIGURA 2.61.** Una ventana de visualización de 400 por 300 píxeles en la posición (50,100) relativa a la esquina superior izquierda de la pantalla de vídeo.

ción para recordar las opciones que se han establecido para nuestra visualización. Más tarde, estudiaremos los modos de color más detalladamente, así como otras opciones de visualización tales como el doble búfer para aplicaciones de animación y selección de parámetros para la visualización de escenas tridimensionales.

## Un programa OpenGL completo

Todavía hay que realizar algunas tareas antes de que tengamos todas las partes necesarias para un programa completo. En la ventana de visualización, podemos elegir un color de fondo. Y necesitamos construir un procedimiento que contenga las funciones apropiadas de OpenGL para la imagen que queremos mostrar en pantalla.

Mediante el empleo de valores de color RGB, establecemos que el color de la ventana de visualización sea blanco, como en la Figura 2.61, con la función de OpenGL

```
glClearColor (1.0, 1.0, 1.0, 0.0);
```

Los tres primeros argumentos de esta función establecen cada una de las componentes de color roja, verde, y azul en el valor de 1.0. Por tanto, obtenemos el color blanco en la ventana de visualización. Si, en lugar de 1.0, establecemos cada una de las componentes de color en 0.0, obtendríamos el color negro como color de fondo. Y si a cada una de las componentes roja, verde, y azul se les asigna un valor intermedio entre 0.0 y 1.0, obtendríamos algún nivel de gris. El cuarto parámetro de la función `glClearColor` se denomina el *valor alfa* del color especificado. Un uso del valor alfa es el parámetro de «fundido» (*blending*). Cuando activamos las operaciones de fundido de OpenGL, los valores alfa se pueden utilizar para calcular el color resultante de dos objetos que se superponen. Un valor alfa de 0.0 indica que el objeto es totalmente transparente, mientras que el valor alfa de 1.0 indica que el objeto es opaco. Las operaciones de fundido no se utilizarán por el momento, por lo que el valor alfa es irrelevante en nuestros primeros programas de ejemplo. Por ahora, establecemos simplemente el valor alfa en 0.0.

Aunque la orden `glClearColor` asigna un color a la ventana de visualización, ésta no muestra la ventana de visualización en la pantalla. Para conseguir que el color asignado a la ventana se visualice, necesitamos invocar la siguiente función de OpenGL.

```
glClear (GL_COLOR_BUFFER_BIT);
```

El argumento `GL_COLOR_BUFFER_BIT` es una constante simbólica que especifica que son los valores de los bits del búfer de color (búfer de refresco) los que se deben asignar a los valores indicados en la función `glClearColor`. (Estudiaremos otros búferes en los capítulos siguientes.)

Además de establecer el color de fondo de la ventana de visualización, podemos elegir entre una gran variedad de esquemas de color para los objetos que queremos mostrar en una escena. En nuestro ejemplo inicial de programación, estableceremos simplemente el color del objeto en rojo, y se aplazará el estudio en profundidad de las variadas opciones de color hasta el Capítulo 4:

```
	glColor3f (1.0, 0.0, 0.0);
```

El sufijo `3f` de la función `glColor` indica que especificamos las tres componentes de color RGB mediante el empleo de valores en punto flotante (`f`). Estos valores se deben encontrar dentro del rango comprendido entre 0.0 y 1.0, y hemos establecido la componente roja en 1.0 y las componentes verde y azul en 0.0.

En nuestro primer programa, simplemente mostramos un segmento bidimensional. Para ello, necesitamos decir a OpenGL como queremos «proyectar» nuestra imagen en la ventana de visualización, porque la generación de una imagen bidimensional se trata en OpenGL como un caso especial de la visualización tridimensional. Por lo que, aunque sólo queramos producir una línea muy simple bidimensional, OpenGL procesa la imagen mediante todas las operaciones de visualización tridimensional. Podemos establecer que el tipo de proyección (modo) y otros parámetros de visualización que necesitemos con las dos funciones siguientes.

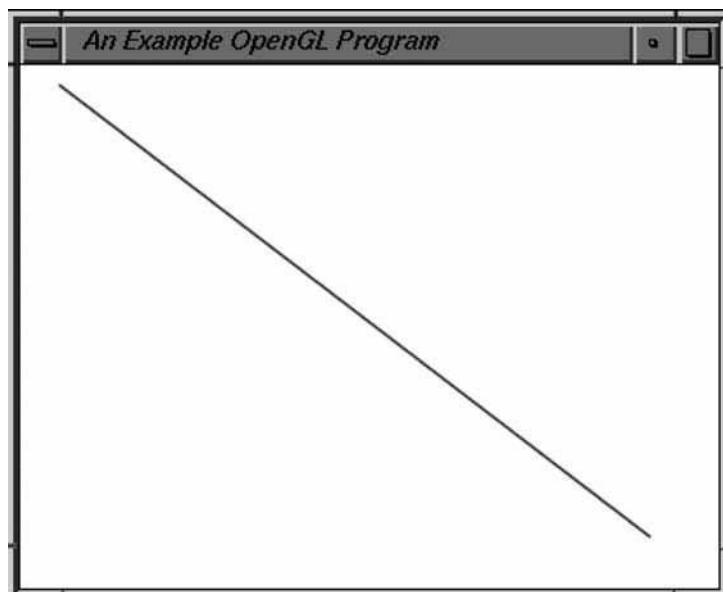
```
glMatrixMode (GL_PROJECTION);
gluOrtho2D (0.0, 200.0, 0.0, 150.0);
```

Esto especifica que se debe utilizar una proyección ortogonal para mapear los contenidos de una zona rectangular bidimensional (2D) de las coordenadas universales a la pantalla, y que los valores de la coordenada  $x$  dentro de este rectángulo varían desde 0,0 hasta 200,0 y que los valores de la coordenada  $y$  varían desde 0,0 hasta 150,0. Objetos cualesquiera que definamos dentro de este rectángulo de coordenadas universales se mostrarán dentro de la pantalla de visualización. Cualquier cosa fuera de este rango de coordenadas no se visualizará. Por tanto, la función de GLU `gluOrtho2D` establece que el sistema de coordenadas de referencia dentro de la ventana de visualización deberá tener las coordenadas (0,0, 0,0) en la esquina inferior izquierda de la ventana de visualización y (200,0, 150,0) en la esquina superior izquierda de la ventana. Ya que sólo describimos un objeto bidimensional, el único efecto que tiene la proyección ortogonal es «pegar» nuestra imagen en la ventana de visualización definida anteriormente. Por ahora, utilizaremos un rectángulo de coordenadas universales con la misma relación de aspecto que la ventana de visualización, para que no haya distorsión en nuestra imagen. Posteriormente, consideraremos cómo podemos mantener una relación de aspecto que sea independiente de la especificación de la ventana de visualización.

Finalmente, necesitamos llamar a las subrutinas apropiadas de OpenGL para crear nuestro segmento. El código siguiente define un segmento bidimensional definiendo sus extremos con coordenadas cartesianas enteras de valores (180, 15) y (10, 145). En el Capítulo 3, presentaremos una explicación detallada de estas funciones y de otras funciones de OpenGL para la generación de primitivas gráficas.

```
glBegin (GL_LINES);
    glVertex2i (180, 15);
    glVertex2i (10, 145);
glEnd ( );
```

Ahora es el momento de reunir todas las piezas. El siguiente programa de OpenGL está organizado en tres procedimientos. Colocamos todas las inicializaciones y los parámetros de configuración relacionados en el procedimiento `init`. Nuestra descripción geométrica de la «imagen» que queremos visualizar está en el procedimiento `lineSegment`, que es el procedimiento que será referenciado por la función de GLUT `glutDisplayFunc`. Y el procedimiento `main` contiene las funciones de GLUT que configuran la ventana de visualización y que muestran nuestro segmento en la pantalla. La Figura 2.62 muestra la ventana de visualización y el segmento rojo (gris en la figura) generado por este programa.



**FIGURA 2.62.** La ventana de visualización y el segmento generados por el programa de ejemplo.

```

#include <GL/glut.h>      // (u otras líneas, dependiendo del sistema que usemos

void init (void)
{
    glClearColor (1.0, 1.0, 1.0, 0.0); // Establece el color de la ventana de
                                         // visualización en blanco.

    glMatrixMode (GL_PROJECTION);        // Establece los parámetros de proyección.
    gluOrtho2D (0.0, 200.0, 0.0, 150.0);
}

void lineSegment (void)
{
    glClear (GL_COLOR_BUFFER_BIT);      // Borra la ventana de visualización.

    glColor3f (1.0, 0.0, 0.0);         // Establece el color del segmento de
                                         // línea en rojo.

    glBegin (GL_LINES);
        glVertex2i (180, 15); // Especifica la geometría del segmento de línea.
        glVertex2i (10, 145);
    glEnd ( );

    glFlush ( );                     // Procesa todas las subrutinas de OpenGL tan rápidamente
                                         // como sea posible.
}

void main (int argc, char** argv)
{
    glutInit (&argc, argv);           // Inicializa GLUT.
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB); // Establece el modo de
                                                 // visualización.

    glutInitWindowPosition (50, 100);   // Establece la posición de la esquina
                                         // superior izquierda de la ventana de visualización.
    glutInitWindowPosition (50, 100);   // Establece la posición de la esquina
                                         // superior izquierda de la ventana de visualización.
    glutInitWindowSize (400, 300);     // Establece el ancho y la altura de la
                                         // ventana de visualización.

    glutCreateWindow («An Example OpenGL Program»); // Crea la ventana de
                                                 // visualización.

    init ( );                        // Ejecuta el procedimiento de inicialización.
    glutDisplayFunc (lineSegment);   // Envía los gráficos a la ventana de
                                         // visualización.

    glutMainLoop ( );               // Muestra todo y espera.
}

```

Al final del procedimiento `lineSegment` hay una función, `glflush`, que todavía no hemos estudiado. Es simplemente una subrutina que fuerza la ejecución de nuestras funciones de OpenGL, las cuales almacenan las computadoras en búferes en diferentes posiciones, dependiendo de cómo esté implementada OpenGL. En una red ocupada, por ejemplo, podría haber retrasos en el procesamiento de algunos búferes. Pero la llamada a `glFlush` fuerza a que todos estos búferes se vacíen y que las funciones de OpenGL se procesen.

El procedimiento `lineSegment` que hemos creado para describir nuestra imagen se denomina *función de respuesta a la visualización (display callback function)*. Este procedimiento lo «registra» `glutDisplayFunc` como la subrutina que se invoca siempre que sea preciso mostrar la ventana de visualización de nuevo. Esto puede ocurrir, por ejemplo, si se mueve la ventana de visualización. En capítulos posteriores mostraremos otros tipos de funciones de respuesta y las subrutinas de GLUT asociadas que se emplean para registrar éstas. Por lo general, los programas que utilizan OpenGL se organizan como un conjunto de funciones de respuesta que se invocan cuando ocurren determinadas acciones.

## 2.10 RESUMEN

---

En este capítulo introductorio, hemos examinado las principales características del hardware y del software de los sistemas de gráficos por computadora. Entre los componentes hardware se incluyen los monitores de vídeo, los dispositivos de copia impresa, varias clases de dispositivos de entrada y componentes para interactuar con entornos virtuales. Algunos sistemas de software, tales como paquetes CAD y programas de dibujo se diseñan para aplicaciones específicas. Otros sistemas de software proporcionan una biblioteca de subrutinas gráficas generales que se puedan utilizar dentro de un lenguaje de programación, tal como C++ para generar imágenes para cualquier aplicación.

El dispositivo de visualización de gráficos predominante es el monitor de refresco de barrido, que se basa en la tecnología de la televisión. Un sistema de barrido utiliza un búfer de imagen para almacenar los valores del color de cada punto de la pantalla (píxel). Entonces las imágenes se pintan en la pantalla mediante la obtención de esta información a partir del búfer de imagen (también llamado búfer de refresco) a medida que el haz de electrones del TRC barre cada línea de exploración, desde arriba hacia abajo. Las pantallas vectoriales más antiguas construyen las imágenes mediante el dibujo de segmentos de línea recta entre los puntos extremos que se especifican.

Hay disponibles otros muchos dispositivos de visualización de vídeo. Concretamente, la tecnología de visualización de las pantallas planas se desarrolla a un ritmo rápido, y estos dispositivos ahora se utilizan en una gran variedad de sistemas, entre los que se incluyen tanto las computadoras de escritorio como las computadoras portátiles. Las pantallas de plasma y los dispositivos de cristal líquido son dos ejemplos de pantallas planas. Existen otras tecnologías de visualización entre las que se pueden incluir los sistemas de visualización tridimensional y estereoscópica. Los sistemas de realidad virtual pueden incluir un casco estereoscópico o un monitor de vídeo estándar.

Para la entrada gráfica, disponemos de una gran gama de dispositivos entre los que elegir. Los teclados, las cajas de botones, y los botones se utilizan para introducir texto, valores de datos o programar opciones. El dispositivo más popular para «apuntar» es el ratón, pero los *trackballs*, *spaceballs*, palancas de mando, teclas de control del cursor y las ruedas de pulgar también se utilizan para posicionar el cursor en la pantalla. En los entornos de realidad virtual, se utilizan habitualmente los guantes de datos. Otros dispositivos de entrada son los escáneres de imagen, los digitalizadores, las pantallas táctiles, los lapiceros ópticos y los sistemas de voz.

Entre los dispositivos de copia impresa para estaciones gráficas se incluyen las impresoras estándar y los trazadores, además de dispositivos para producir diapositivas, transparencias y películas. Las impresoras producen una copia impresa mediante el empleo de una matriz de puntos, un láser, inyección de tinta o métodos electrostáticos o electrotérmicos. Los gráficos y diagramas se pueden producir con un trazador de plumillas o con un dispositivo que sea una combinación de impresora y trazador.

Los paquetes estándar de programación de gráficos desarrollados y aprobados por ISO y ANSI son GKS 3D GKS, PHIGS y PHIGS+. Otros paquetes que han evolucionado hacia estándares son GL y OpenGL. Otras muchas bibliotecas gráficas están disponibles para su uso con lenguajes de programación, entre las que se incluyen Open Inventor, VRML, RenderMan, Java 2D y Java 3D. Otros sistemas, tales como Mathematica, MatLab y Maple, proporcionan a menudo un conjunto de funciones de programación de gráficos.

Normalmente, los paquetes de programación de gráficos requieren que las especificaciones de coordenadas se expresen en sistemas de referencia cartesianos. Cada objeto de una escena se puede definir en un sis-

tema de coordenadas cartesianas de modelado independiente, el cual se mapea entonces a su posición en coordenadas universales para construir la escena. A partir de las coordenadas universales, los objetos tridimensionales se proyectan sobre un plano bidimensional, convertido a coordenadas normalizadas de dispositivo, y entonces transformado a las coordenadas finales del dispositivo de visualización. Las transformaciones desde las coordenadas de modelado a las coordenadas normalizadas de dispositivo son independientes de los dispositivos de salida concretos que se podrían utilizar en una aplicación. Entonces los controladores de dispositivo se utilizan para convertir las coordenadas normalizadas en coordenadas de dispositivo enteras.

Las funciones disponibles en los paquetes de programación de gráficos se pueden clasificar en las siguientes categorías: primitivas de salida de gráficos, atributos, transformaciones de modelado y geométricas, transformaciones de visualización, funciones de entrada, operaciones de estructuración de imagen y operaciones de control.

El sistema OpenGL consta de un conjunto de subrutinas independientes del dispositivo (llamado la biblioteca de núcleo), la biblioteca de utilidades (GLU) y el conjunto de herramientas de utilidades (GLUT). En el conjunto auxiliar de subrutinas proporcionado por GLU, hay funciones disponibles para la generación de objetos complejos, para las especificaciones de parámetros en aplicaciones de visualización bidimensional, para realizar operaciones de representación de superficies y para realizar algunas otras tareas de soporte. En GLUT, tenemos un conjunto extenso de funciones para la gestión de ventanas de visualización, interactuar con los sistemas de ventanas y para la generación de algunas formas tridimensionales. Podemos utilizar GLUT para interactuar con cualquier computadora, o podemos utilizar GLX, Apple GL, WGL u otro paquete de software específico de un sistema.

## REFERENCIAS

---

Se dispone de información general acerca de las pantallas electrónicas en Tannas (1985) y en Sherr (1993). Los dispositivos de pantalla plana se estudian en Depp y Howard (1993). Información adicional acerca de la arquitectura de raterización de gráficos se puede encontrar en Foley, Van Dam, Feiner y Hughes (1990). Las pantallas tridimensionales y estereoscópicas se estudian en Jhonson (1982) y en Grotch (1983). Las pantallas que forman parte de cascos y los entornos de realidad virtual se estudian en Cheng y otros (1989).

Las fuentes estándar para información sobre OpenGL son Woo, Neider, Davis y Shreiner (1999) y Shreiner (2000). Open Inventor se explora en Wernecke (1994). McCarthy y Descartes (1998) se pueden consultar para estudios de VRML. Se puede encontrar una presentación de RenderMan en Upstill (1989). En Knudsen (1999), Hardy (2000), y Horstman y Cornell (2001) se muestran ejemplos de programación con Java 2D. La programación de gráficos empleando Java 3D se explora en Sowizral, Rushforth y Deering (2000); Palmer (2001); Selman (2002); y Walsh y Gehringer (2002).

Para información sobre PHIGS y PHIGS+, véase Howard, Hewitt, Hubbald y Wyrwas (1991); Hopgood y Duce (1991); Gaskins (1992); y Blake (1993). Información acerca del estándar GKS bidimensional y la evolución de los estándares gráficos se encuentra disponible en Hopgood, Duce, Gallop y Sutcliffe (1983). Enderle, Kansy y Pfaff (1984) es una cita bibliográfica adicional para GKS.

## EJERCICIOS

---

- 2.1 Enumere las características de funcionamiento de las siguientes tecnologías de pantalla: sistemas de barrido, sistemas de refresco vectorial, pantallas de plasma y pantallas de cristal líquido (LCD).
- 2.2 Enumere algunas aplicaciones apropiadas para cada tecnología de pantalla del Ejercicio 2.1.
- 2.3 Determine la resolución (píxeles por centímetro) en las direcciones  $x$  e  $y$  para el monitor de vídeo que se utiliza en su sistema. Determine la relación de aspecto y explique cómo se pueden mantener las proporciones relativas de los objetos en su sistema.

- 2.4 Considere tres sistemas diferentes de barrido con resoluciones de 640 por 480, 1280 por 1024 y 2560 por 2048. ¿Qué tamaño de búfer de imagen (en bytes) es necesario para que cada uno de estos sistemas almacene 12 bits por píxel? Qué capacidad de almacenamiento se requiere para cada sistema si se han de almacenar 24 bits por píxel?
- 2.5 Suponga un sistema de barrido RGB que debe ser diseñado empleando una pantalla de 8 pulgadas por 10 pulgadas con una resolución de 100 píxeles por pulgada en cada dirección. Si queremos almacenar 6 bits por píxel en el búfer de imagen, ¿qué capacidad de almacenamiento (en bytes) necesitaremos para el búfer de imagen?
- 2.6 ¿Cuánto tiempo será necesario para cargar un búfer de imagen de 640 por 480 de 12 bits por píxel, si se pueden transferir  $10^5$  bits por segundo? ¿Cuánto tiempo será necesario para cargar un búfer de imagen de 24 bits por píxel con una resolución de 1280 por 1024 empleando la misma tasa de transferencia?
- 2.7 Suponga que dispone de una computadora con palabras de 32 bits y una tasa de transferencia de 1 mip (un millón de instrucciones por segundo). ¿Cuánto tiempo será necesario para llenar el búfer de imagen de una impresora láser de 300 dpi (dot per inch; puntos por pulgada) con un tamaño de página de 8 1/2 pulgadas por 11 pulgadas?
- 2.8 Considere dos sistemas de barrido con resoluciones de 640 por 480 y 1280 por 2024. ¿A cuántos píxeles por segundo podría acceder en cada uno de estos sistemas un controlador de visualización que refresca la pantalla a una velocidad de 60 cuadros por segundo? ¿Cuál es el tiempo de acceso a cada píxel en cada sistema?
- 2.9 Suponga que disponemos de un monitor de vídeo con un área de visualización que mide 12 pulgadas de ancho por 9,6 pulgadas de altura. Si la resolución es de 1280 por 1024 y la relación de aspecto es 1, ¿cuál es el diámetro de cada punto de pantalla?
- 2.10 ¿Cuánto tiempo se necesitará para barrer cada fila de píxeles durante el refresco de la pantalla en un sistema de barrido con una resolución de 1280 por 1024 y una velocidad de refresco de 60 cuadros por segundo?
- 2.11 Considere un monitor de barrido no entrelazado con una resolución de  $n$  por  $m$  ( $m$  líneas de barrido y  $n$  píxeles por línea de barrido), una velocidad de refresco de  $r$  cuadros por segundo, un tiempo de retrazado horizontal de  $t_{horiz}$ , y un tiempo de retrazado vertical de  $t_{vert}$ . ¿Cuál es la fracción del tiempo total de refresco por cuadro que se consume en retrazar el haz de electrones?
- 2.12 ¿Cuál es la fracción del tiempo total de refresco por cuadro necesario para retrazar el haz de electrones en un sistema de barrido no entrelazado con una resolución de 1280 por 1024, una velocidad de refresco de 60 Hz, y un tiempo de retrazado horizontal de 5 microsegundos y un tiempo de retrazado vertical de 500 microsegundos?
- 2.13 Asumiendo que en cierto sistema de barrido RGB de color total (24 bits por píxel) tiene un búfer de imagen de 512 por 500, ¿cuántas elecciones de color distintas (niveles de intensidad) tendríamos disponibles? ¿Cuántos colores diferentes podríamos visualizar a la vez?
- 2.14 Compare las ventajas y las desventajas de un monitor tridimensional que utiliza un espejo varifocal frente a un sistema estereoscópico.
- 2.15 Enumere los diferentes componentes de entrada y salida que se utilizan habitualmente en sistemas de realidad virtual. Explique también cómo los usuarios interactúan con la escena virtual mostrada con diferentes dispositivos de salida, tales como los monitores bidimensionales y los monitores estereoscópicos.
- 2.16 Explique cómo los sistemas de realidad virtual se pueden utilizar en aplicaciones de diseño. ¿Qué otras aplicaciones tienen los sistemas de realidad virtual?
- 2.17 Enumere algunas aplicaciones de las pantallas panorámicas.
- 2.18 Explique las diferencias entre un sistema de gráficos general diseñado para un programador y uno diseñado para una aplicación específica, tal como el diseño en arquitectura.
- 2.19 Explique las diferencias entre la biblioteca de núcleo de OpenGL, la biblioteca de utilidades de OpenGL (GLU) y el kit de herramientas de utilidades de OpenGL (GLUT).
- 2.20 ¿Qué orden podríamos utilizar para establecer o cambiar el color de una ventana de visualización de OpenGL a gris claro? ¿Qué orden podríamos utilizar para cambiar el color de la ventana de visualización al color negro?
- 2.21 Enumere las líneas que se necesitan para configurar la ventana de visualización de OpenGL con su esquina inferior derecha en la posición en píxeles (200, 200), un ancho de ventana de 100 píxeles y una altura de 75 píxeles.
- 2.22 Explique qué significa el término «función de respuesta de visualización de OpenGL».

## CAPÍTULO 3

# Primitivas gráficas



Una escena del vídeo del hombre lobo. La figura animada de este primitivo licántropo está modelada con 61 huesos y ocho capas de piel. Cada imagen de la animación infográfica contiene 100.000 polígonos de superficie.

(Cortesía de NVIDIA Corporation.)

- |   |   |
|---|---|
| <p><b>3.1</b> Sistemas de coordenadas de referencia</p> <p><b>3.2</b> Especificación de un sistema bidimensional de referencia universal en OpenGL</p> <p><b>3.3</b> Funciones de punto en OpenGL</p> <p><b>3.4</b> Funciones OpenGL para líneas</p> <p><b>3.5</b> Algoritmos de dibujo de líneas</p> <p><b>3.6</b> Algoritmos paralelos de dibujo de líneas</p> <p><b>3.7</b> Almacenamiento de los valores en el búfer de imagen</p> <p><b>3.8</b> Funciones OpenGL para curvas</p> <p><b>3.9</b> Algoritmos para generación de círculos</p> <p><b>3.10</b> Algoritmos de generación de elipses</p> <p><b>3.11</b> Otras curvas</p> <p><b>3.12</b> Algoritmos paralelos para curvas</p> <p><b>3.13</b> Direccionamiento de píxeles y geometría de los objetos</p> | <p><b>3.14</b> Primitivas de áreas rellenas</p> <p><b>3.15</b> Áreas de relleno poligonales</p> <p><b>3.16</b> Funciones OpenGL para relleno de áreas poligonales</p> <p><b>3.17</b> Matrices de vértices OpenGL</p> <p><b>3.18</b> Primitivas de matrices de píxeles</p> <p><b>3.19</b> Funciones OpenGL para matrices de píxeles</p> <p><b>3.20</b> Primitivas de caracteres</p> <p><b>3.21</b> Funciones OpenGL de caracteres</p> <p><b>3.22</b> Particionamiento de imágenes</p> <p><b>3.23</b> Listas de visualización de OpenGL</p> <p><b>3.24</b> Funciones OpenGL de redimensionamiento de la ventana de visualización</p> <p><b>3.25</b> Resumen</p> |
|---|---|

Un paquete software de propósito general para aplicaciones gráficas, lo que algunas veces se denomina interfaz de programación de aplicaciones infográficas, proporciona una biblioteca de funciones que podemos utilizar dentro de un lenguaje de programación como C++ para crear imágenes. Como hemos indicado en la Sección 2.8, el conjunto de funciones de la biblioteca puede subdividirse en varias categorías. Una de las primeras cosas que necesitamos hacer al crear una imagen es describir las partes componentes de la escena que hay que mostrar. Los componentes de la imagen podrían ser los árboles y el terreno, muebles y paredes, escaparates y escenas callejeras, automóviles y carteles anunciantes, átomos y moléculas o estrellas y galaxias. Para cada tipo de escena, tenemos que describir la estructura de los objetos individuales y las coordenadas de su ubicación dentro de la escena. Las funciones de un paquete gráfico que se utilizan para describir los distintos componentes de la imagen se denominan **primitivas gráficas** o simplemente **primitivas**. Las primitivas gráficas que describen la geometría de los objetos se denominan normalmente **primitivas geométricas**. Las primitivas geométricas más simples son las que indican posiciones de puntos y segmentos de líneas rectas. Entre las primitivas geométricas adicionales que un paquete gráfico puede incluir están los círculos y otras secciones cónicas, las superficies cuádricas, las superficies y curvas de tipo *spline* y las áreas coloreadas poligonales. Y la mayoría de los sistemas gráficos proporcionan también funciones para mostrar cadenas de caracteres. Después de haber especificado la geometría de una imagen dentro de un sistema de coordenadas de referencia determinado, las primitivas gráficas se proyectan sobre un plano bidimensional, que se corresponde con el área de visualización de un dispositivo de salida, realizando la conversión línea a línea en una serie de posiciones de píxel enteras dentro del búfer de imagen.

En este capítulo, vamos a presentar las primitivas gráficas disponibles en OpenGL y también hablaremos de los algoritmos de nivel de dispositivo que se utilizan para implementar dichas primitivas. La exploración de los algoritmos de implementación de una biblioteca gráfica nos permitirá comprender mejor las capacidades de estos paquetes software. También nos dará una idea de cómo trabajan esas funciones, de cómo se las podría mejorar y de cómo podemos implementar nosotros mismos una serie de rutinas gráficas para algu-

na aplicación especial. Las investigaciones realizadas en el campo de la infografía están continuamente descubriendo técnicas de implementación nuevas o mejoradas para proporcionar métodos para aplicaciones especiales, como las de los gráficos por Internet, y para desarrollar mecanismos de visualización gráfica más rápidos y más realistas.

## 3.1 SISTEMAS DE COORDENADAS DE REFERENCIA

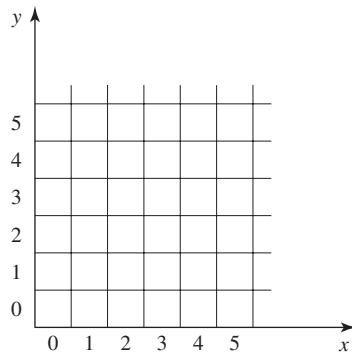
---

Para describir una imagen, primero es necesario seleccionar un sistema de coordenadas cartesianas adecuado, denominado sistema de coordenadas de referencia del mundo, que puede ser bidimensional o tridimensional. Después se describen los objetos de la imagen proporcionando sus especificaciones geométricas en términos de la posición dentro de las coordenadas del mundo. Por ejemplo, definimos un segmento de línea recta proporcionando la posición de los dos puntos extremos, mientras que un polígono se especifica proporcionando el conjunto de posiciones de sus vértices. Estas coordenadas se almacenan en la descripción de la escena, junto con otras informaciones acerca de los objetos, como por ejemplo su color y su **extensión de coordenadas**, que son los valores  $x$ ,  $y$  y  $z$  máximos y mínimos para cada objeto. Un conjunto de coordenadas de extensión también se denomina **recuadro de contorno** del objeto. Para una figura bidimensional, las coordenadas de extensión se denominan a veces **rectángulo de contorno**. A continuación, los objetos se visualizan pasando la información de la escena a las rutinas de visualización, que identifican las superficies visibles y asignan los objetos a sus correspondientes posiciones en el monitor de vídeo. El proceso de conversión de líneas almacena la información sobre la escena, como por ejemplo los valores de color, en las apropiadas ubicaciones dentro del búfer de imagen, y los objetos de la escena se muestran en el dispositivo de salida.

### Coordenadas de pantalla

Las ubicaciones sobre un monitor de vídeo se describen mediante **coordenadas de pantalla** que son números enteros y que se corresponden con las posiciones de píxel dentro del búfer de imagen. Los valores de las coordenadas de píxel proporcionan el *número de línea de exploración* (el valor  $y$ ) y el *número de columna* (el valor  $x$ ) dentro de una línea de exploración. Los procesos hardware, como el de refresco de pantalla, normalmente direccionan las posiciones de píxel con respecto al extremo superior izquierdo de la pantalla. Las líneas de exploración se identifican por tanto comenzando por 0, en la parte superior de la pantalla, y continuando hasta un cierto valor entero,  $y_{\max}$ , en la parte inferior de la pantalla, mientras que las posiciones de píxel dentro de cada línea de exploración se numeran desde 0 a  $x_{\max}$ , de izquierda a derecha. Sin embargo, con una serie de comandos software, podemos configurar cualquier sistema de referencia que nos resulte cómodo para las posiciones de pantalla. Por ejemplo, podríamos especificar un rango entero para las posiciones de pantalla con el origen de coordenadas en la esquina inferior izquierda de una cierta área (Figura 3.1), o bien podríamos utilizar valores cartesianos no enteros para describir una imagen. Los valores de coordenadas que se utilicen para describir la geometría de una escena serán convertidos por las rutinas de visualización a posiciones de píxel enteras dentro del búfer de imagen.

Los algoritmos de líneas de exploración para las primitivas gráficas utilizan las descripciones de coordenadas que definen los objetos para determinar la ubicación de los píxeles que hay que mostrar. Por ejemplo, dadas las coordenadas de los extremos de un segmento de línea, un algoritmo de visualización debe calcular las posiciones para los píxeles comprendidos en la línea definida entre los dos puntos extremos. Puesto que una posición de píxel ocupa un área finita en la pantalla, es preciso tener en cuenta ese tamaño finito de los píxeles dentro de los algoritmos de implementación. Por el momento, vamos a suponer que cada posición entera en la pantalla hace referencia al centro de un área de píxel (en la Sección 3.13 consideraremos otros esquemas alternativos de direccionamiento de píxeles). Una vez identificadas las posiciones de los píxeles para un objeto, hay que almacenar los valores de color apropiados dentro del búfer de imagen. Con este propósito vamos a suponer que tenemos disponible un procedimiento de bajo nivel de la forma



**FIGURA 3.1.** Posiciones de píxel referenciadas con respecto a la esquina inferior izquierda de un área de la pantalla.

```
setPixel (x, y);
```

Este procedimiento almacena el valor actual de color dentro del búfer de imagen, en la posición entera  $(x, y)$ , relativa a la posición seleccionada para el origen de las coordenadas de la pantalla. En ocasiones, se hace necesario también consultar el valor actualmente almacenado en el búfer de imagen para una determinada ubicación de píxel. Por tanto, vamos a suponer que se dispone de la siguiente función de bajo nivel para obtener un valor de color del búfer de imagen:

```
getPixel (x, y, color);
```

En esta función, el parámetro `color` recibe un valor entero que se corresponde con los códigos RGB combinados almacenados para el píxel especificado en la posición  $(x, y)$ .

Aunque sólo necesitamos los valores de color en las posiciones  $(x, y)$  para una imagen bidimensional, se necesita información adicional de coordenadas de pantalla para las escenas tridimensionales. En este caso, las coordenadas de pantalla se almacenan como valores tridimensionales, haciendo referencia la tercera dimensión a la profundidad de las posiciones de objeto en relación con una determinada posición de visualización. Para una escena bidimensional, todos los valores de profundidad son 0.

## Especificaciones absolutas y relativas de coordenadas

Hasta ahora, las referencias de coordenadas de las que hemos hablado son las que se denominan valores de **coordenadas absolutas**. Esto quiere decir que los valores especificados son las posiciones reales dentro del sistema de coordenadas que se esté utilizando.

Sin embargo, algunos paquetes gráficos también permiten especificar las posiciones utilizando **coordenadas relativas**. Este método resulta útil para diversas aplicaciones gráficas, como por ejemplo para generar imágenes con trazadores de tinta, para sistemas de dibujo artístico y para paquetes gráficos para aplicaciones de autoedición e impresión. Tomando este enfoque, podemos especificar una posición de coordenadas en forma de un desplazamiento a partir de la última posición a la que se hubiera hecho referencia (denominada **posición actual**). Por ejemplo, si la ubicación  $(3, 8)$  es la última posición a la que se ha hecho referencia dentro de un programa de aplicación, una especificación de coordenadas relativas  $(2, -1)$  se corresponde con una posición absoluta  $(5, 7)$ . Por tanto, se utiliza una función adicional para establecer una posición actual antes de poder especificar ninguna coordenada para las funciones primitivas. Para describir un objeto, como por ejemplo una serie de segmentos de línea conectados, necesitaremos entonces proporcionar únicamente una secuencia de coordenadas relativas (desplazamientos) después de haber establecido una posición inicial. Dentro de un sistema gráfico, pueden proporcionarse opciones que permitan especificar las ubicaciones utilizando coordenadas relativas o absolutas. En las secciones que siguen, vamos a suponer que todas las coordenadas se especifican como referencias absolutas, a menos que se indique explícitamente lo contrario.

## 3.2 ESPECIFICACIÓN DE UN SISTEMA BIDIMENSIONAL DE REFERENCIA UNIVERSAL EN OpenGL

---

En nuestro primer ejemplo de programa (Sección 2.9), presentamos el comando `gluOrtho2D`, que es una función que podemos utilizar para fijar cualquier sistema de referencia bidimensional cartesiano. Los argumentos de esta función son los cuatro valores que definen los límites de coordenadas  $x$  e  $y$  para la imagen que queremos mostrar. Puesto que la función `gluOrtho2D` especifica una proyección ortogonal, necesitamos también asegurarnos de que los valores de coordenadas se encuentren dentro de la matriz de proyección de OpenGL. Además, podríamos asignar la matriz identidad como matriz de proyección antes de definir el rango de coordenadas del mundo. Esto garantizaría que los valores de coordenadas no se acumularan con cualesquiera valores que pudieramos haber establecido previamente para la matriz de proyección. Así, para nuestros ejemplos bidimensionales iniciales, podemos definir el sistema de coordenadas para la ventana de visualización en pantalla mediante las siguientes instrucciones:

```
glMatrixMode (GL_PROJECTION);
glLoadIdentity ( );
gluOrtho2D (xmin, xmax, ymin, ymax);
```

La ventana de visualización estará entonces referenciada por las coordenadas  $(xmin, ymin)$  en la esquina inferior izquierda y por las coordenadas  $(xmax, ymax)$  en la esquina superior derecha, como se muestra en la Figura 3.2.

A continuación, podemos designar una o más primitivas gráficas para visualización utilizando la referencia de coordenadas especificada en la instrucción `gluOrtho2D`. Si las coordenadas de extensión de una primitiva se encuentran dentro del rango de coordenadas de la ventana de visualización, podremos ver en ésta toda la primitiva. En caso contrario, sólo se mostrarán aquellas partes de la primitiva que caigan dentro de los límites de coordenadas de la ventana de visualización. Asimismo, cuando establezcamos la geometría que describe una imagen, todas las posiciones para las primitivas OpenGL deben proporcionarse en coordenadas absolutas con respecto al sistema de referencia definido en la función `gluOrtho2D`.

## 3.3 FUNCIONES DE PUNTO EN OpenGL

---

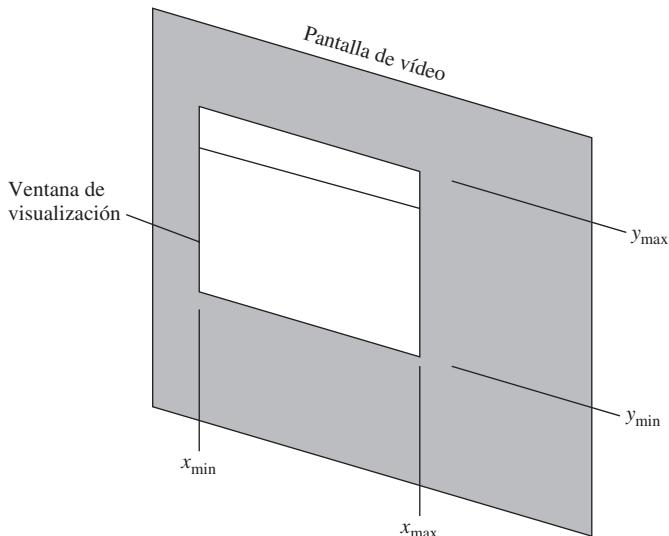
Para especificar la geometría de un punto, simplemente proporcionamos las correspondientes coordenadas dentro del sistema de referencia del mundo. A continuación, estas coordenadas, junto con las demás descripciones geométricas que tengamos en nuestro esquema, se pasan a las rutinas de visualización. A menos que especifiquemos otros valores de atributos, las primitivas de OpenGL se muestran con un tamaño y un color predeterminados. El color predeterminado para las primitivas es el blanco y el tamaño de punto predeterminado es igual al tamaño de un píxel de la pantalla.

Se utiliza la siguiente función OpenGL para indicar los valores de coordenadas para una única posición:

```
 glVertex* ( );
```

donde el asterisco (\*) indica que esta función necesita códigos de sufijo. Estos códigos de sufijo se utilizan para identificar la dimensión espacial, el tipo de datos numérico que hay que utilizar para los valores de las coordenadas y una posible forma vectorial para la especificación de las coordenadas. La función `glVertex` debe situarse entre una función `glBegin` y una función `glEnd`. El argumento de la función `glBegin` se utiliza para identificar el tipo de primitiva gráfica que hay que visualizar, mientras que `glEnd` no toma ningún argumento. Para dibujar puntos, el argumento de la función `glBegin` es la constante simbólica `GL_POINTS`. Así, la manera de especificar en OpenGL la posición de un punto es

```
glBegin (GL_POINTS);
glVertex* ( );
glEnd ( );
```



**FIGURA 3.2.** Límites de las coordenadas del mundo para una ventana de visualización especificada mediante la función `gluOrtho2D`.

Aunque el término *vértice* hace referencia estrictamente a una «esquina» de un polígono, a un punto de intersección de los lados de un ángulo, al punto de intersección de una elipse con su eje principal o a otras posiciones de coordenadas similares dentro de estructuras geométricas, se emplea la función `glVertex` para especificar coordenadas para las posiciones de los puntos. De esta forma, se utiliza una misma función para los puntos, las líneas y los polígonos; en la mayoría de los casos, se emplean recubrimientos poligonales para describir los objetos de una escena.

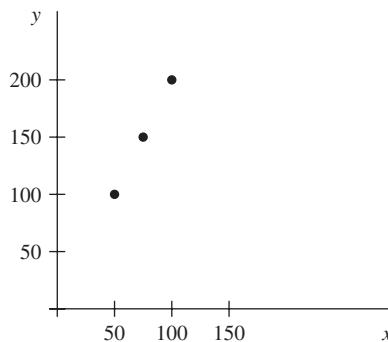
Las coordenadas en OpenGL pueden proporcionarse en dos, tres o cuatro dimensiones. Se utiliza un valor de sufijo de 2, 3 o 4 en la función `glVertex` para indicar la dimensionalidad de unas coordenadas. Una especificación tetradimensional indica una representación en *coordenadas homogéneas*, donde el parámetro *homogéneo h* (la cuarta coordenada) es un factor de escala para los valores de coordenadas cartesianas. Las representaciones en coordenadas homogéneas son útiles para expresar operaciones de transformación en forma matricial, y hablaremos de ellas en detalles en el Capítulo 5. Puesto que OpenGL trata el caso bidimensional como un caso especial del tridimensional, cualquier valor de coordenadas  $(x, y)$  es equivalente a  $(x, y, 0)$  con  $h = 1$ .

También es necesario indicar el tipo de dato que se está usando para los valores numéricos que especifican las coordenadas. Esto se hace con un segundo código de sufijo en la función `glVertex`. Los códigos de sufijo para especificar el tipo de datos numéricos son `i` (integer), `s` (short), `f` (float) y `d` (double). Finalmente, los valores de las coordenadas pueden enumerarse explícitamente en la función `glVertex`, o bien puede utilizarse un único argumento que refiera las coordenadas en forma matricial. Si utilizamos una especificación matricial para las coordenadas, tenemos que añadir un tercer código de sufijo: `v` (vector).

En el siguiente ejemplo se dibujan tres puntos equiespaciados a lo largo de un trayecto lineal bidimensional con una pendiente igual a 2 (Figura 3.3). Las coordenadas se proporcionan como parejas de números enteros:

```
glBegin (GL_POINTS);
    glVertex2i (50, 100);
    glVertex2i (75, 150);
    glVertex2i (100, 200);
glEnd ( );
```

Alternativamente, podríamos especificar las coordenadas de los puntos anteriores mediante matrices, como en:



**FIGURA 3.3.** Visualización de tres puntos generados con `glBegin (GL_POINTS)`.

```
int point1 [ ] = {50, 100};
int point2 [ ] = {75, 150};
int point3 [ ] = {100, 200};
```

e invocar a las funciones OpenGL de dibujo de los tres puntos de la forma siguiente:

```
glBegin (GL_POINTS);
    glVertex2iv (point1);
    glVertex2iv (point2);
    glVertex2iv (point3);
glEnd ( );
```

He aquí un ejemplo de especificación de dos puntos en un marco de referencia tridimensional. En este caso, proporcionamos las coordenadas como valores explícitos en coma flotante.

```
glBegin (GL_POINTS);
    glVertex3f (-78.05, 909.72, 14.60);
    glVertex3f (261.91, -5200.67, 188.33);
glEnd ( );
```

También podríamos definir una clase o estructura (`struct`) C++ para especificar puntos en varias dimensiones. Por ejemplo:

```
class wcPt2D {
public:
    GLfloat x, y;
};
```

Utilizando esta definición de clase, podríamos especificar un punto en coordenadas bidimensionales universales mediante las instrucciones:

```
wcPt2D pointPos;

pointPos.x = 120.75;
pointPos.y = 45.30;
glBegin (GL_POINTS);
    glVertex2f (pointPos.x, pointPos.y);
glEnd ( );
```

Y podemos emplear las funciones OpenGL de dibujo de puntos dentro de un procedimiento C++ para implementar el comando `setPixel`.

## 3.4 FUNCIONES OpenGL PARA LÍNEAS

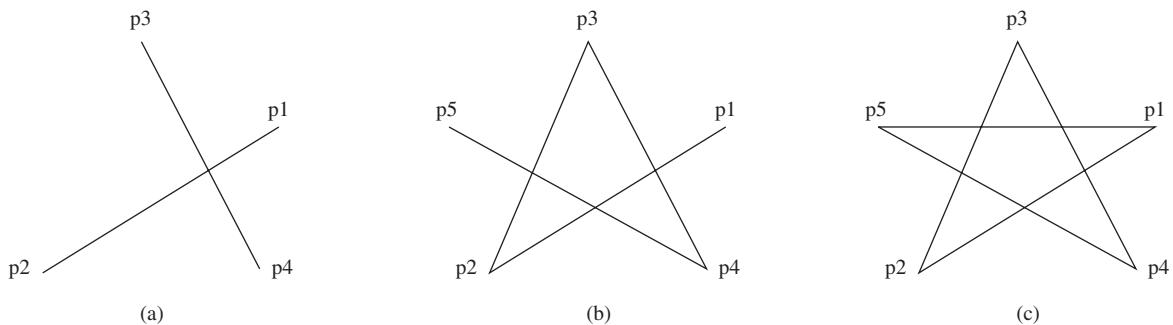
Los paquetes gráficos proporcionan normalmente una función para especificar uno o más segmentos de línea rectos, donde cada segmento de línea esté definido por las coordenadas de los dos extremos. En OpenGL, podemos seleccionar la posición de las coordenadas de uno de los extremos utilizando la función `glVertex`, al igual que hicimos para el caso de un punto. Y también podemos encerrar una lista de funciones `glVertex` entre la pareja de comandos `glBegin/glEnd`. Pero ahora vamos a emplear una constante simbólica como argumento para la función `glBegin` con el fin de que esta función interprete una lista de posiciones como las coordenadas de los extremos de una serie de segmentos lineales. Hay tres constantes simbólicas en OpenGL que podemos usar para especificar cómo debe conectarse una lista de vértices para formar un conjunto de segmentos de línea rectos. De manera predeterminada, cada una de esas constantes simbólicas muestra líneas blancas continuas.

Para generar un conjunto de segmentos de línea rectos entre pares sucesivos de puntos de una lista, se utiliza la constante de línea primitiva `GL_LINES`. En general, esto da como resultado un conjunto de líneas no conectadas, a menos que repitamos algunos de los puntos. Si sólo se especifica un único punto, no se visualizará nada; asimismo, si el número de puntos es impar, el último de esos números no será procesado. Por ejemplo, si tenemos cinco coordenadas de puntos, etiquetadas como `p1` a `p5`, y cada uno de los puntos está representado como una matriz bidimensional, el siguiente código generaría el gráfico que se muestra en la Figura 3.4(a).

```
glBegin (GL_LINES);
    glVertex2iv (p1);
    glVertex2iv (p2);
    glVertex2iv (p3);
    glVertex2iv (p4);
    glVertex2iv (p5);
glEnd ( );
```

Como vemos, se obtiene un segmento de línea entre el primer y el segundo puntos y otro segmento de línea entre los puntos tercero y cuarto. En este caso, el número de puntos especificados es impar, por lo que se ignoran las coordenadas del último punto.

Con la constante primitiva OpenGL `GL_LINE_STRIP`, lo que se obtiene es una **polilínea**. En este caso, el gráfico es una secuencia de segmentos de línea conectados que van desde el primer punto de la lista hasta el último. El primer segmento de la polilínea se traza entre el primer y el segundo puntos; el segundo segmento de línea irá desde el segundo al tercer punto, etc., hasta llegar al último vértice. El gráfico estará vacío si no incluimos las coordenadas de al menos dos puntos. Utilizando los mismos cinco conjuntos de coordenadas que en el ejemplo anterior, se obtiene el gráfico de la Figura 3.4(b) con el código



**FIGURA 3.4.** Segmentos de línea que pueden visualizarse en OpenGL utilizando una lista de cinco vértices. (a) Un conjunto de líneas no conectado generado con la constante de línea primitiva `GL_LINES`. (b) Una polilínea generada con `GL_LINE_STRIP`. (c) Una polilínea cerrada generada con `GL_LINE_LOOP`.

```

glBegin (GL_LINE_STRIP);
    glVertex2iv (p1);
    glVertex2iv (p2);
    glVertex2iv (p3);
    glVertex2iv (p4);
    glVertex2iv (p5);
glEnd ( );

```

La tercera primitiva de línea OpenGL es `GL_LINE_LOOP`, que genera una **polilínea cerrada**. Se añade una línea adicional a la secuencia de líneas del ejemplo anterior, de modo que se conecta el último vértice de la polilínea con el primero. La Figura 3.4(c) muestra el gráfico correspondiente a nuestra lista de puntos cuando se selecciona esta opción para el trazado de líneas.

```

glBegin (GL_LINE_LOOP);
    glVertex2iv (p1);
    glVertex2iv (p2);
    glVertex2iv (p3);
    glVertex2iv (p4);
    glVertex2iv (p5);
glEnd ( );

```

Como hemos indicado anteriormente, los componentes de las imágenes se describen en un marco de referencia universal que al final se hace corresponder con el sistema de coordenadas del dispositivo de salida. Entonces, la información geométrica relativa a la imagen se convierte en posiciones de píxel. En la siguiente sección, vamos a examinar los algoritmos de conversión que se utilizan para implementar las funciones de línea OpenGL.

## 3.5 ALGORITMOS DE DIBUJO DE LÍNEAS

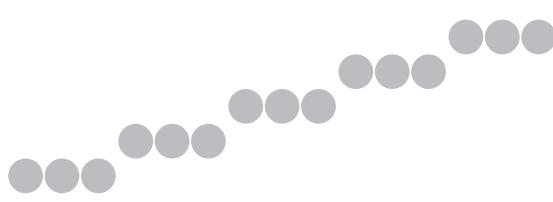
---

Un segmento de línea recta dentro de una escena está definido por las coordenadas de los dos extremos del segmento. Para mostrar la línea en un monitor digital, el sistema gráfico debe primero proyectar las coordenadas de los extremos para obtener coordenadas de pantalla de valor entero y determinar las posiciones de píxel más próximas a lo largo de la línea que conecta los dos extremos. Entonces, se cargará en el búfer de imagen el color correspondiente a la línea en las coordenadas de píxel apropiadas. Al leer los datos del búfer de imagen, la controladora de vídeo dibujará los píxeles en pantalla. Este proceso lo que hace es digitalizar la línea para obtener un conjunto de posiciones enteras discretas que, en general, únicamente sirve como aproximación del verdadero trayecto seguido por la línea. Una posición de línea calculada como (10.48, 20.51), por ejemplo, se convierte a la posición de píxel (10, 21). Este redondeo de los valores de las coordenadas para obtener enteros hace que todas las líneas (excepto las horizontales y las verticales) se dibujen con una apariencia escalonada, como puede verse en la Figura 3.5. La forma escalonada característica de las líneas digitalizadas es particularmente apreciable en los sistemas de baja resolución, pudiéndose mejorar un poco su apariencia si se utiliza un monitor de resolución más alta. Otras técnicas más efectivas para suavizar la línea digitalizada se basan en ajustar las intensidades de los píxeles a lo largo del trayecto de la línea (Sección 4.17).

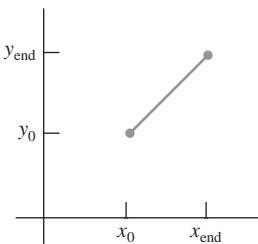
### Ecuaciones de las líneas

Para determinar las posiciones de los píxeles a lo largo de un trayecto de línea recta se utilizan las propiedades geométricas de la línea. La *ecuación punto-pendiente* cartesiana para una línea recta es:

$$y = m \cdot x + b \quad (3.1)$$



**FIGURA 3.5.** Efecto de escalonamiento que se produce cuando se genera una línea mediante una serie de posiciones de píxel.



**FIGURA 3.6.** Trayecto lineal entre dos vértices  $(x_0, y_0)$  y  $(x_{\text{end}}, y_{\text{end}})$ .

siendo  $m$  la pendiente de la línea y  $b$  el punto de intersección con el eje  $y$ . Puesto que los dos extremos del segmento de línea tienen las coordenadas  $(x_0, y_0)$  y  $(x_{\text{end}}, y_{\text{end}})$ , como se muestra en la Figura 3.6, podemos determinar los valores de la pendiente  $m$  y de punto  $b$  de intersección con el eje  $y$  mediante las fórmulas siguientes:

$$m = \frac{y_{\text{end}} - y_0}{x_{\text{end}} - x_0} \quad (3.2)$$

$$b = y_0 - m \cdot x_0 \quad (3.3)$$

Los algoritmos para la visualización de líneas rectas están basados en la Ecuación 3.1 y en los cálculos indicados por las Ecuaciones 3.2 y 3.3.

Para cualquier intervalo horizontal  $\delta x$  a lo largo de una línea, podemos calcular el correspondiente intervalo vertical  $\delta y$  a partir de la Ecuación 3.2, de la forma siguiente:

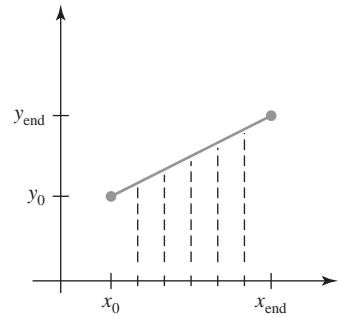
$$\delta y = m \cdot \delta x \quad (3.4)$$

De forma similar, podemos obtener el intervalo  $\delta x$  correspondiente a un valor  $\delta y$  especificado mediante la fórmula:

$$\delta x = \frac{\delta y}{m} \quad (3.5)$$

Estas ecuaciones forman la base para determinar las tensiones de deflexión en los monitores analógicos, como por ejemplo los sistemas de visualización vectorial, donde pueden conseguirse cambios arbitrariamente pequeños en la tensión de deflexión. Para líneas con un valor de pendiente  $|m| < 1$ ,  $\delta x$  puede definirse de forma proporcional a una pequeña tensión de deflexión horizontal, y entonces la correspondiente deflexión vertical será proporcional al valor de  $\delta y$  calculada a partir de la Ecuación 3.4. Para las líneas cuyas pendientes tienen un módulo  $|m| > 1$ , puede asignar a  $\delta y$  un valor proporcional a una pequeña tensión de deflexión vertical, con lo que la correspondiente tensión de deflexión horizontal será proporcional a  $\delta x$ , que se calcula a partir de la Ecuación 3.5. Para las líneas con  $m = 1$ ,  $\delta x = \delta y$  y las tensiones de deflexión horizontal y vertical serán iguales. En cada uno de los casos, se generará una línea perfectamente recta y con pendiente  $m$  entre los dos extremos especificados.

En los monitores digitales, las líneas se dibujan mediante píxeles, y los pasos en la dirección horizontal o vertical están restringidos por la separación entre los píxeles. En otras palabras, debemos “muestrear” la línea en posiciones discretas y determinar el píxel más cercano a la línea en cada posición de muestreo. Este proceso de digitalización de las líneas rectas se ilustra en la Figura 3.7, utilizando posiciones de muestreo discretas a lo largo del eje  $x$ .



**FIGURA 3.7.** Segmento de línea recta con cinco posiciones de muestreo a lo largo del eje  $x$  entre  $x_0$  y  $x_{end}$ .

### Algoritmo DDA

El algoritmo de *análisis diferencial digital* (DDA, Digital Differential Analyzer) es un algoritmo de digitalización de líneas basado en calcular  $\delta y$  o  $\delta x$  utilizando la Ecuación 3.4 o la Ecuación 3.5. Las líneas se muestrean a intervalos unitarios según una de las coordenadas y los correspondientes valores enteros más próximos al trayecto lineal se calculan para la otra coordenada.

Vamos a considerar primero una línea con pendiente positiva, como se muestra en la Figura 3.6. Si la pendiente es menor o igual que 1, muestreemos a intervalos unitarios según el eje de las  $x$  ( $\delta x = 1$ ) y calculamos los valores sucesivos de  $y$  como:

$$y_{k+1} = y_k + m \quad (3.6)$$

El subíndice  $k$  adopta valores enteros comenzando en 0 para el primer punto e incrementándose en una unidad cada vez hasta que se alcanza el extremo de la línea. Puesto que  $m$  puede ser cualquier número real entre 0.0 y 1.0, cada valor calculado de  $y$  debe redondearse al entero más próximo, lo que nos dará la posición de un píxel de pantalla en la columna  $x$  que estemos procesando.

Para las líneas con una pendiente positiva superior a 1.5, invertimos los papeles  $x$  e  $y$ , es decir, muestreamos a intervalos unitarios de  $y$  ( $\delta y = 1$ ) y calculamos los valores de  $x$  consecutivos mediante:

$$x_{k+1} = x_k + \frac{1}{m} \quad (3.7)$$

En este caso, cada valor de  $x$  calculado se redondea a la posición de píxel más cercana en la línea de exploración y actual.

Las Ecuaciones 3.6 y 3.7 están basadas en la suposición de que las líneas deben procesarse desde el extremo situado más a la izquierda hasta el extremo situado más a la derecha (Figura 3.6). Si invertimos este procesamiento, de modo que se tome como punto inicial el situado a la derecha, entonces tendremos  $\delta x = -1$  y

$$y_{k+1} = y_k - m \quad (3.8)$$

o (cuando la pendiente sea superior a 1) tendremos  $\delta y = -1$  con

$$x_{k+1} = x_k - \frac{1}{m} \quad (3.9)$$

Se realizarán cálculos similares utilizando las Ecuaciones 3.6 a 3.9 para determinar las posiciones de los píxeles a lo largo de una línea con pendiente negativa. Así, si el valor absoluto de la pendiente es inferior a 1 y el punto inicial se encuentra a la izquierda, haremos  $\delta x = 1$  y calcularemos los valores de  $y$  mediante la Ecuación 3.6. Cuando el punto inicial esté a la derecha (para la misma pendiente), haremos  $\delta x = -1$  y obtendremos los valores  $y$  utilizando la Ecuación 3.8. Para una pendiente negativa con valor absoluto superior a 1, usaremos  $\delta y = -1$  y la Ecuación 3.9 o  $\delta y = 1$  y la Ecuación 3.7.

Este algoritmo se resume en el siguiente procedimiento, que acepta como entrada dos posiciones enteras en la pantalla correspondientes a los dos extremos de un segmento lineal. Las diferencias horizontales y verticales entre los dos extremos se asignan a los parámetros  $dx$  y  $dy$ . La diferencia con mayor valor absoluto determina el valor del parámetro  $steps$ . Partiendo de la posición  $(x_0, y_0)$ , determinemos el desplazamiento requerido en cada paso para generar el siguiente píxel de la línea. Este proceso se ejecuta en bucle un número de veces igual a  $steps$ . Si el módulo de  $dx$  es superior al módulo de  $dy$  y  $x_0$  es inferior a  $xEnd$ , los valores de los incrementos en las direcciones  $x$  e  $y$  son 1 y  $m$ , respectivamente. Si el cambio mayor se realiza en la dirección  $x$ , pero  $x_0$  es superior a  $xEnd$ , entonces se utilizan los decrementos  $-1$  y  $-m$  para generar cada nuevo punto de la línea. En los restantes casos, utilizamos un incremento (o decremento) unitario en la dirección  $y$  y un incremento (o decremento) para  $x$  igual a  $\frac{1}{m}$ .

---

```
#include <stdlib.h>
#include <math.h>

inline int round (const float a) { return int (a + 0.5); }

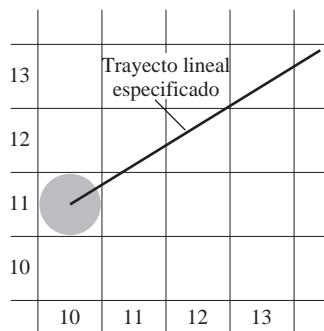
void lineDDA (int x0, int y0, int xEnd, int yEnd)
{
    int dx = xEnd - x0, dy = yEnd - y0, steps, k;
    float xIncrement, yIncrement, x = x0, y = y0;

    if (fabs (dx) > fabs (dy))
        steps = fabs (dx);
    else
        steps = fabs (dy);
    xIncrement = float (dx) / float (steps);
    yIncrement = float (dy) / float (steps);

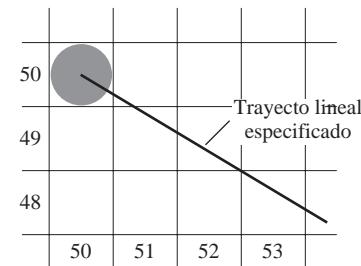
    setPixel (round (x), round (y));
    for (k = 0; k < steps; k++) {
        x += xIncrement;
        y += yIncrement;
        setPixel (round (x), round (y));
    }
}
```

---

El algoritmo DDA es un método para el cálculo de posiciones de píxeles más rápido que implementar directamente la Ecuación 3.1. Se elimina la multiplicación de la Ecuación 3.1 haciendo uso de las propias características del proceso de digitalización, aplicándose los incrementos apropiados en las direcciones  $x$  o  $y$  para pasar de una posición de píxel a la siguiente a lo largo de la línea. Sin embargo, la acumulación de errores de redondeo en las sucesivas sumas del incremento de coma flotante pueden hacer que las posiciones de píxel sufran cierta deriva con respecto al verdadero trayecto lineal, para segmentos lineales largos. Además, las operaciones de redondeo y la aritmética en coma flotante inherentes a este procedimiento siguen consumiendo mucho tiempo. Podemos mejorar la velocidad del algoritmo DDA separando los incrementos  $m$  y  $\frac{1}{m}$  en sus partes entera y fraccionaria, reduciendo así todos los cálculos a operaciones con números enteros. En la Sección 4.10 se explica el método para calcular los incrementos  $\frac{1}{m}$  en pasos enteros. Además, en la siguiente sección vamos a considerar una técnica más genérica de digitalización de líneas que puede aplicarse tanto a líneas como a curvas.



**FIGURA 3.8.** Una sección de una pantalla donde hay que dibujar un segmento lineal, comenzando a partir del píxel situado en la columna 10 de la línea de exploración 11.



**FIGURA 3.9.** Una sección de una pantalla donde hay que dibujar un segmento lineal de pendiente negativa, partiendo del píxel situado en la columna 50 de la línea de exploración 50.

### Algoritmo de Bresenham para dibujo de líneas

En esta sección, vamos a presentar un algoritmo preciso y eficiente para la generación de líneas digitalizadas; este algoritmo, inventado por Bresenham, utiliza sólo cálculos enteros para determinar los incrementos. Además, el algoritmo de Bresenham para dibujo de líneas puede adaptarse para dibujar círculos y otras líneas. Las Figuras 3.8 y 3.9 muestran sendas secciones de una pantalla en las que tenemos que dibujar dos segmentos lineales. El eje vertical muestra las posiciones de las líneas de exploración, mientras que el eje horizontal identifica las columnas de píxeles. Muestreando a intervalos unitarios según el eje  $x$  en estos ejemplos, necesitamos decidir cuál de las dos posibles posiciones de píxel está más próxima al trayecto lineal en cada paso de muestreo. Comenzando a partir del extremo izquierdo mostrado en la Figura 3.8, necesitamos determinar en la siguiente posición de muestreo si debemos dibujar el píxel correspondiente a la posición (11, 11) o el de la posición (11, 12). De forma similar, la Figura 3.9 muestra un trayecto lineal con pendiente negativa que comienza a partir del extremo izquierdo situado en la posición de píxel (50, 50). En este caso, ¿debemos seleccionar como siguiente posición de píxel las coordenadas (51, 50) o (51, 49)? Estas preguntas se responden con el algoritmo de Bresenham comprobando el signo de un parámetro entero cuyo valor es proporcional a la diferencia entre las separaciones verticales de las dos posiciones de píxel con respecto al trayecto lineal.

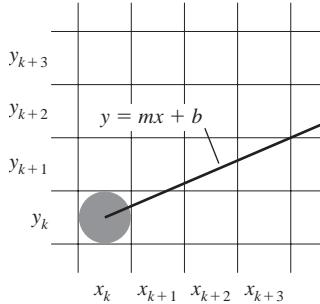
Para ilustrar el algoritmo de Bresenham, vamos primero a analizar el proceso de digitalización de líneas con pendiente positiva inferior a 1.0. Las posiciones de píxel a lo largo de un trayecto lineal se determinan entonces muestreando a intervalos unitarios según el eje  $x$ . Comenzando a partir del extremo izquierdo  $(x_0, y_0)$  de una línea dada, vamos recorriendo cada una de las sucesivas columnas (posición  $x$ ) y dibujando el píxel cuyo valor  $y$  sea más próximo al trayecto lineal. La Figura 3.10 ilustra el paso  $k$ -ésimo de este proceso. Supongamos que hemos determinado que hay que dibujar el píxel situado en  $(x_k, y_k)$ ; entonces tendremos que decidir qué píxel dibujar en la columna  $x_{k+1} = x_k + 1$ . Las dos opciones existentes son los píxeles de las posiciones  $(x_k + 1, y_k)$  y  $(x_k + 1, y_k + 1)$ .

En la posición de muestreo  $x_k + 1$ , etiquetamos las separaciones verticales de los píxeles con respecto al trayecto lineal matemático con los nombres  $d_{\text{lower}}$  y  $d_{\text{upper}}$  (Figura 3.11). La coordenada  $y$  de la línea matemática en la columna de píxel  $x_k + 1$  se calcula como:

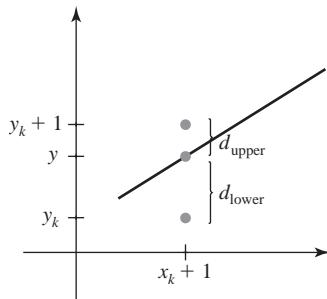
$$y = m(x_k + 1) + b \quad (3.10)$$

Entonces:

$$\begin{aligned} d_{\text{lower}} &= y - y_k \\ &= m(x_k + 1) + b - y_k \end{aligned} \quad (3.11)$$



**FIGURA 3.10.** Una sección de la pantalla que muestra un píxel de la columna  $x_k$  correspondiente a la línea de exploración  $y_k$  y que hay que dibujar como parte del trayecto de un segmento lineal con pendiente  $0 < m < 1$ .



**FIGURA 3.11.** Distancias verticales entre las posiciones de los píxeles y la coordenada y de la línea, en la posición de muestreo  $x_{k+1}$ .

y

$$\begin{aligned} d_{\text{upper}} &= (y_{k+1}) - y \\ &= y_k + 1 - m(x_k + 1) - b \end{aligned} \quad (3.12)$$

Para determinar cuál de los dos píxeles está más próximo a la línea, podemos realizar una comprobación muy eficiente que se basa en la diferencia entre las dos separaciones de los píxeles:

$$d_{\text{lower}} - d_{\text{upper}} = 2m(x_{k+1}) - 2y_k + 2b - 1 \quad (3.13)$$

Podemos obtener un parámetro de decisión  $p_k$  para el paso  $k$ -ésimo del algoritmo de digitalización de líneas reordenando la Ecuación 3.13 para que sólo haya que realizar cálculos enteros. Podemos hacer esto realizando la sustitución  $m = \Delta y / \Delta x$ , donde  $\Delta y$  y  $\Delta x$  son las separaciones vertical y horizontal entre los dos extremos de la línea, y definiendo el parámetro de decisión como

$$p_k = \Delta x(d_{\text{lower}} - d_{\text{upper}}) = 2\Delta y \cdot x_k - 2\Delta x \cdot y_k + c \quad (3.14)$$

El signo de  $p_k$  es igual al de  $d_{\text{lower}} - d_{\text{upper}}$  porque  $\Delta x > 0$  en nuestro ejemplo. El parámetro  $c$  es constante y tiene el valor  $2\Delta y + \Delta x(2b - 1)$ , que es independiente de la posición del píxel y se eliminará en los cálculos recursivos de  $p_k$ . Si el píxel de  $y_k$  está “más próximo” al trayecto lineal que el píxel de  $y_{k+1}$  (es decir,  $d_{\text{lower}} < d_{\text{upper}}$ ), entonces el parámetro de decisión  $p_k$  será negativo. En dicho caso, dibujaremos el píxel inferior; en caso contrario, dibujaremos el superior.

Los cambios de coordenadas a lo largo de la línea se producen en pasos unitarios en las direcciones  $x$  o  $y$ . Por tanto, podemos obtener los valores de los sucesivos parámetros de decisión utilizando cálculos enteros incrementales. En el paso  $k + 1$ , el parámetro de decisión se evalúa a partir de la Ecuación 3.14 como:

$$p_{k+1} = 2\Delta y \cdot x_{k+1} - 2\Delta x \cdot y_{k+1} + c$$

Pero si restamos la Ecuación 3.14 de ésta última ecuación, tendremos:

$$p_{k+1} - p_k = 2\Delta y(x_{k+1} - x_k) - 2\Delta x(y_{k+1} - y_k)$$

Y como  $x_{k+1} = x_k + 1$ , nos queda:

$$p_{k+1} = p_k + 2\Delta y - 2\Delta x(y_{k+1} - y_k) \quad (3.15)$$

donde el término  $y_{k+1} - y_k$  es 0 o 1, dependiendo del signo del parámetro  $p_k$ .

Este cálculo recursivo de los parámetros de decisión se realiza en cada posición entera  $x$  comenzando por el extremo izquierdo de la línea. El primer parámetro,  $p_0$ , se evalúa a partir de la Ecuación 3.14 en la posición inicial de píxel  $(x_0, y_0)$  y con  $m$  igual a  $\Delta y / \Delta x$ :

$$p_0 = 2\Delta y - \Delta x \quad (3.16)$$

Resumimos el algoritmo de Bresenham para líneas con una pendiente positiva inferior a 1 en el siguiente recuadro. Las constantes  $2\Delta y$  y  $2\Delta y - 2\Delta x$  se calculan una única vez para cada línea que hay que digitalizar, por lo que las operaciones aritméticas sólo requieren sumas y restas enteras de estas dos constantes.

### Algoritmo de Bresenham para dibujo de líneas con $|m| < 1.0$

1. Introducir los dos extremos de la línea y almacenar el extremo izquierdo en  $(x_0, y_0)$ .
2. Configurar el color para la posición  $(x_0, y_0)$  del búfer del imagen, es decir, dibujar el primer punto.
3. Calcular las constantes  $\Delta x$ ,  $\Delta y$ ,  $2\Delta y$  y  $2\Delta y - 2\Delta x$ , y obtener el valor inicial del parámetro de decisión, que será

$$p_0 = 2\Delta y - \Delta x$$

4. Para cada  $x_k$  a lo largo de la línea, comenzando en  $k = 0$ , realizar la siguiente comprobación. Si  $p_k < 0$ , el siguiente punto que hay que dibujar será  $(x_k + 1, y_k)$  y

$$p_{k+1} = p_k + 2\Delta y$$

En caso contrario, el siguiente punto que habrá que dibujar es  $(x_k + 1, y_k + 1)$  y

$$p_{k+1} = p_k + 2\Delta y - 2\Delta x$$

5. Realizar el Paso 4  $\Delta x - 1$  veces.

### Ejemplo 3.1 Dibujo de líneas mediante el algoritmo de Bresenham

Para ilustrar el algoritmo, vamos a digitalizar la línea definida por los vértices  $(20, 10)$  y  $(30, 18)$ . Esta línea tiene una pendiente de 0,8, con:

$$\Delta x = 10, \quad \Delta y = 8$$

El parámetro de decisión inicial tiene el valor:

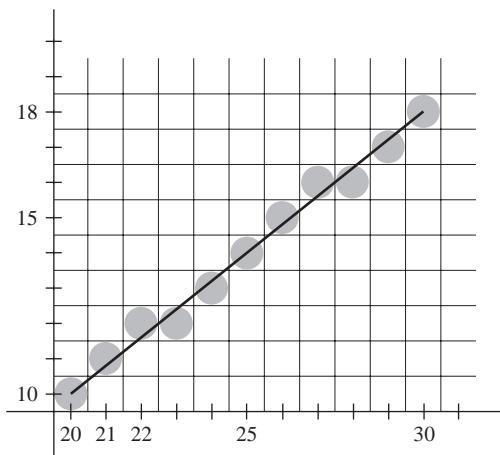
$$p_0 = 2\Delta y - \Delta x = 6$$

y los incrementos para calcular los sucesivos parámetros de decisión son:

$$2\Delta y = 16, \quad 2\Delta y - 2\Delta x = -4$$

Dibujamos el punto inicial  $(x_0, y_0) = (20, 10)$  y determinamos las posiciones sucesivas de los píxeles a lo largo de la línea teniendo en cuenta los valores del parámetro de decisión:

$k$	$p_k$	$(x_{k+1}, y_{k+1})$	$k$	$p_k$	$(x_{k+1}, y_{k+1})$	$k$	$p_k$	$(x_{k+1}, y_{k+1})$
0	6	(21, 11)	4	10	(25, 14)	7	-2	(28, 16)
1	2	(22, 12)	5	6	(26, 15)	8	14	(29, 17)
2	-2	(23, 12)	6	2	(27, 16)	9	10	(30, 18)
3	14	(24, 13)						



**FIGURA 3.12.** Posiciones de los píxeles a lo largo de la línea comprendida entre los vértices  $(20, 10)$  y  $(30, 18)$ , dibujada con el algoritmo de Bresenham.

En la Figura 3.12 se muestra una gráfica de los píxeles generados para este proyecto líreal.

En el siguiente procedimiento se da una implementación del algoritmo de dibujo de líneas de Bresenham para pendientes comprendidas en el rango  $0 < m < 1.0$ . En este píxel se introducen las coordenadas de los extremos de la línea y los píxeles se dibujan comenzando a partir del extremo izquierdo.

```
#include <stdlib.h>
#include <math.h>

/* Algoritmo de dibujo de líneas de Bresenham para  $|m| < 1.0$ . */
void lineBres (int x0, int y0, int xEnd, int yEnd)
{
    int dx = fabs (xEnd - x0), dy = fabs(yEnd - y0);
    int p = 2 * dy - dx;
    int twoDy = 2 * dy, twoDyMinusDx = 2 * (dy - dx);
    int x, y;

    /* Determinar qué extremo usar como posición inicial. */
    if (x0 > xEnd) {
        x = xEnd;
        y = yEnd;
        xEnd = x0;
    }
    else {
        x = x0;
        y = y0;
    }
    setPixel (x, y);

    while (x < xEnd) {
        x++;
        if (p < 0)
            p += twoDy;
```

```

        else {
            y++;
            p += twoDyMinusDx;
        }
        setPixel (x, y);
    }
}

```

El algoritmo de Bresenham puede generalizarse a líneas de pendiente arbitraria considerando la simetría existente entre los diversos octantes y cuadrantes del plano  $x$ . Para una línea con pendiente positiva superior a 1.0, intercambiamos los papeles de las direcciones  $x$  e  $y$ . En otras palabras, avanzamos paso a paso por la dirección  $y$  con incrementos unitarios y calculamos los valores de  $x$  sucesivos más próximos a la trayectoria de la línea. Asimismo, podríamos también revisar el programa para dibujar los píxeles comenzando a partir de cualquiera de los dos extremos. Si la posición inicial para una línea con pendiente positiva es el extremo derecho, habrá que decrementar tanto  $x$  como  $y$  a medida que avanzamos paso a paso de derecha a izquierda. Para garantizar que siempre se dibujen los mismos píxeles independientemente de cuál extremo se utilice como punto inicial, elegiremos siempre el pixel superior (o inferior) de los dos candidatos cuando las dos separaciones verticales con respecto al trayecto de la línea sean iguales ( $d_{lower} = d_{upper}$ ). Para pendientes negativas, los procedimientos son similares, salvo porque ahora una de las coordenadas se decrementa a medida que la otra se incrementa. Finalmente, los casos especiales pueden tratarse por separado: las líneas horizontales ( $\Delta y = 0$ ), las líneas verticales ( $\Delta x = 0$ ) y las líneas diagonales ( $|\Delta x| = |\Delta y|$ ) pueden cargarse directamente en el búfer de imagen sin necesidad de procesarlas mediante el algoritmo de dibujo de líneas.

## Visualización de polilíneas

La implementación de un procedimiento de dibujo de polilíneas se realiza invocando  $n - 1$  veces un algoritmo de dibujo de líneas, con el fin de visualizar las líneas que conectan los  $n$  vértices. Cada llamada sucesiva pasa al procedimiento la pareja de coordenadas necesaria para dibujar el siguiente segmento lineal, utilizando como primer punto el último punto de la anterior pasada. Después de haber cargado en el búfer de imagen los valores de color para las posiciones de píxeles situadas a lo largo del primer segmento lineal, procesamos los subsiguientes segmentos lineales comenzando con la siguiente posición de píxel situada después del primer vértice de este segmento. De esta forma, nos evitamos asignar dos veces el valor de color a algunos de los vértices. En la Sección 3.13 presentaremos con más detalle algunos métodos que se utilizan para evitar el solapamiento de los objetos visualizados.

## 3.6 ALGORITMOS PARALELOS DE DIBUJO DE LÍNEAS

Los algoritmos de generación de líneas que hemos presentado hasta ahora determinan las posiciones de los píxeles de manera secuencial. Utilizando procesamiento en paralelo, podemos calcular múltiples posiciones de píxel simultáneamente a lo largo del trayecto de línea, dividiendo los cálculos entre los diversos procesadores disponibles. Una técnica para resolver el problema del particionamiento consiste en adaptar un algoritmo secuencial existente con el fin de aprovechar la existencia de múltiples procesadores. Alternativamente, podemos examinar otras formas de realizar el procesamiento que permitan calcular eficientemente en paralelo las posiciones de los píxeles. Una consideración importante que hay que tener presente a la hora de desarrollar un algoritmo paralelo es que hay que equilibrar la carga de procesamiento entre todos los procesadores disponibles.

Si tenemos  $n_p$  procesadores, podemos implementar un algoritmo paralelo de dibujo de líneas por el método de Bresenham subdividiendo el trayecto lineal en  $n_p$  particiones y generando simultáneamente los segmentos lineales correspondientes a cada uno de estos subintervalos. Para una línea con pendiente  $0 < m < 1.0$  y

un extremo izquierdo con coordenadas  $(x_0, y_0)$ , particionaremos la línea a lo largo de la dirección  $x$  positiva. La distancia entre las posiciones  $x$  iniciales de las particiones adyacentes puede calcularse como:

$$\Delta x_p = \frac{\Delta x + n_p - 1}{n_p} \quad (3.17)$$

donde  $\Delta x$  es la anchura de la línea y el valor  $\Delta x_p$  de anchura de la partición se calcula empleando una división entera. Si numeramos las particiones y los procesadores como 0, 1, 2, hasta  $n_p - 1$ , podemos calcular la coordenada  $x$  inicial de la partición  $k$ -ésima mediante la fórmula:

$$x_k = x_0 + k\Delta x_p \quad (3.18)$$

Como ejemplo si tenemos  $n_p = 4$  procesadores con  $\Delta x = 15$ , la anchura de las particiones será 4 y los valores  $x$  iniciales de las particiones serán  $x_0, x_0 + 4, x_0 + 8$  y  $x_0 + 12$ . Con este esquema de particionamiento, la anchura del último subintervalo (el situado más a la derecha) será menor que la de los otros en algunos casos. Además, si los extremos de cada línea no tienen valores enteros, los errores de truncamiento pueden hacer que existan anchuras de partición variables a lo largo de la línea.

Para aplicar el algoritmo de Bresenham a las particiones, necesitamos el valor inicial de la coordenada  $y$  y el valor inicial del parámetro de decisión para cada partición. El cambio  $\Delta y_p$  en la partición  $y$  para cada partición se calcula a partir de la pendiente  $n$  de la línea  $y$  y de la anchura  $\Delta x_p$  de la partición:

$$\Delta y_p = m\Delta x_p \quad (3.19)$$

Para la partición  $k$ -ésima, la coordenada  $y$  inicial será entonces

$$y_k = y_0 + \text{round}(k\Delta y_p) \quad (3.20)$$

El parámetro de decisión inicial para el algoritmo de Bresenham al principio del subintervalo  $k$ -ésimo se obtiene a partir de la Ecuación 3.14:

$$p_k = (k\Delta x_p)(2\Delta y) - \text{round}(k\Delta y_p)(2\Delta x) + 2\Delta y - \Delta x \quad (3.21)$$

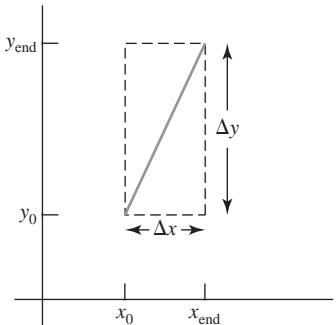
Cada procesador calculará entonces las posiciones de los píxeles para su subintervalo asignado, utilizando el valor inicial del parámetro de decisión que acabamos de calcular y las coordenadas iniciales  $(x_k, y_k)$ . Los cálculos en coma flotante pueden reducirse a operaciones aritméticas enteras utilizando los valores iniciales  $y_k$  y  $p_k$  pero sustituyendo  $m = \Delta y / \Delta x$  y reordenando los términos. Podemos ampliar el algoritmo paralelo de Bresenham para una línea con pendiente superior a 1.0 particionando la línea en la dirección  $y$  y calculando los valores  $x$  iniciales para las distintas particiones. Para las pendientes negativas, incrementaremos los valores de la coordenada en una dirección y decrementaremos en la otra.

Otra forma de implementar algoritmos paralelos en sistemas digitales consiste en asignar a cada procesador un grupo concreto de píxeles de la pantalla. Con un número suficiente de procesadores, podemos asignar cada procesador a un píxel dentro de una determinada zona de la pantalla. Esta técnica puede adaptarse a la visualización de líneas asignando un procesador a cada uno de los píxeles comprendidos dentro de los límites de las extensiones de coordenadas de la línea y calculando las distancias de los píxeles con respecto al trayecto lineal. El número de píxeles dentro del recuadro de contorno de una línea es  $\Delta x \cdot \Delta y$  (Figura 3.13). La distancia perpendicular  $d$  entre la línea de la Figura 3.13 y un píxel de coordenadas  $(x, y)$  se obtiene mediante la fórmula:

$$d = Ax + By + C \quad (3.22)$$

donde

$$A = \frac{-\Delta y}{\text{longitudlínnea}}$$



**FIGURA 3.13.** Recuadro de contorno para una línea con separaciones  $\Delta x$  y  $\Delta y$  entre los dos extremos.

$$B = \frac{\Delta x}{\text{longitudlínea}}$$

$$C = \frac{x_0 \Delta y - y_0 \Delta x}{\text{longitudlínea}}$$

con

$$\text{longitudlínea} = \sqrt{\Delta x^2 + \Delta y^2}$$

Una vez evaluadas las constantes  $A$ ,  $B$  y  $C$  para la línea, cada procesador debe realizar dos multiplicaciones y dos sumas para calcular la distancia  $d$  del píxel. El píxel será dibujado si  $d$  es inferior a un parámetro determinado que especifique el grosor de la línea.

En lugar de particionar la pantalla en píxeles, podemos asignar a cada procesador una línea de exploración o una columna de píxeles, dependiendo de la pendiente de la línea. Cada procesador calculará entonces la intersección de la línea con la fila horizontal o columna vertical de píxeles que se le han asignado. Para una línea con pendiente  $|m| < 1.0$ , cada procesador simplemente despeja el valor de  $y$  en la ecuación de la línea a partir de un valor de columna  $x$ . Para una línea con pendiente de magnitud superior a 1.0, cada procesador despeja  $x$  en la ecuación de la línea, dado un cierto valor  $y$  que especifica la línea de exploración. Estos métodos directos, aunque resultan lentos en las máquinas secuenciales, pueden realizarse de manera eficiente utilizando múltiples procesadores.

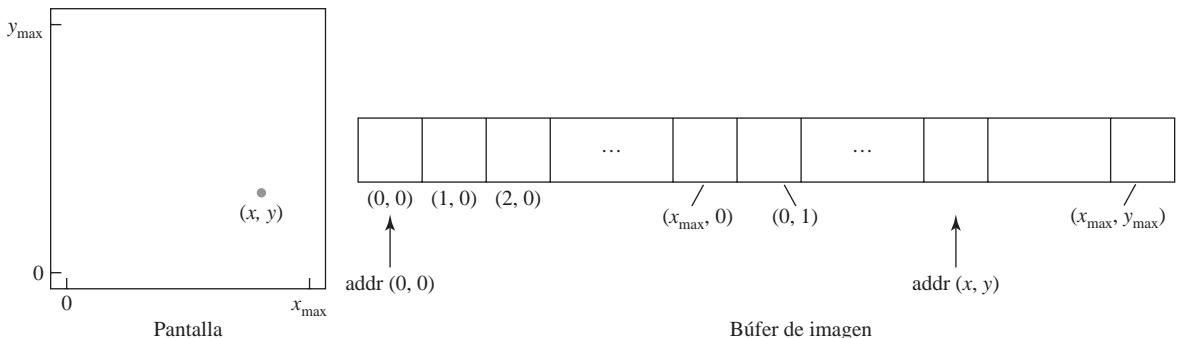
## 3.7 ALMACENAMIENTO DE LOS VALORES EN EL BÚFER DE IMAGEN

El paso final en los procedimientos de implementación relativos a los segmentos lineales y a otros objetos consiste en establecer unos valores de color en el búfer de imagen. Puesto que los algoritmos de digitalización generan posiciones de píxel a intervalos unitarios sucesivos, también se pueden utilizar operaciones incrementales para acceder de manera eficiente al búfer de imagen en cada paso del proceso de digitalización.

Como ejemplo específico, suponga que accedemos a la matriz del búfer de imagen por orden ascendente de filas y que las posiciones de píxel están etiquetadas desde  $(0, 0)$  en la esquina inferior izquierda de la pantalla hasta  $(x_{max}, y_{max})$  en la esquina superior derecha (Figura 3.14). Para un sistema monocromo (un bit por píxel), la dirección de bit en el búfer de imagen para la posición de píxel  $(x, y)$  se calcula como:

$$\text{addr}(x, y) = \text{addr}(0, 0) + y(x_{max} + 1) + x \quad (3.23)$$

Si nos movemos a lo largo de una línea de exploración, podemos calcular la dirección en el búfer de imagen para el píxel  $(x + 1, y)$  como el siguiente desplazamiento a partir de la dirección correspondiente a la posición  $(x, y)$ :



**FIGURA 3.14.** Posiciones en pantalla de los píxeles almacenados linealmente en orden ascendente de filas dentro del búfer de imagen.

$$\text{addr}(x+1, y) = \text{addr}(x, y) + 1 \quad (3.24)$$

Si nos movemos en diagonal hasta la siguiente línea de exploración a partir de  $(x, y)$ , tendremos la dirección del búfer de imagen correspondiente a  $(x + 1, y + 1)$  sin más que aplicar la fórmula:

$$\text{addr}(x+1, y+1) = \text{addr}(x, y) + x_{\max} + 2 \quad (3.25)$$

donde la constante  $x_{\max} + 2$  se precalcula una única vez para todos los segmentos de línea. Podemos obtener cálculos incrementales similares a partir de la Ecuación 3.23 para pasos unitarios en las direcciones negativas de  $x$  e  $y$ . Cada uno de los cálculos de direcciones implica únicamente una suma entera.

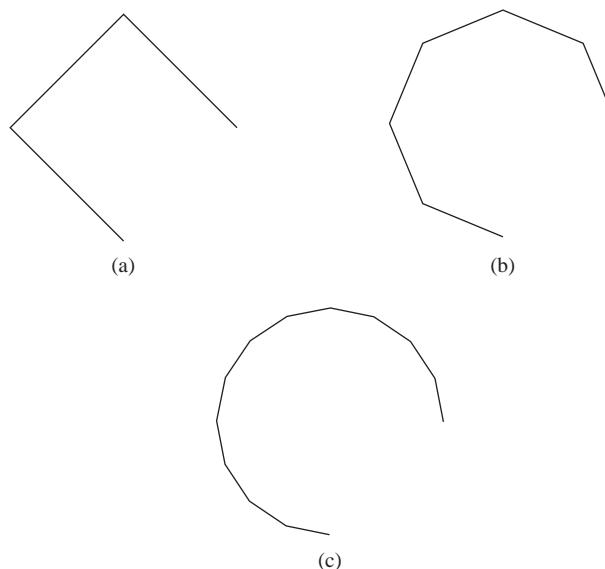
Los métodos para implementar estos procedimientos dependen de las capacidades de cada sistema concreto y de los requisitos de diseño del paquete software. En aquellos sistemas que pueden mostrar un rango de valores de intensidad para cada píxel, los cálculos de las direcciones del búfer de imagen incluirán la anchura de los píxeles (número de bits), además de la ubicación del píxel en pantalla.

## 3.8 FUNCIONES OpenGL PARA CURVAS

Las rutinas para generar curvas básicas, como círculos y elipses, no están incluidas como funciones primitivas en la biblioteca OpenGL básica. Pero esta biblioteca sí que contiene funciones para dibujar *splines* de Bézier, que son polinomios que se definen mediante un conjunto de puntos discreto. Y la utilidad OpenGL (GLU, OpenGL Utility) tiene rutinas para cuádricas tridimensionales, como esferas y cilindros, además de rutinas para generar B-splines racionales, que son una clase genérica de *splines* en la que están incluidas las curvas de Bézier, más simples. Utilizando *B-splines* racionales, podemos dibujar círculos, elipses y otras cuádricas bidimensionales. Además, se incluyen rutinas en el conjunto de herramientas de GLU (GLUT, OpenGL Utility Toolkit) que podemos utilizar para mostrar algunas cuádricas tridimensionales, como esferas y conos, y algunas otras formas geométricas. Sin embargo, todas estas rutinas son bastante más complejas que las primitivas básicas que estamos presentando en este capítulo, por lo que dejaremos el análisis de este grupo de funciones para el Capítulo 8.

Otro método que podemos utilizar para generar la gráfica de una curva simple consiste en aproximarla utilizando una polilínea. Basta con localizar un conjunto de puntos a lo largo del trayecto de la curva y conectar dichos puntos mediante segmentos de línea recta. Cuantas más secciones lineales incluyamos en la polilínea, más suave será la apariencia de la curva. Como ejemplo, la Figura 3.15 muestra varias gráficas de polilíneas que podrían utilizarse para aproximar un segmento circular.

Una tercera alternativa consiste en escribir nuestras propias funciones de generación de curvas basándonos en los algoritmos presentados en las siguientes secciones. Hablaremos primero de algunos métodos efí-



**FIGURA 3.15.** Un arco circular aproximado mediante (a) tres segmentos de línea recta, (b) seis segmentos de línea y (c) doce segmentos de línea.

cientes para la generación de círculos y elipses y luego echaremos un vistazo a los procedimientos utilizados para mostrar otras secciones cónicas, polinomios y *splines*.

## 3.9 ALGORITMOS PARA GENERACIÓN DE CÍRCULOS

---

Puesto que el círculo es un componente muy frecuentemente utilizado en dibujos y gráficas, muchos paquetes gráficos incluyen un procedimiento para generar círculos completos o arcos circulares. Asimismo, en ocasiones hay disponible una función genérica en las bibliotecas gráficas para mostrar diversos tipos de curvas, incluyendo círculos y elipses.

### Propiedades de los círculos

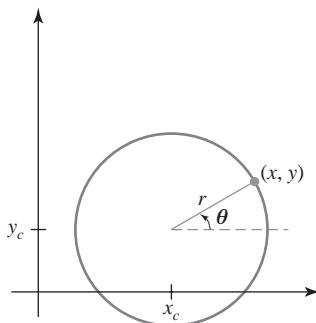
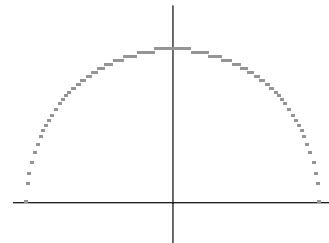
Un círculo (Figura 3.16) se define como el conjunto de puntos que se encuentran a una distancia determinada  $r$  con respecto a una posición central  $(x_c, y_c)$ . Para cualquier punto  $(x, y)$  del círculo, esta relación de distancia se expresa mediante el teorema de Pitágoras en coordenadas cartesianas:

$$(x - x_c)^2 + (y - y_c)^2 = r^2 \quad (3.26)$$

Podemos utilizar esta ecuación para calcular la posición de los puntos sobre una circunferencia, recorriendo el eje  $x$  en pasos unitarios desde  $x_c - r$  a  $x_c + r$  y calculando los correspondientes valores  $y$  en cada posición mediante la fórmula:

$$y = y_c \pm \sqrt{r^2 - (x_c - x)^2} \quad (3.27)$$

Pero este no es el mejor método para generar un círculo. Uno de los problemas con este método es que requiere unos considerables cálculos en cada paso. Además, el espaciado entre los píxeles dibujados no es uniforme, como se ilustra en la Figura 3.17. Podríamos ajustar el espaciado intercambiando  $x$  e  $y$  (recorriendo los valores  $y$  y calculando los valores  $x$ ) cuando el valor absoluto de la pendiente del círculo sea superior a 1, pero esto simplemente incrementa la cantidad de cálculos y de procesamiento requerida por el algoritmo.

FIGURA 3.16. Círculo con centro en  $(x_c, y_c)$  y radio  $r$ .FIGURA 3.17. Parte superior de un círculo dibujada mediante la Ecuación 3.27 y con  $(x_c, y_c) = (0, 0)$ .

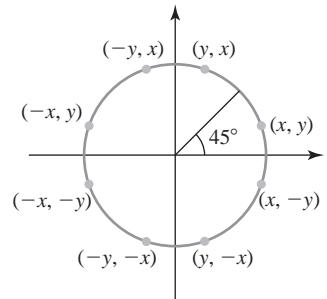
Otra forma de eliminar el espaciado desigual mostrado en la Figura 3.17 consiste en calcular los puntos de la circunferencia utilizando las coordenadas polares  $r$  y  $\theta$  (Figura 3.16). Si expresamos la ecuación de la circunferencia en forma paramétrica polar, obtenemos la pareja de ecuaciones:

$$\begin{aligned} x &= x_c + r \cos \theta \\ y &= y_c + r \sin \theta \end{aligned} \quad (3.28)$$

Cuando se genera una gráfica con estas ecuaciones utilizando un paso angular fijo, se dibujará un círculo con puntos equiespaciados a lo largo de toda la circunferencia. Para reducir el número de cálculos, podemos utilizar una gran separación angular entre los puntos de la circunferencia y conectar los puntos mediante segmentos de línea recta con el fin de aproximar la trayectoria circular. Para obtener un trazado más continuo en un monitor digital, podemos fijar como paso angular el valor  $\frac{1}{r}$ . Esto nos da posiciones de píxel que están separadas aproximadamente una unidad. Pero aunque las coordenadas polares proporcionan un espaciado homogéneo de los puntos, los cálculos trigonométricos siguen siendo bastante laboriosos.

Para cualquiera de los métodos anteriores de generación de círculos, podemos reducir los cálculos considerando la simetría que los círculos presentan. La forma del círculo es similar en cada uno de los cuadrantes. Por tanto, si determinamos las posiciones de la curva en el primer cuadrante, podemos generar la sección circular del segundo cuadrante del plano  $x$  observando que ambas secciones son simétricas con respecto al eje  $y$ . Y la secciones circulares de los cuadrantes tercero y cuarto pueden obtenerse a partir de las secciones de los dos primeros cuadrantes considerando la simetría con respecto al eje  $x$ . Podemos llevar este razonamiento un paso más allá y observar que también existe simetría entre los octantes. Las secciones circulares situadas en octantes adyacentes dentro de un mismo cuadrante son simétricas con respecto a la línea de ángulo  $45^\circ$  que divide los dos octantes. Estas condiciones de simetría se ilustran en la Figura 3.18, en la que un punto en la posición  $(x, y)$  sobre un sector de un octavo de círculo se hace corresponder con los otros siete puntos del círculo situados en los restantes octantes del plano  $x$ . Aprovechando la simetría del círculo de esta forma, podemos generar todas las posiciones de píxel alrededor del círculo calculando únicamente los puntos correspondientes al sector que va desde  $x = 0$  a  $x = y$ . La pendiente de la curva en este octante tiene una magnitud igual o inferior a 1.0. Para  $x = 0$ , la pendiente del círculo es 0 y para  $x = y$ , la pendiente es  $-1,0$ .

Determinar las posiciones de los píxeles sobre una circunferencia utilizando las consideraciones de simetría y la Ecuación 3.26 o la Ecuación 3.28 sigue requiriendo una gran cantidad de cálculos. La ecuación cartesiana 3.26 implica realizar multiplicaciones y raíces cuadradas, mientras que las ecuaciones paramétricas contienen multiplicaciones y cálculos trigonométricos. Otros algoritmos de generación de circunferencias más eficientes se basan en cálculos incrementales de parámetros de decisión, como en el algoritmo de Bresenham para líneas; estos cálculos basados en parámetros de decisión sólo implican realizar operaciones simples con enteros.



**FIGURA 3.18.** Simetría de un círculo. El cálculo de un punto  $(x, y)$  del círculo en uno de los octantes nos da los puntos del círculo que se muestran para los otros siete octantes.

Podemos adaptar el algoritmo de dibujo de líneas de Bresenham a la generación de círculos definiendo los parámetros de decisión para hallar el píxel más cercano a la circunferencia en cada paso de muestreo. Sin embargo, la Ecuación 3.26 del círculo es no lineal, por lo que haría falta calcular raíces cuadradas para hallar las distancias de los píxeles con respecto a la trayectoria circular. El algoritmo de Bresenham para círculos evita estos cálculos de raíces cuadradas comparando los cuadrados de las distancias de separación de los píxeles.

Sin embargo, se puede realizar una comparación directa de distancias sin necesidad de hallar raíces cuadradas. La idea básica que subyace a este método consiste en comprobar si el punto medio situado entre dos píxeles está situado dentro o fuera del círculo. Este método se puede, asimismo, generalizar más fácilmente a otras cónicas y para un círculo de radio entero, esta técnica del punto medio genera las mismas posiciones de píxel que el algoritmo de Bresenham para círculos. Para un segmento de línea recta, el método del punto medio es completamente equivalente al algoritmo de Bresenham para líneas. Asimismo, el error máximo a la hora de determinar las posiciones de los píxeles para cualquier sección cónica utilizando el test del punto medio está limitado a un medio de la separación entre píxeles.

### Algoritmo del punto medio para círculos

Como en el ejemplo de digitalización de líneas, muestreamos a intervalos unitarios y determinados la posición de píxel más próxima a la trayectoria circular especificada. Para un radio  $r$  dado y unas coordenadas del centro de valor  $(x_c, y_c)$ , podemos primero ejecutar el algoritmo para calcular las posiciones de los píxeles alrededor de una trayectoria circular centrada en el origen de coordenadas  $(0, 0)$ . Después, movemos cada posición calculada  $(x, y)$  a la posición de pantalla adecuada sumando  $x_c$  a  $x$  e  $y_c$  a  $y$ . A lo largo de la sección circular que va desde  $x = 0$  a  $x = y$  en el primer cuadrante, la pendiente de la curva varía desde 0 a  $-1.0$ . Por tanto, podemos tomar pasos unitarios en la dirección  $x$  positiva a lo largo de este octante y utilizar un parámetro de decisión para determinar cuál de las dos posibles posiciones de píxel en cada columna está más cerca verticalmente a la trayectoria circular. Las posiciones en los otros siete octantes se obtienen entonces por simetría.

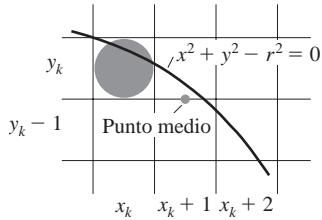
Para aplicar el método del punto medio, definimos una función circular como:

$$f_{\text{circ}}(x, y) = x^2 + y^2 - r^2 \quad (3.29)$$

Cualquier punto  $(x, y)$  en la frontera del círculo de radio  $r$  satisfará la ecuación  $f_{\text{circ}}(x, y) = 0$ . Si el punto se encuentra en el interior del círculo, la función tomará un valor negativo, mientras que si el punto se encuentra fuera del círculo, el valor de la función será positivo. En resumen, la posición relativa de cualquier punto  $(x, y)$  puede determinarse comprobando el signo de la función generadora del círculo:

$$f_{\text{circ}}(x, y) \begin{cases} < 0, & \text{si } (x, y) \text{ se encuentra dentro del círculo} \\ = 0, & \text{si } (x, y) \text{ se encuentra sobre la circunferencia} \\ > 0, & \text{si } (x, y) \text{ se encuentra fuera del círculo} \end{cases} \quad (3.30)$$

Las comprobaciones de la Ecuación 3.30 se realizan para los puntos intermedios situados en la vecindad de la trayectoria circular en cada paso de muestreo. Así, la función generadora del círculo es un parámetro de



**FIGURA 3.19.** Punto medio entre los dos píxeles candidatos para la posición de muestreo  $x_k + 1$  a lo largo de una trayectoria circular.

decisión en el algoritmo del punto medio, y podemos determinar los cálculos incrementales necesarios para esta función, como hicimos con el algoritmo de generación de líneas.

La Figura 3.19 muestra el punto medio entre los dos píxeles candidatos para la posición de muestreo  $x_k + 1$ . Suponiendo que acabemos de dibujar el píxel  $(x_k, y_k)$ , necesitaremos a continuación determinar si el píxel en la posición  $(x_k + 1, y_k)$  se encuentra más cerca o más lejos del círculo que el situado en la posición  $(x_k + 1, y_k - 1)$ . Nuestro parámetro de decisión será la Ecuación 3.29 de generación del círculo, evaluado en el punto medio entre estos dos píxeles:

$$\begin{aligned} p_k &= f_{\text{circ}} \left( x_k + 1, y_k - \frac{1}{2} \right) \\ &= (x_k + 1)^2 + \left( y_k - \frac{1}{2} \right)^2 - r^2 \end{aligned} \quad (3.31)$$

Si  $p_k < 0$ , este punto medio se encontrará en el interior del círculo y el píxel situado en la línea de exploración  $y_k$  estará más cerca de la frontera del círculo. En caso contrario, el punto intermedio se encuentra fuera del círculo o sobre la misma circunferencia, y seleccionaremos el píxel correspondiente a la línea de exploración  $y_k - 1$ .

Los sucesivos parámetros de decisión se obtienen utilizando cálculos incrementales. Podemos obtener una fórmula recursiva para el siguiente parámetro de decisión evaluando la función circular en la posición de muestreo  $x_k + 1 = x_k + 2$ :

$$\begin{aligned} p_{k+1} &= f_{\text{circ}} \left( x_{k+1} + 1, y_{k+1} - \frac{1}{2} \right) \\ &= [(x_k + 1) + 1]^2 + \left( y_{k+1} - \frac{1}{2} \right)^2 - r^2 \end{aligned}$$

o

$$p_{k+1} = p_k + 2(x_k + 1) + (y_{k+1}^2 - y_k^2) - (y_{k+1} - y_k) + 1 \quad (3.32)$$

donde  $y_{k+1}$  es  $y_k$  o  $y_k - 1$ , dependiendo del signo de  $p_k$ .

Los incrementos para obtener  $p_{k+1}$  son  $2x_{k+1} + 1$  (si  $p_k$  es negativo) o  $2x_{k+1} + 1 - 2y_{k+1}$ . La evaluación de los términos  $2x_{k+1}$  y  $2y_{k+1}$  también puede hacerse incrementalmente mediante las fórmulas:

$$2x_{k+1} = 2x_k + 2$$

$$2y_{k+1} = 2y_k - 2$$

En la posición inicial  $(0, r)$ , estos dos términos tienen los valores 0 y  $2r$ , respectivamente. Cada valor sucesivo para el término  $2x_{k+1}$  se obtiene sumando 2 al valor anterior y cada valor sucesivo del término  $2y_{k+1}$  se obtiene restando 2 al valor anterior.

El parámetro de decisión inicial se obtiene evaluando la función de generación del círculo en la posición inicial  $(x_0, y_0) = (0, r)$ :

$$\begin{aligned}
 p_0 &= f_{\text{circ}}\left(1, r - \frac{1}{2}\right) \\
 &= 1 + \left(r - \frac{1}{2}\right)^2 - r^2 \\
 &= \frac{5}{4} - r
 \end{aligned} \tag{3.33}$$

o

Si el radio  $r$  está especificado como un valor entero, podemos simplemente redondear  $p_0$  de la forma siguiente:

$$p_0 = 1 - r \quad (\text{para } r \text{ entero})$$

dado que todos los incrementos son enteros.

Como en el algoritmo de líneas de Bresenham, el método del punto medio calcula las posiciones de los píxeles a lo largo de la circunferencia utilizando sumas y restas enteras, suponiendo que los parámetros del círculo estén especificados en coordenadas enteras de pantalla. Podemos resumir los pasos del algoritmo del punto medio para generación de círculos de la forma siguiente.

### Algoritmo del punto medio para generación de círculos

1. Introducir el radio  $r$  y el centro del círculo  $(x_c, y_c)$  y luego establecer las coordenadas para el primer punto de la circunferencia de un círculo centrado en el origen mediante la fórmula:

$$(x_c, y_c) = (0, r)$$

2. Calcular el valor inicial del parámetro de decisión como

$$p_0 = \frac{5}{4} - r$$

3. Para cada posición  $x_k$ , comenzando en  $k = 0$ , realizar la siguiente comprobación. Si  $p_k < 0$ , el siguiente punto a lo largo de un círculo centrado en  $(0,0)$  será  $(x_{k+1}, y_k)$  y,

$$p_{k+1} = p_k + 2x_{k+1} + 1$$

En caso contrario, el siguiente punto del círculo será  $(x_k + 1, y_k - 1)$  y,

$$p_{k+1} = p_k + 2x_{k+1} + 1 - 2y_{k+1}$$

donde  $2x_{k+1} = 2x_k$  y  $2y_{k+1} = 2y_k - 2$ .

4. Determinar los puntos simétricos en los otros siete octantes.
5. Mover cada posición de píxel  $(x, y)$  calculada hasta la trayectoria circular centrada en  $(x_c, y_c)$  y dibujar los valores de coordenadas:

$$x = x + x_c, \quad y = y + y_c$$

6. Repetir los Pasos 3 a 5 hasta que  $x \geq y$ .

**Ejemplo 3.2** Dibujo de un círculo mediante el algoritmo del punto medio.

Dado un círculo de radio  $r = 10$ , vamos a ilustrar el algoritmo del punto medio para generación de círculos determinando las posiciones a lo largo del octante del círculo situado en el primer cuadrante, entre  $x = 0$  y  $x = y$ . El valor inicial del parámetro de decisión es:

$$p_0 = 1 - r = -9$$

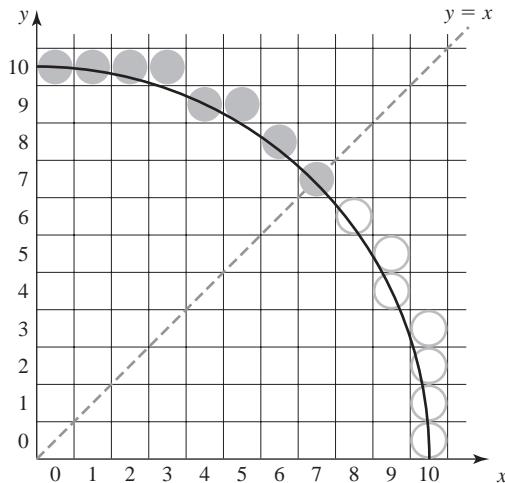
Para el círculo centrado en el origen de coordenadas, el punto inicial es  $(x_0, y_0) = (0, 10)$  y los términos de incremento iniciales para el cálculo de los parámetros de decisión son:

$$2x_0 = 0, \quad 2y_0 = 20$$

En la siguiente tabla se enumeran los valores sucesivos del parámetro de decisión del punto medio y las correspondientes coordenadas a lo largo de la circunferencia.

$k$	$p_k$	$(x_{k+1}, y_{k+1})$	$2x_{k+1}$	$2y_{k+1}$
0	-9	(1, 10)	2	20
1	-6	(2, 10)	4	20
2	-1	(3, 10)	6	20
3	6	(4, 9)	8	18
4	-3	(5, 9)	10	18
5	8	(6, 8)	12	16
6	5	(7, 7)	14	14

En la Figura 3.20 se muestra una gráfica de las posiciones de píxel generadas en el primer cuadrante.



**FIGURA 3.20.** Posiciones de los círculos (círculos rellenos) a lo largo de un arco circular centrado en el origen y con radio  $r = 10$ , calculadas mediante el algoritmo del punto medio para generación de círculos. Los círculos abiertos («huecos») muestran las posiciones simétricas en el primer cuadrante.

El siguiente segmento de código ilustra los procedimientos que podrían emplearse para implementar el algoritmo del punto medio para generación de círculos. Al procedimiento `circleMidpoint` hay que pasarle el valor del radio del círculo y las coordenadas del centro del círculo. Entonces, se calcula una posición de píxel dentro del primer octante de la trayectoria circular y se pasa dicha posición al procedimiento `circlePlotPoints`. Este procedimiento almacena el color correspondiente al círculo en el búfer de imagen para todas las posiciones simétricas dentro del círculo, efectuando llamadas repetidas a las rutina `setPixel`, que está implementada con las funciones de dibujo de puntos de OpenGL.

```
#include <GL/glut.h>

class screenPt
{
private:
    GLint x, y;

public:
    /* Constructor predeterminado: inicializa las coordenadas a (0, 0). */
    screenPt ( ) {
        x = y = 0;
    }
    void setCoords (GLint xCoordValue, GLint yCoordValue) {
        x = xCoordValue;
        y = yCoordValue;
    }
    GLint getx ( ) const {
        return x;
    }
    GLint gety ( ) const {
        return y;
    }
    void incrementx ( ) {
        x++;
    }
    void decremente ( ) {
        y--;
    }
};

void setPixel (GLint xCoord, GLint yCoord)
{
    glBegin (GL_POINTS);
    glVertex2i (xCoord, yCoord);
    glEnd ( );
}

void circleMidpoint (GLint xc, GLint yc, GLint radius)
{
    screenPt circPt;
```

```

GLint p = 1 - radius;      // Valor inicial para el parámetro de punto medio.

circPt.setCoords (0, radius); // Establecer coordenadas para
                             // punto superior del círculo.

void circlePlotPoints (GLint, GLint, screenPt);

/* Dibujar el punto inicial en cada cuadrante del círculo. */
circlePlotPoints (xc, yc, circPt);
/* Calcular el siguiente punto y dibujarlo en cada octante. */
while (circPt.getx ( ) < circPt.gety ( )) {
    circPt.incrementx ( );
    if (p < 0)
        p += 2 * circPt.getx ( ) + 1;
    else {
        circPt.decrementsy ( );
        p += 2 * (circPt.getx ( ) - circPt.gety ( )) + 1;
    }
    circlePlotPoints (xc, yc, circPt);
}
}

void circlePlotPoints (GLint xc, GLint yc, screenPt circPt)
{
    setPixel (xc + circPt.getx ( ), yc + circPt.gety ( ));
    setPixel (xc - circPt.getx ( ), yc + circPt.gety ( ));
    setPixel (xc + circPt.getx ( ), yc - circPt.gety ( ));
    setPixel (xc - circPt.getx ( ), yc - circPt.gety ( ));
    setPixel (xc + circPt.gety ( ), yc + circPt.getx ( ));
    setPixel (xc - circPt.gety ( ), yc + circPt.getx ( ));
    setPixel (xc + circPt.gety ( ), yc - circPt.getx ( ));
    setPixel (xc - circPt.gety ( ), yc - circPt.getx ( ));
}

```

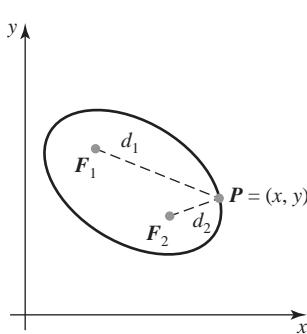
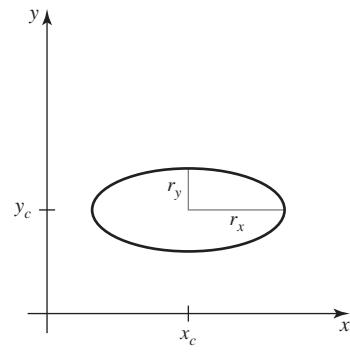
## 3.10 ALGORITMOS DE GENERACIÓN DE ELIPSSES

---

En términos simples, una elipse es un círculo alargado. También podemos escribir una elipse como un círculo modificado cuyo radio varía desde un valor máximo en una dirección hasta un valor mínimo en la dirección perpendicular. Los segmentos de línea recta trazados en el interior de la elipse en estas dos direcciones perpendiculares se denominan eje mayor y menor de la elipse.

### Propiedades de las elipses

Puede darse una definición precisa de una elipse en términos de la distancia desde cualquier punto de la elipse a dos posiciones fijas, denominadas focos de la elipse. La suma de estas dos distancias es constante para todos los puntos de la elipse (Figura 3.21). Si etiquetamos como  $d_1$  y  $d_2$  las distancias a los dos focos desde cualquier punto  $\mathbf{P} = (x, y)$  de la elipse, la ecuación general de la elipse puede escribirse:

FIGURA 3.21. Elipse generada con los focos  $F_1$  y  $F_2$ .FIGURA 3.22. Elipse centrada en  $(x_c, y_c)$  con semieje mayor  $r_x$  y semieje menor  $r_y$ .

$$d_1 + d_2 = \text{constante} \quad (3.34)$$

Expresando las distancias  $d_1$  y  $d_2$  en términos de las coordenadas de los focos  $F_1 = (x_1, y_1)$  y  $F_2 = (x_2, y_2)$ , tendremos:

$$\sqrt{(x - x_1)^2 + (y - y_1)^2} + \sqrt{(x - x_2)^2 + (y - y_2)^2} = \text{constante} \quad (3.35)$$

Elevando esta ecuación al cuadrado, despejando la raíz cuadrada restante y volviendo a elevar al cuadrado, podremos reescribir la ecuación general de la elipse de la forma

$$Ax^2 + By^2 + Cxy + Dx + Ey + F = 0 \quad (3.36)$$

donde los coeficientes  $A, B, C, D, E$  y  $F$  se evalúan en términos de las coordenadas de los focos y de las dimensiones de los ejes mayor y menor de la elipse. El eje mayor es el segmento de línea recta que se extiende desde un lado de la elipse hasta el otro a través de los focos. El eje menor abarca la dimensión más pequeña de la elipse, bisecando perpendicularmente el eje mayor en su punto medio (centro de la elipse) situado entre los dos focos.

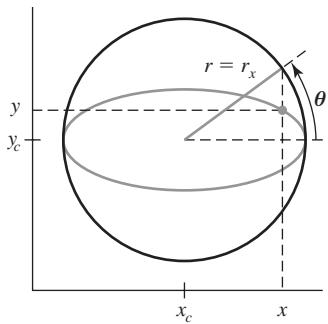
Un método interactivo para especificar una elipse con una orientación arbitraria consiste en introducir los dos focos y un punto de la elipse. Con estos tres conjuntos de coordenadas, podemos evaluar la constante de la Ecuación 3.35. Entonces, se pueden calcular los valores de los coeficientes de la Ecuación 3.36 y utilizarlos para generar los píxeles a lo largo de la trayectoria elíptica.

Las ecuaciones de la elipse se pueden simplificar enormemente si se alinean los ejes mayor y menor con los ejes de coordenadas. En la Figura 3.22 se muestra una elipse en “posición estándar”, con los ejes mayor y menor orientados en paralelo a los ejes  $x$  e  $y$ . El parámetro  $r_x$  de este ejemplo indica el semieje mayor, mientras que el parámetro  $r_y$  indica el semieje menor. La ecuación de la elipse mostrada en la Figura 3.22 puede escribirse en términos de las coordenadas del centro de la elipse y de los parámetros  $r_x$  y  $r_y$ , de la forma siguiente:

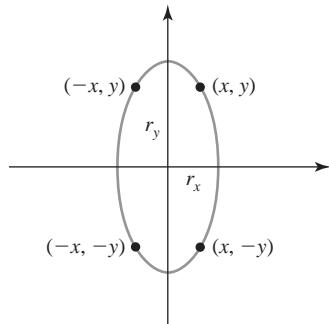
$$\left( \frac{x - x_c}{r_x} \right)^2 + \left( \frac{y - y_c}{r_y} \right)^2 = 1 \quad (3.37)$$

Utilizando las coordenadas polares  $r$  y  $\theta$ , podemos también describir la elipse en su posición estándar con las ecuaciones paramétricas:

$$\begin{aligned} x &= x_c + r_x \cos \theta \\ y &= y_c + r_y \sin \theta \end{aligned} \quad (3.38)$$



**FIGURA 3.23.** El círculo circunscrito y el ángulo de excentricidad  $\theta$  para una elipse con  $r_x > r_y$ .



**FIGURA 3.24.** Simetría de una elipse. El cálculo de un punto  $(x, y)$  en un cuadrante nos da los puntos de la elipse que se muestran para los otros tres cuadrantes.

El ángulo  $\theta$ , denominado *ángulo de excentricidad* de la elipse, se mide a lo largo del perímetro de un círculo circunscrito. Si  $r_x > r_y$ , el radio del círculo circunscrito es  $r = r_x$  (Figura 3.23). En caso contrario, el círculo circunscrito tiene como radio  $r = r_y$ .

Al igual que con el algoritmo del círculo, pueden utilizarse consideraciones de simetría para reducir los cálculos. Una elipse en posición estándar presenta simetría entre los distintos cuadrantes pero, a diferencia del círculo, los dos octantes de cada cuadrante no son simétricos. Por tanto, deberemos calcular las posiciones de los píxeles a lo largo del arco elíptico que recorre un cuadrante y luego utilizar las consideraciones de simetría para obtener las posiciones de la curva en los tres cuadrantes restantes (Figura 3.24).

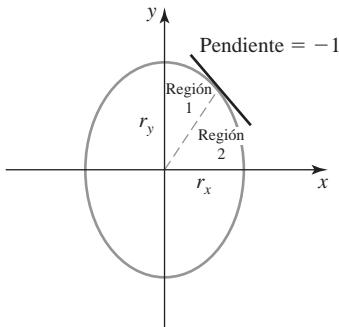
### Algoritmo del punto medio para la elipse

La técnica que vamos a utilizar es similar a la que hemos empleado para visualizar el círculo digitalizado. Dados los parámetros  $r_x$ ,  $r_y$  y  $(x_c, y_c)$ , determinamos las posiciones  $(x, y)$  de la curva para una elipse en posición estándar centrada en el origen y luego desplazamos todos los puntos utilizando un desplazamiento constante, de modo que la elipse está centrada en  $(x_c, y_c)$ . Si queremos también mostrar la elipse en posición no estándar, podemos rotarla alrededor de su centro con el fin de reorientar los ejes mayor y menor en las direcciones deseadas. Pero por el momento, vamos a considerar únicamente la visualización de elipses en posición estándar. Hablaremos de los métodos generales para transformar las orientaciones y posiciones de los objetos en el Capítulo 5.

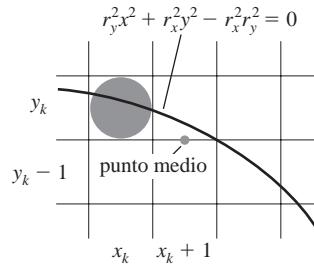
El método del punto medio para la elipse se aplica en dos partes para todo el primer cuadrante. La Figura 3.25 muestra la división del primer cuadrante de acuerdo con la pendiente de una elipse con  $r_x < r_y$ . Procesamos este cuadrante tomando pasos unitarios en la dirección  $x$  allí donde la pendiente de la curva tenga una magnitud inferior a 1.0 y luego tomando pasos unitarios en la dirección  $y$  cuando la pendiente tenga una magnitud superior a 1.0.

Las regiones 1 y 2 (Figura 3.25) pueden procesarse de diversas formas. Podemos empezar en la posición  $(0, r_y)$  y avanzar en el sentido de las agujas del reloj a lo largo del primer cuadrante de la trayectoria elíptica, pasando de utilizar pasos unitarios según  $x$  a pasos unitarios según  $y$  cuando la pendiente sea inferior a  $-1.0$ . Alternativamente, podríamos empezar en  $(r_x, 0)$  y seleccionar los puntos en sentido contrario a las agujas del reloj, pasando de utilizar pasos unitarios según  $y$  a pasos unitarios según  $x$  cuando la pendiente sea superior a  $-1.0$ . Si tuviéramos varios procesadores trabajando en paralelo, podríamos calcular las posiciones de los píxeles en ambas regiones simultáneamente. Como ejemplo de implementación secuencial del algoritmo del punto medio, vamos a tomar como posición inicial  $(0, r_y)$  y a recorrer la trayectoria de la elipse en el sentido de las agujas del reloj para todo el primer cuadrante.

Podemos definir una función de la elipse a partir de la Ecuación 3.37 con  $(x_c, y_c) = (0, 0)$ , de la forma siguiente:



**FIGURA 3.25.** Regiones de procesamiento para la elipse. En la región 1, la magnitud de la pendiente de la elipse es inferior a 1.0; en la región 2, la magnitud de la pendiente es superior a 1.0.



**FIGURA 3.26.** Punto medio entre los píxeles candidatos para la posición de muestreo  $x_k + 1$  a lo largo de una trayectoria elíptica.

$$f_{\text{elipse}}(x, y) = r_y^2 x^2 + r_x^2 y^2 - r_x^2 r_y^2 \quad (3.39)$$

que tiene las siguientes propiedades:

$$f_{\text{elipse}}(x, y) \begin{cases} < 0, & \text{si } (x, y) \text{ está dentro de la elipse} \\ = 0, & \text{si } (x, y) \text{ está sobre la elipse} \\ > 0, & \text{si } (x, y) \text{ está fuera de la elipse} \end{cases} \quad (3.40)$$

Así, la función de la elipse  $f_{\text{elipse}}(x, y)$  se puede utilizar como parámetro de decisión para el algoritmo del punto medio. En cada posición de muestreo, seleccionamos el siguiente píxel de la trayectoria elíptica de acuerdo con el signo de la función de la elipse, evaluado en el punto medio entre los dos píxeles candidatos.

Comenzando en  $(0, r_y)$ , tomamos pasos unitarios en la dirección  $x$  hasta que alcanzamos la frontera entre las regiones 1 y 2 (Figura 3.25). Después, pasamos a utilizar pasos unitarios en la dirección  $y$  para el resto de la curva dentro del primer cuadrante. En cada paso, necesitamos comprobar el valor de la pendiente de la curva. La pendiente de la elipse se calcula a partir de la Ecuación 3.39:

$$\frac{dy}{dx} = -\frac{2r_y^2 x}{2r_x^2 y} \quad (3.41)$$

En la frontera entre la región 1 y la región 2,  $dy/dx = -1.0$  y,

$$2r_y^2 x = 2r_x^2 y$$

Por tanto, habremos salido de la región 1 cuando:

$$2r_y^2 x \geq 2r_x^2 y \quad (3.42)$$

La Figura 3.26 muestra el punto medio entre los dos píxeles candidatos en la posición de muestreo  $x_k + 1$ , dentro de la primera región. Suponiendo que hayamos seleccionado la posición  $(x_k, y_k)$  en el paso anterior, determinamos la siguiente posición a lo largo de la trayectoria elíptica evaluando el parámetro de decisión (es decir, la función de la elipse dada en la Ecuación 3.39) en dicho punto intermedio:

$$p1_k = f_{\text{elipse}}\left(x_k + 1, y_k - \frac{1}{2}\right) = r_y^2 (x_k + 1)^2 + r_x^2 \left(y_k - \frac{1}{2}\right)^2 - r_x^2 r_y^2 \quad (3.43)$$

Si  $p1_k < 0$ , el punto medio estará dentro de la elipse y el píxel de la línea de exploración  $y_k$  estará más próximo a la frontera de la elipse. En caso contrario, el punto medio está fuera de la elipse o sobre ella y seleccionaremos el píxel situado en la línea de exploración  $y_k - 1$ .

En la siguiente posición de muestreo ( $x_{k+1} + 1 = x_k + 2$ ), el parámetro de decisión para la región 1 se evalúa como:

$$\begin{aligned} p1_{k+1} &= f_{\text{elipse}} \left( x_{k+1} + 1, y_{k+1} - \frac{1}{2} \right) \\ &= r_y^2 [(x_k + 1) + 1]^2 + r_x^2 \left( y_{k+1} - \frac{1}{2} \right)^2 - r_x^2 r_y^2 \end{aligned}$$

o

$$p1_{k+1} = p1_k + 2r_y^2(x_k + 1) + r_y^2 + r_x^2 \left[ \left( y_{k+1} - \frac{1}{2} \right)^2 - \left( y_k - \frac{1}{2} \right)^2 \right] \quad (3.44)$$

donde  $y_{k+1}$  puede ser  $y_k$  o  $y_k - 1$ , dependiendo del signo de  $p1_k$ .

Los parámetros de decisión se incrementan de la forma siguiente:

$$\text{incremento} = \begin{cases} 2r_y^2 x_{k+1} + r_y^2, & \text{si } p1_k < 0 \\ 2r_y^2 x_{k+1} + r_y^2 - 2r_x^2 y_{k+1}, & \text{si } p1_k \geq 0 \end{cases}$$

Los incrementos para los parámetros de decisión pueden calcularse utilizando únicamente sumas y restas, como en el algoritmo de los círculos, ya que los valores para los términos  $2r_y^2 x$  y  $2r_x^2 y$  pueden obtenerse incrementalmente. En la posición inicial  $(0, r_y)$ , estos dos términos tienen como valor:

$$2r_y^2 x = 0 \quad (3.45)$$

$$2r_x^2 y = 2r_x^2 r_y \quad (3.46)$$

A medida que se incrementan  $x$  e  $y$ , los valores actualizados se obtienen sumando  $2r_y^2$  al valor actual del término de incremento de la Ecuación 3.45 y restando  $2r_x^2$  del valor actual del término de incremento de la Ecuación 3.46. Los valores de incremento actualizados se comparan en cada caso y nos moveremos de la región 1 a la región 2 cuando se satisfaga la condición 3.42.

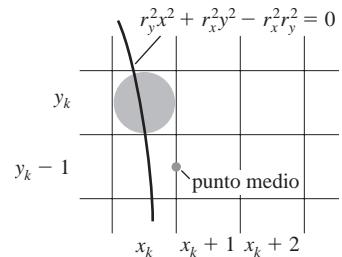
En la región 1, el valor inicial del parámetro de decisión se obtiene evaluando la función de la elipse en la posición inicial  $(x_0, y_0) = (0, r_y)$ :

$$\begin{aligned} p1_0 &= f_{\text{elipse}} \left( 1, r_y - \frac{1}{2} \right) \\ &= r_y^2 + r_x^2 \left( r_y - \frac{1}{2} \right)^2 - r_x^2 r_y^2 \end{aligned}$$

o

$$p1_0 = r_y^2 - r_x^2 r_y + \frac{1}{4} r_x^2 \quad (3.47)$$

En la región 2, muestreamos a intervalos unitarios en la dirección  $y$  negativa y el punto medio se tomará ahora entre píxeles horizontales para cada paso de muestreo (Figura 3.27). Para esta región, el parámetro de decisión se evalúa como:



**FIGURA 3.27.** Punto medio entre píxeles candidatos en la posición de muestreo  $y_k - 1$  a lo largo de una trayectoria elíptica.

$$\begin{aligned}
 p2_k &= f_{\text{ellipse}}\left(x_k + \frac{1}{2}, y_k - 1\right) \\
 &= r_y^2 \left(x_k + \frac{1}{2}\right)^2 + r_x^2 (y_k - 1)^2 - r_x^2 r_y^2
 \end{aligned} \tag{3.48}$$

Si  $p2_k > 0$ , el punto medio se encontrará fuera de la elipse y seleccionaremos el píxel correspondiente a  $x_k$ . Si  $p2_k \leq 0$ , el punto medio estará sobre la elipse o dentro de ella y seleccionaremos la posición de píxel  $x_{k+1}$ .

Para determinar la relación entre parámetros de decisión sucesivos dentro de la región 2, evaluamos la función de la elipse en el siguiente punto de muestreo  $y_{k+1} - 1 = y_k - 2$ :

$$\begin{aligned}
 p2_{k+1} &= f_{\text{ellipse}}\left(x_{k+1} + \frac{1}{2}, y_{k+1} - 1\right) \\
 &= r_y^2 \left(x_{k+1} + \frac{1}{2}\right)^2 + r_x^2 [(y_k - 1) - 1]^2 - r_x^2 r_y^2
 \end{aligned} \tag{3.49}$$

o

$$p2_{k+1} = p2_k - 2r_x^2(y_k - 1) + r_x^2 + r_y^2 \left[ \left(x_{k+1} + \frac{1}{2}\right)^2 - \left(x_k + \frac{1}{2}\right)^2 \right] \tag{3.50}$$

donde  $x_{k+1}$  vale  $x_k$  o  $x_k + 1$ , dependiendo del signo de  $p2_k$ .

Cuando entramos en la región 2, se toma como posición inicial  $(x_0, y_0)$  la última posición seleccionada en la región 1, y el parámetro de decisión inicial en la región 2 será entonces:

$$\begin{aligned}
 p2_0 &= f_{\text{ellipse}}\left(x_0 + \frac{1}{2}, y_0 - 1\right) \\
 &= r_y^2 \left(x_0 + \frac{1}{2}\right)^2 + r_x^2 (y_0 - 1)^2 - r_x^2 r_y^2
 \end{aligned} \tag{3.51}$$

Para simplificar el cálculo de  $p2_0$ , podemos seleccionar las posiciones de los píxeles en sentido contrario a las agujas del reloj, comenzando en  $(r_x, 0)$ . Los pasos unitarios se tomarían entonces en la dirección  $y$  positiva, hasta alcanzar la última posición seleccionada en la región 1.

El algoritmo de punto medio puede adaptarse para generar una elipse en posición no estándar, utilizando la función de la elipse dada por la Ecuación 3.36 y calculando las posiciones de los píxeles a lo largo de toda la trayectoria elíptica. Alternativamente, podemos reorientar los ejes de la elipse para ponerlos en posición estándar, utilizando los métodos de transformación explicados en el Capítulo 5, después de lo cual se apli-

ría el algoritmo del punto medio para elipses con el fin de determinar las posiciones de la curva; finalmente, las posiciones de píxel calculadas se convertirían para obtener las posiciones correspondientes según la orientación original de la elipse.

Suponiendo que nos den  $r_x$ ,  $r_y$  y el centro de la elipse en coordenadas de pantalla enteras, sólo hacen falta cálculos incrementales enteros para determinar los valores de los parámetros de decisión en el algoritmo del punto medio para generación de elipses. Los incrementos  $r_x^2$ ,  $r_y^2$ ,  $2r_x^2$  y  $2r_y^2$  se evalúan una única vez al principio del procedimiento. En el siguiente resumen, se enumeran los pasos para dibujar una elipse utilizando el algoritmo del punto medio.

### Algoritmo del punto medio para generación de una elipse

1. Introducir  $r_x$ ,  $r_y$  y el centro de la elipse  $(x_c, y_c)$  y obtener el primer punto sobre una elipse centrada en el origen, de la forma siguiente:

$$(x_0, y_0) = (0, r_y)$$

2. Calcular el valor inicial del parámetro de decisión en la región 1 mediante la fórmula

$$p1_0 = r_y^2 - r_x^2 r_y + \frac{1}{4} r_x^2$$

3. En cada posición  $x_k$  dentro de la región 1, comenzando en  $k = 0$ , realizar la siguiente comprobación. Si  $p1_k < 0$ , el siguiente punto a lo largo de la elipse centrada en  $(0, 0)$  es  $(x_{k+1}, y_k)$  y,

$$p1_{k+1} = p1_k + 2r_y^2 x_{k+1} + r_y^2$$

En caso contrario, el siguiente punto a lo largo de la elipse será  $(x_k + 1, y_k - 1)$  y,

$$p1_{k+1} = p1_k + 2r_y^2 x_{k+1} - 2r_x^2 y_{k+1} + r_y^2$$

con

$$2r_y^2 x_{k+1} = 2r_y^2 x_k + 2r_y^2, \quad 2r_x^2 y_{k+1} = 2r_x^2 y_k + 2r_x^2$$

debiendo continuar este proceso hasta que  $2r_y^2 x \geq 2r_x^2 y$ .

4. Calcular el valor inicial del parámetro de decisión en la región 2 mediante la fórmula:

$$p2_0 = r_y^2 \left( x_0 + \frac{1}{2} \right)^2 + r_x^2 (y_0 - 1)^2 - r_x^2 r_y^2$$

donde  $(x_0, y_0)$  es la última posición calculada para la región 1.

5. En cada posición  $y_k$  de la región 2, comenzando en  $k = 0$ , realizar la siguiente comprobación. Si  $p2_k > 0$ , el siguiente punto a lo largo de la elipse centrada en  $(0, 0)$  será  $(x_k, y_k - 1)$  y,

$$p2_{k+1} = p2_k - 2r_x^2 y_{k+1} + r_x^2$$

En caso contrario, el siguiente punto a lo largo de la elipse será  $(x_k + 1, y_k - 1)$  y,

$$p2_{k+1} = p2_k + 2r_y^2 x_{k+1} - 2r_x^2 y_{k+1} + r_y^2$$

utilizando los mismos cálculos incrementales para  $x$  e  $y$  que en la región 1. Este proceso debe continuar hasta que  $y = 0$ .

6. Para ambas regiones, determinar los puntos simétricos en los otros tres cuadrantes.
7. Mover cada posición de píxel  $(x, y)$  calculada a la trayectoria elíptica centrada en  $(x_c, y_c)$  y dibujar los valores de coordenadas:

$$x = x + x_c, \quad y = y + y_c$$

### Ejemplo 3.3 Dibujo de una elipse mediante el algoritmo del punto medio

Dados los parámetros de entrada para la elipse  $r_x = 8$  y  $r_y = 6$ , vamos a ilustrar los pasos del algoritmo del punto medio para el cálculo de la elipse determinando las posiciones digitalizadas a lo largo del trayecto elíptico en el primer cuadrante. Los valores e incrementos iniciales para los cálculos del parámetro de decisión son:

$$2r_y^2 x = 0 \quad (\text{con incremento } 2r_y^2 = 72)$$

$$2r_x^2 y = 2r_x^2 r_y \quad (\text{con incremento } -2r_x^2 = -128)$$

Para la región 1, el punto inicial para la elipse centrada en el origen será  $(x_0, y_0) = (0, 6)$  y el valor inicial del parámetro de decisión será:

$$p1_0 = r_y^2 - r_x^2 r_y + \frac{1}{4} r_x^2 = -332$$

La tabla siguiente muestra los valores sucesivos del parámetro de decisión para el punto medio y las posiciones de los píxeles a lo largo de la elipse.

$k$	$p1_k$	$(x_{k+1}, y_{k+1})$	$2r_y^2 x_{k+1}$	$2r_x^2 y_{k+1}$
0	-332	(1, 6)	72	768
1	-224	(2, 6)	144	768
2	-44	(3, 6)	216	768
3	208	(4, 5)	288	640
4	-108	(5, 5)	360	640
5	288	(6, 4)	432	512
6	244	(7, 3)	504	384

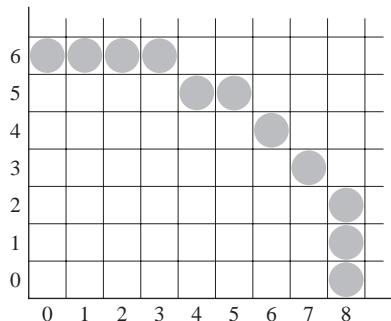
Ahora salimos de la región 1, ya que  $2r_y^2 x > 2r_x^2 y$ .

Para la región 2, el punto inicial es  $(x_0, y_0) = (7, 3)$  y el parámetro de decisión inicial es:

$$p2_0 = f_{\text{elipse}}\left(7 + \frac{1}{2}, 2\right) = -151$$

Las posiciones restantes a lo largo del trayecto elíptico en el primer cuadrante se pueden calcular entonces como:

$k$	$p1_k$	$(x_{k+1}, y_{k+1})$	$2r_y^2 x_{k+1}$	$2r_x^2 y_{k+1}$
0	-151	(8, 2)	576	256
1	233	(8, 1)	576	128
2	745	(8, 0)	—	—



**FIGURA 3.28.** Posiciones de los píxeles a lo largo de un trayecto elíptico centrado en el origen con  $r_x = 8$  y  $r_y = 6$ , utilizando el algoritmo del punto medio para calcular las ubicaciones dentro del primer cuadrante.

En la Figura 3.28 se muestra una gráfica de las posiciones calculadas para la elipse dentro del primer cuadrante.

En el siguiente segmento de código, se proporcionan procedimientos de ejemplo para la implementación del algoritmo del punto medio para el cálculo de una elipse. Es necesario introducir en el procedimiento `ellipseMidpoint` los parámetros de la elipse Rx, Ry, xCenter e yCenter. Entonces se calculan las posiciones a lo largo de la curva en el primer cuadrante y dichas posiciones se pasan al procedimiento `ellipsePlotPoints`. Se utilizan consideraciones de simetría para obtener las posiciones de la elipse en los otros tres cuadrantes, y la rutina `setPixel` asigna el color de la elipse a las posiciones del búfer de imagen correspondientes a estas posiciones de pantalla.

```

inline int round (const float a) { return int (a + 0.5); }

/* El siguiente procedimiento acepta valores que definen el centro
 * de la elipse y sus semiejes mayor y menor, calculando las
 * posiciones de la elipse mediante el algoritmo del punto medio.
 */
void ellipseMidpoint (int xCenter, int yCenter, int Rx, int Ry)
{
    int Rx2 = Rx * Rx;
    int Ry2 = Ry * Ry;
    int twoRx2 = 2 * Rx2;
    int twoRy2 = 2 * Ry2;
    int p;
    int x = 0;
    int y = Ry;
    int px = 0;
    int py = twoRx2 * y;
    void ellipsePlotPoints (int, int, int, int);

    /* Dibujar el punto inicial en cada cuadrante. */

```

```

ellipsePlotPoints (xCenter, yCenter, x, y);

/* Región 1 */
p = round (Ry2 - (Rx2 * Ry) + (0.25 * Rx2));
while (px < py) {
    x++;
    px += twoRy2;
    if (p < 0)
        p += Ry2 + px;
    else {
        y--;
        py -= twoRx2;
        p += Ry2 + px - py;
    }
    ellipsePlotPoints (xCenter, yCenter, x, y);
}

/* Región 2 */
p = round (Ry2 * (x+0.5) * (x+0.5) + Rx2 * (y-1) * (y-1) - Rx2 * Ry2);
while (y > 0) {
    y--;
    py -= twoRx2;
    if (p > 0)
        p += Rx2 - py;
    else {
        x++;
        px += twoRy2;
        p += Rx2 - py + px;
    }
    ellipsePlotPoints (xCenter, yCenter, x, y);
}

void ellipsePlotPoints (int xCenter, int yCenter, int x, int y);
{
    setPixel (xCenter + x, yCenter + y);
    setPixel (xCenter - x, yCenter + y);
    setPixel (xCenter + x, yCenter - y);
    setPixel (xCenter - x, yCenter - y);
}

```

## 3.11 OTRAS CURVAS

---

Hay diversas funciones generadoras de curvas que resultan útiles para modelar objetos, para especificar trácticos de animación, para dibujar gráficas de funciones y de datos y para otras aplicaciones gráficas. Entre las curvas más comunes encontramos la cónicas, las funciones trigonométricas y exponenciales, las distribuciones de probabilidad, los polinomios generales y las funciones de tipo *spline*. Pueden generarse gráficas de estas curvas con métodos similares a los que hemos explicado para las funciones circulares y elípticas.

Podemos obtener las posiciones a lo largo de los trayectos curvos directamente a partir de las ecuaciones explícitas  $y = f(x)$ , o mediante ecuaciones paramétricas. Alternativamente, podemos aplicar el método incremental del punto medio para dibujar las curvas descritas mediante funciones implícitas  $f(x, y) = 0$ .

Un método simple para dibujar una línea curva consiste en aproximarla mediante segmentos de línea recta. En este caso suelen resultar útiles las representaciones paramétricas, con el fin de obtener posiciones equiespaciadas a lo largo del trayecto curvo, posiciones que utilizaremos para definir los extremos de los segmentos lineales. También podemos generar posiciones equiespaciadas a partir de una ecuación explícita seleccionando la variable independiente de acuerdo con la pendiente de la curva. Cuando la pendiente de  $y = f(x)$  tenga una magnitud inferior a 1, seleccionaremos  $x$  como variable independiente y calcularemos los valores  $y$  para incrementos iguales de  $x$ . Para obtener un espacio igual cuando la pendiente tenga una magnitud superior a 1, utilizaremos la función inversa,  $x = f^{-1}(y)$  y calcularemos los valores de  $x$  para pasos constantes de  $y$ .

Para generar una gráfica de un conjunto de valores de datos discretos se utilizan aproximaciones mediante líneas rectas o curvas. Podemos unir los puntos discretos con segmentos lineales o podemos utilizar técnicas de regresión lineal (mínimos cuadrados) para aproximar el conjunto de datos mediante una única línea recta. Las técnicas no lineales de mínimos cuadrados se utilizan para mostrar el conjunto de datos mediante alguna función de aproximación, que usualmente es un polinomio.

Al igual que con los círculos y las elipses, muchas funciones poseen propiedades de simetría que pueden aprovecharse para reducir el número de cálculos de coordenadas a lo largo de los trayectos curvos. Por ejemplo, la función de distribución de probabilidad normal es simétrica en torno a la posición central (la media) y todos los puntos de un ciclo de una curva sinusoidal pueden generarse a partir de los puntos contenidos en un intervalo de  $90^\circ$ .

## Secciones cónicas

En general, podemos describir una **sección cónica** (o **cónica**) mediante la ecuación de segundo grado:

$$Ax^2 + By^2 + Cxy + Dx + Ey + F = 0 \quad (3.52)$$

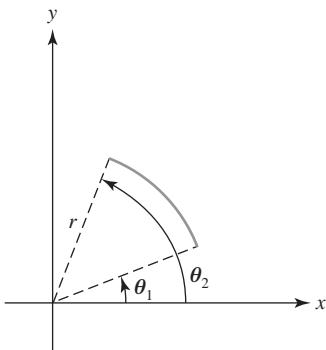
donde los valores de los parámetros  $A, B, C, D, E$  y  $F$  determinan el tipo de curva que tenemos que dibujar. Dado este conjunto de coeficientes, podemos determinar la cónica concreta definida por la ecuación evaluando el discriminante  $B^2 - 4AC$ :

$$B^2 - 4AC \begin{cases} < 0, & \text{genera una elipse (o círculo)} \\ = 0, & \text{genera una parábola} \\ > 0, & \text{genera una hipérbola} \end{cases} \quad (3.53)$$

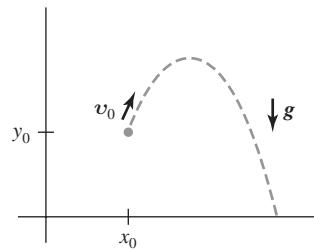
Por ejemplo, obtenemos la ecuación del círculo 3.26 cuando  $A = B = 1, C = 0, D = -2x_c, E = -2y_c$  y  $F = x_c^2 + y_c^2 - r^2$ . La Ecuación 3.52 también describe las cónicas «degeneradas»: puntos y líneas rectas.

En algunas aplicaciones, los arcos circulares y elípticos pueden especificarse cómodamente proporcionando los valores angulares inicial y final del arco, como se ilustra en la Figura 3.29, y dichos arcos se definen en ocasiones indicando las coordenadas de sus extremos. En cualquiera de los casos, podemos generar el arco utilizando un algoritmo del punto medio modificado, o bien podemos dibujar un conjunto de segmentos lineales que aproximen el arco.

Las elipses, las hipérbolas y las parábolas son particularmente útiles en ciertas aplicaciones de animación. Estas curvas describen movimientos orbitales y otros tipos de movimientos para objetos sobre los que actúan fuerzas gravitatorias, electromagnéticas o nucleares. Las órbitas planetarias del sistema solar, por ejemplo, pueden aproximarse mediante elipses y un objeto proyectado en un campo gravitatorio uniforme describe una trayectoria parabólica. La Figura 3.30 muestra un trayecto parabólico en posición estándar para un campo gravitatorio que actúa en la dirección y negativa. La ecuación explícita para la trayectoria parabólica del objeto mostrado puede escribirse como:



**FIGURA 3.29.** Un arco circular centrado en el origen, definido mediante el ángulo inicial  $\theta_1$ , el ángulo final  $\theta_2$  y el radio  $r$ .



**FIGURA 3.30.** Trayecto parabólico de un objeto arrojado en un campo gravitatorio descendente, con posición inicial  $(x_0, y_0)$ .

$$y = y_0 + a(x - x_0)^2 + b(x - x_0) \quad (3.54)$$

con constantes  $a$  y  $b$  determinadas por la velocidad inicial  $v_0$  del objeto y por la aceleración  $g$  debida a la fuerza gravitatoria uniforme. También podemos describir dicho tipo de movimientos parabólicos mediante ecuaciones paramétricas, utilizando un parámetro temporal  $t$ , medido en segundos a partir del punto inicial de proyección:

$$\begin{aligned} x &= x_0 + v_{x0}t \\ y &= y_0 + v_{y0}t - \frac{1}{2}gt^2 \end{aligned} \quad (3.55)$$

Aquí,  $v_{x0}$  y  $v_{y0}$  son las componentes iniciales de la velocidad y el valor de  $g$  cerca de la superficie de la Tierra es aproximadamente de 980 cm/sec<sup>2</sup>. A continuación, las posiciones del objeto a lo largo de la trayectoria parabólica se calculan para intervalos de tiempo seleccionados.

Las curvas hiperbólicas (Figura 3.31) son útiles en diversas aplicaciones de visualización científica. El movimiento de objetos a lo largo de trayectorias hiperbólicas se produce en conexión con la colisión de partículas cargadas y también en relación con ciertos problemas gravitatorios. Por ejemplo, los cometas o meteoritos que se mueven alrededor del Sol pueden describir trayectorias parabólicas y escapar hacia el espacio exterior, para no volver nunca. La rama concreta (izquierda o derecha, en la Figura 3.31) que describe el movimiento de un objeto depende de las fuerzas que intervengan en el problema. Podemos escribir la ecuación estándar para la hipérbola de la Figura 3.31, centrada en el origen, de la forma siguiente:

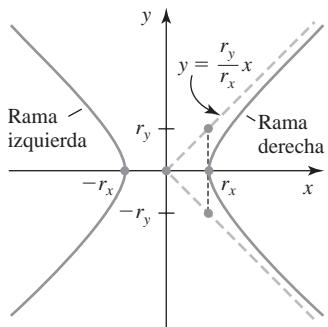
$$\left(\frac{x}{r_x}\right)^2 - \left(\frac{y}{r_y}\right)^2 = 1 \quad (3.56)$$

donde  $x \leq -r_x$  para la rama izquierda y  $x \geq r_x$  para la rama derecha. Puesto que esta ecuación difiere de la Ecuación 3.39 para una elipse estándar únicamente en el signo correspondiente a los términos  $x^2$  e  $y^2$ , podemos generar los puntos a lo largo de un trayecto hiperbólico utilizando un algoritmo de dibujo de elipses ligeramente modificado.

Las parábolas y las hipérbolas poseen un eje de simetría. Por ejemplo, la parábola descrita por la Ecuación 3.55 es simétrica con respecto al eje:

$$x = x_0 + v_{x0}v_{y0}/g$$

Los métodos utilizados en el algoritmo del punto medio para el dibujo de elipses pueden aplicarse directamente para obtener puntos a lo largo de uno de los lados del eje de simetría de las trayectorias hiperbólicas



**FIGURA 3.31.** Ramas izquierda y derecha de una hipérbola en posición estándar con el eje de simetría dispuesto según el eje  $x$ .



**FIGURA 3.32.** Una curva de tipo *spline* formada mediante secciones polinómicas cúbicas individuales definidas entre una serie de puntos especificados.

y parabólicas, dentro de las dos regiones: (1) cuando la magnitud de la pendiente de la curva es inferior a 1 y (2) cuando la magnitud de la pendiente es superior a 1. Para esto, primero seleccionamos la forma apropiada de la Ecuación 3.52 y luego empleamos la función seleccionada para establecer las expresiones correspondientes a los parámetros de decisión en las dos regiones.

### Polinomios y curvas de tipo *spline*

Una función polinómica de grado  $n$  en  $x$  se define como:

$$\begin{aligned} y &= \sum_{k=0}^n a_k x^k \\ &= a_0 + a_1 x + \cdots + a_{n-1} x^{n-1} + a_n x^n \end{aligned} \quad (3.57)$$

donde  $n$  es un entero no negativo y los valores  $a_k$  son constantes, con  $a_n \neq 0$ . Se obtiene una curva cuadrática cuando  $n = 2$ , un polinomio cúbico cuando  $n = 3$ , una curva cuádrica cuando  $n = 4$ , etc. Y en el caso de  $n = 1$  tendremos una línea recta. Los polinomios son útiles en diversas aplicaciones gráficas, incluyendo el diseño de formas de objetos, la especificación de trayectos de animación y la representación de tendencias a partir de un conjunto discreto de puntos de datos.

El diseño de formas de objetos o de trayectorias de movimiento se suele realizar especificando primero unos cuantos puntos con el fin de definir el contorno general de la curva y luego ajustando los puntos seleccionados mediante un polinomio. Una forma de realizar el ajuste de la curva consiste en construir una sección curva de polinomio cúbico entre cada par de puntos especificados. Cada sección curva se describe entonces en forma paramétrica de la manera siguiente:

$$\begin{aligned} x &= a_{x0} + a_{x1}u + a_{x2}u^2 + a_{x3}u^3 \\ y &= a_{y0} + a_{y1}u + a_{y2}u^2 + a_{y3}u^3 \end{aligned} \quad (3.58)$$

donde el parámetro  $u$  varía a lo largo del intervalo que va de 0 a 1.0. Los valores de los coeficientes de  $u$  en las ecuaciones precedentes se determinan a partir de las ecuaciones de contorno aplicables a las secciones curvas. Una condición de contorno es que dos secciones curvas adyacentes tengan las mismas coordenadas de posición en la frontera entre ambas, y una segunda condición consiste en ajustar las pendientes de las dos curvas también en la frontera, con el fin de obtener una única curva suave continua (Figura 3.32). Las curvas continuas que se forman a partir de polinomios se denominan **curvas de tipo *spline*** o simplemente **splines**. Hay otras formas de definir las curvas de tipo *spline* y en el Capítulo 8 se analizan diversos métodos de generación de *splines*.

## 3.12 ALGORITMOS PARALELOS PARA CURVAS

---

Los métodos para aprovechar el paralelismo en la generación de curvas son similares a los que se emplean a la hora de visualizar segmentos de línea recta. Podemos adaptar un algoritmo secuencial realizando una partición de la curva y asignando procesadores a los distintos segmentos, o podemos desarrollar otros métodos y asignar los procesadores a una serie de particiones de la pantalla.

Un método paralelo del punto medio para la visualización de círculos consiste en dividir el arco circular comprendido entre  $45^\circ$  y  $90^\circ$  en una serie de subarcos de igual tamaño y asignar un procesador distinto a cada subarco. Como en el algoritmo paralelo de Bresenham para generación de líneas, necesitaremos realizar los cálculos para determinar el valor  $y$  y el parámetro de decisión  $p_k$  iniciales para cada procesador. Entonces se calcularán las posiciones de píxel para cada subarco y las posiciones en los otros octantes del círculo pueden obtenerse por simetría. De forma similar, el método paralelo del punto medio para la generación de elipses dividirá el arco elíptico del primer cuadrante en una serie de subarcos iguales y los asignará a un conjunto de procesadores independientes. De nuevo, las posiciones de los píxeles en los otros cuadrantes se determinará por simetría. Un esquema de particionamiento de pantalla para los círculos y las elipses consiste en asignar cada línea de exploración que cruza la curva a un procesador distinto. En este caso, cada procesador utiliza la ecuación del círculo o de la elipse para calcular las coordenadas de intersección con la curva.

Para la visualización de arcos elípticos u otras curvas, podemos simplemente utilizar el método de particionamiento según líneas de exploración. Cada procesador utilizará la ecuación de la curva para localizar las intersecciones que se producen para su línea de exploración asignada. Asignando procesadores a los píxeles individuales, cada procesador calcularía la distancia (o la distancia al cuadrado) desde la curva a su píxel asignado. Si la distancia calculada es inferior a un cierto valor predefinido, se dibujará el píxel.

## 3.13 DIRECCIONAMIENTO DE PÍXELES Y GEOMETRÍA DE LOS OBJETOS

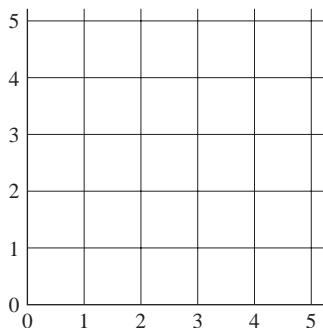
---

Al hablar de los algoritmos de digitalización para la visualización de primitivas gráficas, hemos asumido que las coordenadas del búfer de imagen hacían referencia al centro de cada posición de píxel en la pantalla. Vamos a considerar ahora los efectos de diferentes esquemas de direccionamiento y un método alternativo de direccionamiento de píxeles utilizados por algunos paquetes gráficos, incluido OpenGL.

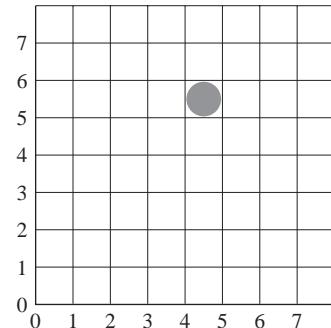
La descripción de un objeto que se introduce en un programa gráfico está expresada en términos de posiciones precisas definidas mediante coordenadas universales, posiciones que son puntos matemáticos infinitesimalmente pequeños. Pero cuando el objeto se digitaliza para almacenarlo en el búfer de imagen, la descripción de entrada se transforma en coordenadas de píxeles que hacen referencia a áreas de pantalla finitas, con lo que la imagen digitalizada que se visualiza puede no corresponderse exactamente con las dimensiones relativas del objeto de entrada. Si resulta importante preservar la geometría especificada de los objetos, podemos tratar de compensar esa traducción de los puntos matemáticos de entrada a áreas de píxel finitas. Una forma de hacer esto consiste simplemente en ajustar las dimensiones de los píxeles de los objetos visualizados para que se correspondan con las dimensiones indicadas en la descripción matemática original de la escena. Por ejemplo, si se especifica que un rectángulo tiene una anchura de 40 cm, podemos ajustar la visualización en pantalla para que el rectángulo tenga una anchura de 40 píxeles, donde la anchura de cada píxel representa un centímetro. Otro método consistiría en asignar las coordenadas universales a posiciones de pantalla intermedias entre los píxeles, para alinear las fronteras de los objetos con las fronteras de los píxeles, en lugar de con los centros de los píxeles.

### Coordenadas de cuadrícula de pantalla

La Figura 3.33 muestra una sección de la pantalla con una serie de líneas de cuadrícula que marcan las fronteras entre los píxeles, estando los píxeles separados por una distancia unitaria. En este esquema, cada posición de pantalla se especifica mediante la pareja de valores enteros que identifican una intersección de la



**FIGURA 3.33.** Sección inferior izquierda de un área de pantalla en la que las coordenadas se referencian mediante la intersección de líneas de la cuadrícula.



**FIGURA 3.34.** Píxel iluminado en la posición (4, 5).

cuadrícula entre dos píxeles. La dirección correspondiente a cualquier píxel estará ahora en su esquina inferior izquierda, como se ilustra en la Figura 3.34 y un trayecto lineal se trazará ahora entre las intersecciones de la cuadrícula. Por ejemplo, el trayecto lineal matemático para una polilínea que tuviera los vértices definidos mediante las coordenadas (0, 0), (5, 2) y (1, 4) sería como la mostrada en la Figura 3.35.

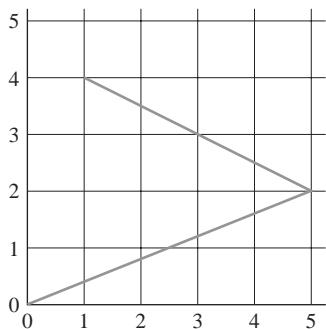
Utilizando las coordenadas de cuadrícula de pantalla, podemos identificar el área ocupada por un píxel con coordenadas de pantalla ( $x, y$ ) como el cuadrado de tamaño unidad que tiene sus esquinas opuestas en los puntos  $(x, y)$  y  $(x + 1, y + 1)$ . Este método de direccionamiento de píxeles tiene varias ventajas: evita que las fronteras de los píxeles estén definidas mediante valores semienteros, facilita la representación precisa de los objetos y simplifica el procesamiento necesario para muchos algoritmos y procedimientos de digitalización.

Los algoritmos para dibujo de líneas y generación de curvas que hemos analizado en las secciones anteriores siguen siendo válidos cuando se los aplica a posiciones de entrada expresadas mediante la cuadrícula de pantalla. Los parámetros de decisión de estos algoritmos serán ahora una medida de las diferencias de separación con respecto a la cuadrícula de pantalla, en lugar de las diferencias de separación con respecto a los centros de los píxeles.

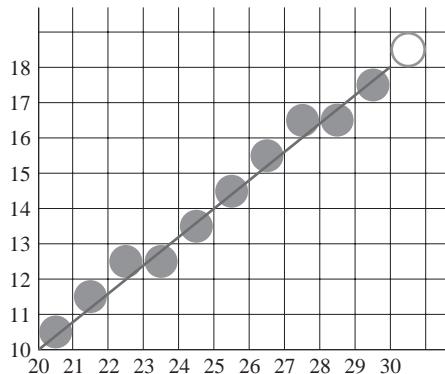
## Mantenimiento de las propiedades geométricas de los objetos visualizados

Cuando transformamos las descripciones geométricas de los objetos en representaciones mediante píxeles, lo que hacemos es transformar puntos matemáticos y líneas en una serie de áreas de pantalla finitas. Si queremos mantener para un objeto las medidas geométricas originales especificadas mediante las coordenadas de entrada, tenemos que tener en cuenta el tamaño finito de los píxeles a la hora de transformar la definición del objeto en una imagen en pantalla.

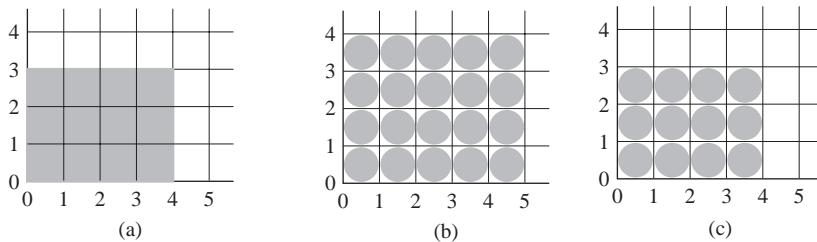
La Figura 3.36 muestra la línea dibujada en el ejemplo de aplicación del algoritmo de Bresenham de la Sección 3.5. Interpretando los dos extremos de la línea (20, 10) y (30, 18) como posiciones precisas de puntos de la cuadrícula, vemos que la línea no debe ir más allá de la posición (30, 18) de la cuadrícula. Si dibujáramos el píxel con coordenadas de pantalla (30, 18), como en el ejemplo proporcionado en la Sección 3.5, estaríamos mostrando una línea que abarcaría once unidades horizontales y 9 unidades verticales. Sin embargo, para la línea matemática,  $\Delta x = 10$  y  $\Delta y = 8$ . Si direccionamos los píxeles mediante las posiciones de sus centros, podemos ajustar la longitud de la línea mostrada omitiendo uno de los píxeles de los extremos, sin embargo, si consideramos que las coordenadas de pantalla están direccionando las fronteras de los píxeles, como se muestra en la Figura 3.36, dibujaremos la línea utilizando únicamente aquellos píxeles que estén «dentro» del trayecto lineal, es decir, únicamente aquellos píxeles que estén comprendidos entre los dos puntos extremos de la línea. En nuestro ejemplo, dibujaríamos el píxel de la izquierda en la posición (20, 10) y el píxel de más a la derecha en la posición (29, 17). Esto hace que se muestre una línea con la misma medida geométrica que la línea matemática que va de (20, 10) a (30, 18).



**FIGURA 3.35.** Trayecto formado por dos segmentos de línea conectados entre posiciones de pantalla definidas mediante una cuadrícula.



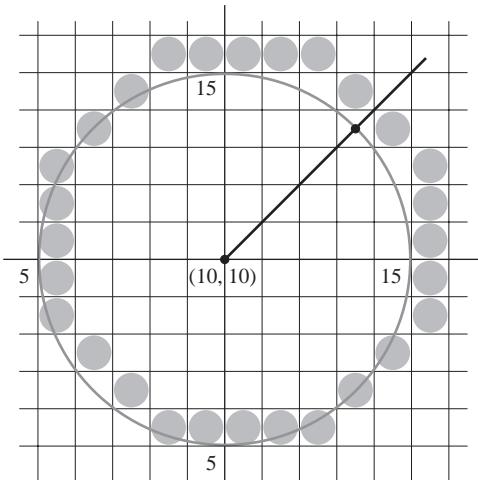
**FIGURA 3.36.** Trayectoria lineal y visualización correspondiente de los píxeles para las coordenadas (20, 10) y (30, 18) de sendos puntos extremos definidos mediante las líneas de la cuadrícula.



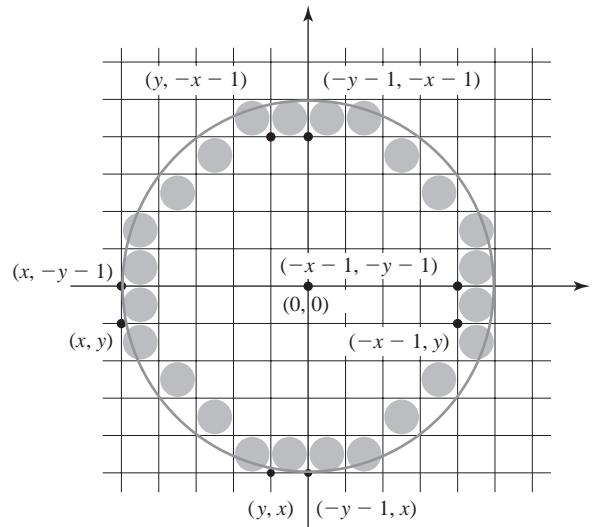
**FIGURA 3.37.** Conversión de un rectángulo (a) con vértices en las coordenadas de pantalla (0, 0), (4, 0), (4, 3) y (0, 3), con una visualización (b) que incluye las fronteras derecha y superior y con una visualización (c) que mantiene las magnitudes geométricas.

Para un área cerrada, podemos mantener las propiedades geométricas de entrada visualizando el área únicamente mediante aquellos píxeles que estén situados en el interior de las fronteras del objeto. Por ejemplo, el rectángulo definido mediante los vértices que se muestran en la Figura 3.37(a), expresados en coordenadas de pantalla, es mayor cuando lo mostramos relleno con píxeles hasta incluir las líneas de píxeles del borde que unen los vértices especificados. Tal como está definido, el área del rectángulo es de 12 unidades, pero si lo muestra como en la Figura 3.37(b) tendrá un área de 20 unidades. En la Figura 3.37(c), se mantienen las medidas originales del rectángulo, al mostrar únicamente los píxeles internos. La frontera derecha del rectángulo de entrada se encuentra en  $x = 4$ . Para mantener la anchura del rectángulo en la pantalla, fijamos la coordenada de cuadrícula del píxel de más a la derecha del rectángulo con el valor  $x = 3$ , ya que los píxeles de esta columna vertical abarcan el intervalo que va de  $x = 3$  a  $x = 4$ . De forma similar, la frontera matemática superior del rectángulo se encuentra en  $y = 3$ , así que la fila de píxeles superior para el rectángulo mostrado estará en  $y = 2$ .

Esta compensación del tamaño finito de los píxeles puede aplicarse a otros objetos, incluyendo aquellos que tienen fronteras curvas, para que la representación digitalizada mantenga las especificaciones de entrada de los objetos. Un círculo con radio 5 y posición central (10, 10) por ejemplo, se mostraría como la Figura 3.38 si empleamos el algoritmo del punto medio para generación de círculos utilizando los centros de los píxeles como posiciones de las coordenadas de pantalla. Pero el círculo dibujado tiene un diámetro de 11. Para dibujar el círculo con el diámetro definido de 10, podemos modificar el algoritmo del círculo con el fin de acortar cada línea de exploración y cada columna de píxeles, como en la Figura 3.39.



**FIGURA 3.38.** Una gráfica obtenida mediante el algoritmo del punto medio para la ecuación del círculo  $(x - 10)^2 + (y - 10)^2 = 5^2$ , utilizando las coordenadas de los centros de los píxeles.



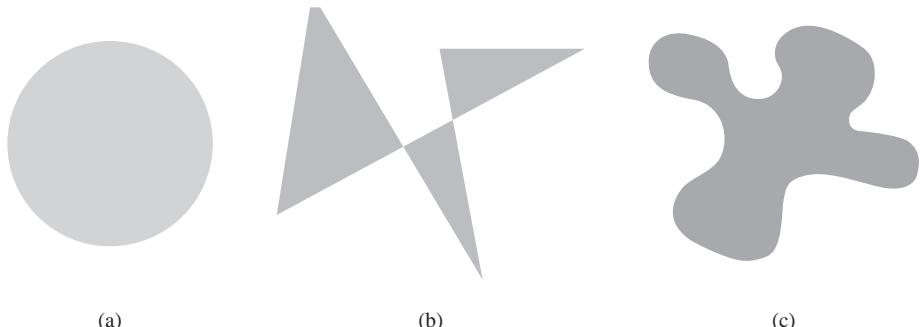
**FIGURA 3.39.** Modificación de la gráfica del círculo de la FIGURA 3.38 para mantener el diámetro del círculo especificado, que tiene valor 10.

Una forma de hacer esto consiste en generar los puntos en el sentido de las agujas del reloj a lo largo del arco circular del tercer cuadrante, comenzando en las coordenadas de pantalla  $(10, 5)$ . Para cada punto generado, se obtienen los otros siete puntos simétricos del círculo reduciendo los valores de la coordenada  $x$  en 1 unidad a lo largo de las líneas de exploración y reduciendo los valores de la coordenada  $y$  en 1 unidad a lo largo de las columnas de píxeles. Pueden aplicarse métodos similares a los algoritmos de generación de elipses con el fin de mantener las proporciones especificadas a la hora de visualizar una elipse.

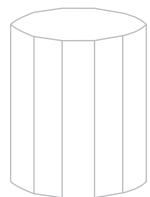
## 3.14 PRIMITIVAS DE ÁREAS RELLENAS

Otro elemento útil, además de los puntos, los segmentos lineales y las curvas, para la descripción de los componentes de una imagen son las áreas rellenas con algún color homogéneo o patrón. Un elemento de imagen de este tipo se denomina normalmente **área rellena**. La mayoría de las veces, las áreas rellenas se utilizan para describir las superficies de los objetos sólidos, pero también resultan útiles en diversas otras aplicaciones. Asimismo, las regiones rellenas suelen ser superficies planas, principalmente polígonos. Pero en términos generales hay muchos tipos de formas posibles para una región del dibujo que podamos querer llenar con algún determinado color. La Figura 3.40 ilustra algunos ejemplos de áreas rellenas. Por el momento, vamos a suponer que todas las áreas rellenas deben mostrarse con un color homogéneo especificado. En el Capítulo 4 se hablará de otras opciones de relleno.

Aunque pueden existir áreas rellenas de cualquier forma, las bibliotecas gráficas no suelen soportar la especificación de formas de relleno arbitrarias. La mayoría de las rutinas de biblioteca requieren que las áreas de relleno se especifiquen en forma de polígonos. Las rutinas gráficas pueden procesar más eficientemente los polígonos que otros tipos de áreas de relleno, porque las fronteras de los polígonos se describen mediante ecuaciones lineales. Además, la mayoría de las superficies curvas pueden aproximarse razonablemente bien mediante una serie de parches poligonales, de la misma forma que una línea curva puede aproximarse mediante un conjunto de segmentos lineales. Y cuando se aplican efectos de iluminación y procedimientos de sombreado de superficies, la superficie curva aproximada puede mostrarse con un grado de realismo bastante



**FIGURA 3.40.** Áreas rellenas de color homogéneo especificadas con diversas fronteras. (a) Una región circular rellena. (b) Un área rellena delimitada por una polilínea cerrada. (c) Un área rellena especificada mediante una frontera irregular curva.



**FIGURA 3.41.** Representación alámbrica de un cilindro, donde se muestran únicamente las caras delanteras (visibles) de la malla poligonal utilizada para aproximar las superficies.

bueno. La aproximación de una superficie curva mediante caras poligonales se denomina en ocasiones *teselación de la superficie*, o ajuste de la superficie mediante una *malla poligonal*. La Figura 3.41 muestra las superficies lateral y superior de un cilindro metálico que se ha aproximado mediante una malla poligonal. La visualización de tales tipos de figuras puede generarse muy rápidamente mediante vistas *alámbricas*, que sólo muestran las aristas de los polígonos con el fin de proporcionar una indicación general de la estructura de la superficie. Posteriormente, el modelo alámbrico puede sombrearse para generar una imagen de una superficie material con aspecto natural. Los objetos descritos con un conjunto de parches de superficie poligonales se suelen denominar **objetos gráficos estándar** o simplemente **objetos gráficos**.

En general, podemos crear áreas rellenas con cualquier especificación de contorno, como por ejemplo un círculo o un conjunto conectado de secciones de curvas de tipo *spline*. Y algunos de los métodos poligonales explicados en la siguiente sección pueden adaptarse para mostrar áreas de relleno con contornos no lineales. En el Capítulo 4 se explican otros métodos de relleno de áreas para objetos con contornos curvados.

## 3.15 ÁREAS DE RELLENO POLIGONALES

Desde el punto de vista matemático, un **polígono** se define como una figura plana especificada mediante un conjunto de tres o más puntos, denominados *vértices*, que se conectan en secuencia mediante segmentos lineales, denominados *bordes*, o *aristas* del polígono. Además, en geometría básica, es necesario que las aristas del polígono no tengan ningún punto en común aparte de los extremos. Así, por definición, un polígono debe tener todos sus vértices en un mismo plano y las aristas no pueden cruzarse. Como ejemplos de polígonos podemos citar los triángulos, los rectángulos, los octógonos y los decágonos. Algunas veces, cualquier figura plana con un contorno de tipo polilínea cerrado se denomina también polígono, y si además sus aristas no se cortan se le denomina *polígono estándar* o *polígono simple*. Para tratar de evitar referencias ambiguas a los objetos, utilizaremos el término «polígono» para referirnos exclusivamente a las formas planas que tienen un contorno de tipo polilínea cerrada y en el que las aristas no se cortan.

Para una aplicación infográfica, es posible que un conjunto designado de vértices de un polígono no caigan exactamente en un mismo plano. Esto puede deberse a los errores de redondeo en el cálculo de los valo-

res numéricos, a errores en la selección de las coordenadas de los vértices o, más normalmente, a la aproximación de una superficie curva mediante un conjunto de parches poligonales. Una forma de rectificar este problema consiste simplemente en dividir la malla de superficie especificada en una serie de triángulos. Pero en algunos casos puede haber razones para retener la forma original de los parches de la malla, así que se han desarrollado métodos para aproximar una forma poligonal no plana mediante una figura plana. Hablaremos de cómo calcular estas aproximaciones planas en la subsección dedicada a las ecuaciones del plano.

## Clasificaciones de los polígonos

Un **ángulo interior** de un polígono es un ángulo dentro del contorno del polígono que está formado por dos aristas adyacentes. Si todos los ángulos interiores de un polígono son menores o iguales que  $180^\circ$ , el polígono es **convexo**. Una definición equivalente de polígono convexo es la que dice que el interior del polígono está completamente contenido en uno de los lados de la línea de extensión infinita de cualquiera de sus aristas. Asimismo, si seleccionamos cualesquiera dos puntos del interior de un polígono convexo, el segmento de línea que une los dos puntos se encuentra también en el interior del polígono. Un polígono que no es convexo se denomina **cóncavo**. La Figura 3.42 proporciona ejemplos de polígonos convexos y cóncavos.

A menudo se utiliza el término **polígono degenerado** para describir un conjunto de vértices que son colineales o en el que algunos de los vértices son coincidentes. Los vértices polineales generan un segmento de línea, mientras que los vértices repetidos pueden generar una forma poligonal con líneas extrañas, aristas solapadas o aristas de longitud 0. Algunas veces se aplica también el término polígono degenerado a una lista de vértices que contienen menos de tres vértices.

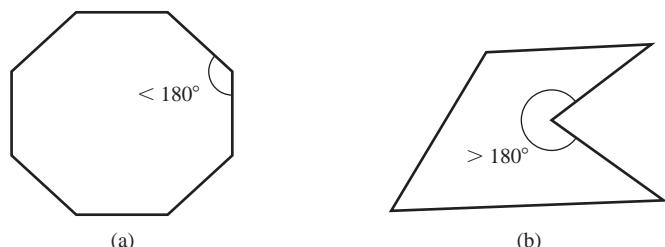
Para ser robusto, un paquete gráfico podría rechazar los conjuntos de vértices degenerados o no planares, pero esto requiere pasos de procesamiento adicionales para identificar estos problemas, por lo que los sistemas gráficos suelen dejar dicho tipo de consideraciones al programador.

Los polígonos cóncavos también presentan problemas. La implementación de los algoritmos de rellenado y de otras rutinas gráficas es más complicada para los polígonos cóncavos, por lo que suele ser más eficiente partir un polígono en un conjunto de polígonos convexos antes de continuar con su procesamiento. Al igual que sucede con otros algoritmos de preprocessamiento de polígonos, la división de polígonos cóncavos no suele incluirse en las bibliotecas gráficas. Algunos paquetes gráficos, incluyendo OpenGL, requieren que todos los polígonos llenados sean convexos, mientras que otros sistemas sólo aceptan áreas de relleno triangulares, que simplifican enormemente muchos de los algoritmos de visualización.

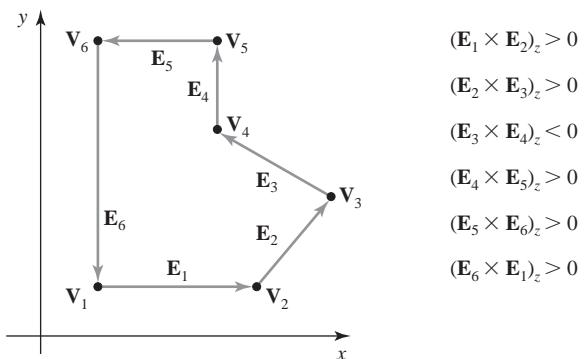
## Identificación de polígonos cóncavos

Un polígono cóncavo tiene al menos uno de sus ángulos interiores superior a  $180^\circ$ . Asimismo, la extensión de algunas aristas de un polígono cóncavo se cortará con otras aristas, y algunas parejas de puntos del interior del polígono producirá un segmento de línea que intersectará con el contorno del polígono. Por tanto, podemos utilizar cualquiera de estas características de un polígono cóncavo como base para construir un algoritmo de identificación.

Si asignamos un vector a cada arista de un polígono, podemos utilizar el producto vectorial de las aristas adyacentes para comprobar la concavidad. Todos esos productos vectoriales tendrán el mismo signo (positi-



**FIGURA 3.42.** Un polígono convexo (a) y un polígono cóncavo (b).



**FIGURA 3.43.** Identificación de un polígono cónvexo mediante el cálculo de los productos vectoriales de las parejas sucesivas de vectores representativos de las aristas.

vo o negativo) en los polígonos convexos. Por tanto, si algún producto vectorial nos da un valor positivo y otro nos da un valor negativo, tendremos un polígono cónvexo. La Figura 3.43 ilustra el método del producto vectorial de los vectores de las aristas para la identificación de polígonos cónvexos.

Otra forma de identificar un polígono cónvexo consiste en examinar las posiciones de los vértices del polígono en relación con la línea de extensión de cualquier arista. Si algunos de los vértices se encuentran a un lado de la línea de extensión y otros están en el otro lado, el polígono será cónvexo.

### División de los polígonos cónvexos

Una vez identificado un polígono cónvexo, podemos dividirlo en un conjunto de polígonos convexos. Esto puede realizarse utilizando los vectores de las aristas y los productos vectoriales correspondientes. Alternativamente, podemos utilizar las posiciones de los vértices en relación con la línea de extensión de una arista para determinar qué vértices se encuentran en un lado de la línea y cuáles están en el otro. Para los siguientes algoritmos, vamos a suponer que todos los polígonos se encuentran en el plano  $xy$ . Por supuesto, la posición original de un polígono descrita en coordenadas universales puede no estar en el plano  $xy$ , pero siempre podemos moverlo hasta ese plano utilizando los métodos de transformación que se explican en el Capítulo 5.

Con el **método vectorial** para dividir un polígono cónvexo, primero necesitamos formar los vectores de las aristas. Dadas dos posiciones de vértice consecutivas,  $V_k$  y  $V_{k+1}$ , definimos el vector de la arista que los une de la forma siguiente:

$$E_k = V_{k+1} - V_k$$

A continuación calculamos los productos vectoriales de los sucesivos vectores de las aristas, siguiendo el orden del perímetro del polígono. Si la componente  $z$  de algún producto vectorial es positiva mientras que otros productos vectoriales tienen una componente  $z$  negativa, el polígono será cónvexo, mientras que en caso contrario el polígono será cónvexo. Esto presupone que no haya tres vértices sucesivos o lineales, ya que en ese caso el producto vectorial de los dos vectores representativos de las aristas que conectan estos vértices sería cero. Si todos los vértices son colineales, tendremos un polígono degenerado (una línea recta). Podemos aplicar el método vectorial procesando los vectores de las aristas en sentido contrario a las agujas del reloj. Si cualquiera de los productos vectoriales tiene una componente  $z$  negativa (como la Figura 3.43), el polígono será cónvexo y podemos partirla utilizando la línea del primero de los vectores con los que se ha calculado el producto vectorial. El siguiente ejemplo ilustra este método para la división de un polígono cónvexo.

#### Ejemplo 3.4 Método vectorial para la división de polígonos cónvexos

La Figura 3.44 muestra un polígono cónvexo con seis aristas. Los vectores de las aristas para este polígono pueden expresarse como:

$$\mathbf{E}_1 = (1, 0, 0) \quad \mathbf{E}_2 = (1, 1, 0)$$

$$\mathbf{E}_3 = (1, -1, 0) \quad \mathbf{E}_4 = (0, 2, 0)$$

$$\mathbf{E}_5 = (-3, 0, 0) \quad \mathbf{E}_6 = (0, -2, 0)$$

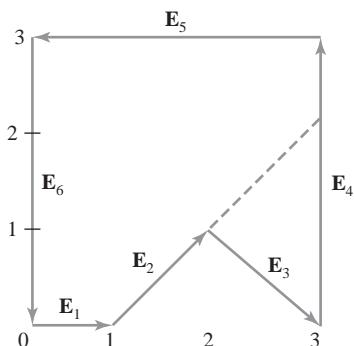
donde la componente  $z$  es 0, ya que todas las aristas se encuentran en el plano  $xy$ . El producto vectorial  $\mathbf{E}_j \times \mathbf{E}_k$  para dos vectores sucesivos es un vector perpendicular al plano  $xy$  con una componente  $z$  igual a  $E_{jx}E_{ky} - E_{kx}E_{jy}$ :

$$\mathbf{E}_1 \times \mathbf{E}_2 = (0, 0, 1) \quad \mathbf{E}_2 \times \mathbf{E}_3 = (0, 0, -2)$$

$$\mathbf{E}_3 \times \mathbf{E}_4 = (0, 0, 2) \quad \mathbf{E}_4 \times \mathbf{E}_5 = (0, 0, 6)$$

$$\mathbf{E}_5 \times \mathbf{E}_6 = (0, 0, 6) \quad \mathbf{E}_6 \times \mathbf{E}_1 = (0, 0, 2)$$

Puesto que el producto vectorial  $\mathbf{E}_2 \times \mathbf{E}_3$  tiene una componente  $z$  negativa, partiremos el polígono según la línea correspondiente al vector  $\mathbf{E}_2$ . La ecuación de la línea correspondiente a esta arista tiene una pendiente de 1 e intercepta el eje  $y$  en  $-1$ . A continuación determinamos la intersección de esta línea con las otras aristas del polígono, con el fin de partir el polígono en dos partes. No hay ningún otro producto vectorial negativo, por lo que los dos nuevos polígonos serán convexos.

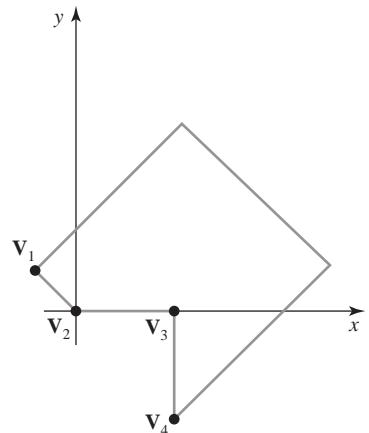


**FIGURA 3.44.** División de un polígono cóncavo utilizando el método de los vectores.

También podemos dividir un polígono cóncavo utilizando un **método rotacional**. Vamos a movernos en sentido contrario a las agujas del reloj siguiendo las aristas del polígono, y vamos a desplazar la posición del polígono de modo que cada vértice  $V_k$  se encuentre por turno en el origen de coordenadas. A continuación, rotamos el polígono alrededor del origen en el sentido de las agujas del reloj, de modo que el siguiente vértice  $V_{k+1}$  se encuentre sobre el eje  $x$ . Si el siguiente vértice,  $V_{k+2}$ , está por debajo del eje  $x$ , el polígono será cóncavo. En este caso, partiremos el polígono según el eje  $x$  para formar dos nuevos polígonos y repetiremos el test de concavidad para cada uno de los dos nuevos polígonos. Los pasos anteriores se repiten hasta que hayamos comprobado todos los vértices de la lista de polígonos. En el Capítulo 5 se explican en detalle los métodos para rotar y desplazar un objeto. La Figura 3.45 ilustra el método rotacional para la división de un polígono cóncavo.

### División de un polígono convexo en un conjunto de triángulos

Una vez obtenida la lista de vértices para un polígono convexo, podemos transformarlo en un conjunto de triángulos. Esto puede hacerse seleccionando primero cualquier secuencia de tres vértices consecutivos y for-



**FIGURA 3.45.** División de un polígono cóncavo utilizando el método rotacional. Después de mover  $\mathbf{V}_2$  hasta el origen de coordenadas y de rotar  $\mathbf{V}_3$  hasta situarlo en el eje  $x$ , vemos que  $\mathbf{V}_4$  está por debajo del eje  $x$ , así que dividimos el polígono según la línea de extensión del vector  $\overline{\mathbf{V}_2\mathbf{V}_3}$  que es el eje  $x$ .

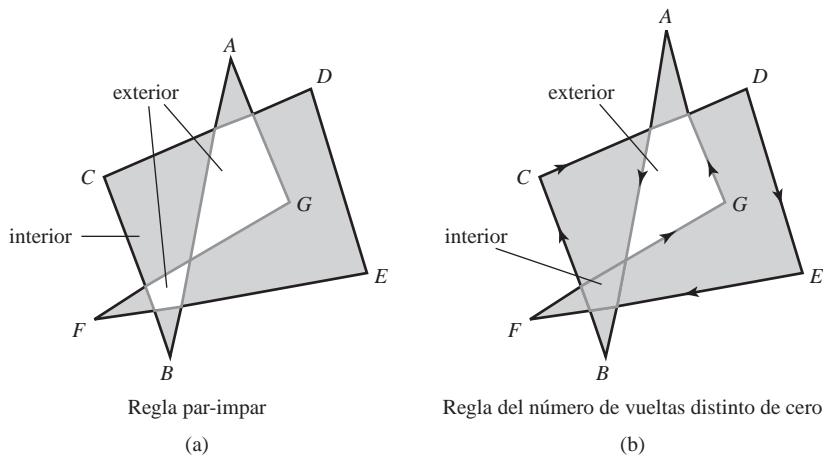
mando con ella un nuevo polígono (un triángulo). Entonces se borra el vértice intermedio del triángulo de la lista de vértices original, volviéndose a aplicar el mismo procedimiento a esta lista de vértices modificada para extraer otro triángulo. Continuamos formando triángulos de esta forma hasta que el polígono original quede reducido a sólo tres vértices, que definirán el último triángulo del conjunto. Un polígono cóncavo también puede dividirse en un conjunto de triángulos utilizando esta técnica, siempre y cuando los tres vértices seleccionados en cada caso formen un ángulo interior que sea inferior a  $180^\circ$  (un ángulo convexo).

### Pruebas dentro-fuera

Son varios los procesos gráficos que necesitan poder identificar las regiones interiores de los objetos. Identificar el interior de un objeto simple, como un polígono convexo, un círculo o una esfera resulta generalmente sencillo. Pero en ocasiones debemos tratar con objetos más complejos. Por ejemplo, podríamos definir una región de relleno compleja con aristas que se cortarán, como en la Figura 3.46. Para este tipo de formas, no siempre está claro qué regiones del plano  $xy$  debemos denominar “interiores” y qué regiones hay que designar como “exteriores” según el contorno del objeto. Dos algoritmos comúnmente utilizados para identificar las áreas interiores de una figura plana son la regla par-impar y la regla del número de vueltas distinto de cero.

Podemos aplicar la **regla par-impar**, también denominada *regla de paridad impar*, dibujando primero conceptualmente una línea desde cualquier posición  $\mathbf{P}$  hasta un punto distante, situado fuera del recuadro de contorno de la polilínea cerrada. Entonces contamos el número de segmentos de línea que se cortan con esta línea. Si el número de segmentos cruzados por esta línea es impar, entonces consideramos  $\mathbf{P}$  como un punto *interior*. En caso contrario,  $\mathbf{P}$  es un punto *exterior*. Para obtener un recuento preciso de las intersecciones con los segmentos, debemos asegurarnos de que la línea elegida no pase por ninguno de los vértices de los segmentos. La Figura 3.46(a) muestra las regiones interiores y exteriores obtenidas utilizando la regla par-impar para una polilínea cerrada que se auto-intersecta. Podemos utilizar este procedimiento, por ejemplo, para llenar con un color especificado la región interior comprendida entre dos círculos concéntricos o dos polígonos concéntricos.

Otro método para definir las regiones interiores es la **regla del número de vueltas distinto de cero**, que cuenta el número de veces que el contorno de un objeto «da la vuelta» alrededor de un punto concreto en el sentido de las agujas del reloj. Este número se denomina **número de vueltas** y los puntos interiores de un objeto bidimensional pueden definirse como aquellos que tienen un valor de número de vueltas distinto de cero. Para aplicar la regla del número de vueltas distinto de cero, inicializamos el número de vueltas a 0 e imaginamos de nuevo una línea dibujada desde cualquier posición  $\mathbf{P}$  hasta un punto distante situado más allá del recuadro de contorno del objeto. La línea que elijamos no debe pasar a través de ningún vértice. A medida que nos movemos a lo largo de la línea desde la posición  $\mathbf{P}$  hasta el punto distante, contamos el número de seg-



**FIGURA 3.46.** Identificación de las regiones exterior e interior de una polilínea cerrada que contiene segmentos que se auto-intersecan.

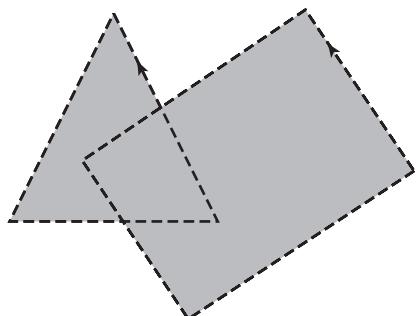
mentos de línea del objeto que cruzan la línea de referencia en cada dirección. Añadiremos un 1 al número de vueltas cada vez que cortemos un segmento que cruza la línea de derecha a izquierda y restaremos 1 cada vez que crucemos un segmento que corte la línea de izquierda a derecha. El valor final del número de vueltas, después de haber contado todos los cruces con los segmentos lineales de contorno, determinará la posición relativa de  $P$ . Si el número de vueltas es distinto de cero,  $P$  se considerará un punto interior. En caso contrario,  $P$  será considerado un punto exterior. La Figura 3.46(b) muestra las regiones interior y exterior definidas mediante la regla del número de vueltas distinto de cero para una polilínea cerrada que se auto-interseca. Para objetos, como polígonos y círculos, la regla del número de vueltas distinto de cero y la regla par-impar proporcionan los mismos resultados, pero para formas más complejas, los dos métodos pueden proporcionar regiones interiores y exteriores distintas, como en el Ejemplo de la Figura 3.46.

Una forma de determinar la dirección de los cruces con los segmentos de línea que forman el contorno consiste en definir una serie de vectores para las aristas del objeto y para la propia línea de referencia. Entonces, calculamos el producto vectorial del vector  $\mathbf{u}$  que va desde  $P$  hasta un punto distante con el vector  $\mathbf{E}$  correspondiente a la arista del objeto, para cada arista que se intersecte con la línea. Suponiendo que tenemos un objeto bidimensional en el plano  $xy$ , la dirección de cada producto vectorial estará en la dirección  $+z$  o en la dirección  $-z$ . Si la componente  $z$  de un producto vectorial  $\mathbf{u} \times \mathbf{E}$  para una intersección concreta es positiva, dicho segmento cruza de derecha a izquierda y añadiremos 1 al número de vueltas. En caso contrario, el segmento cruza de izquierda a derecha y restaremos 1 del número de vueltas.

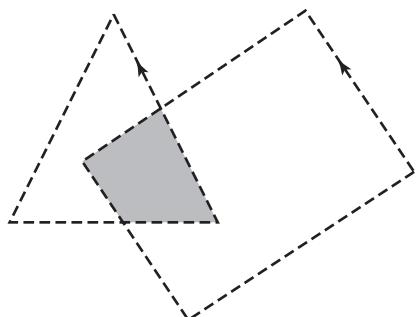
Una forma algo más simple de calcular la dirección de los cruces consiste en utilizar productos escalares en lugar de productos vectoriales. Para hacer esto, definimos un vector que sea perpendicular al vector  $\mathbf{u}$  y que vaya de derecha a izquierda según se mira a lo largo de la línea que parte de  $P$  en la dirección de  $\mathbf{u}$ . Si las componentes de  $\mathbf{u}$  son  $(u_x, u_y)$ , entonces el vector perpendicular a  $\mathbf{u}$  tendrá componentes  $(-u_y, u_x)$  (Apéndice A). Ahora, si el producto escalar de este vector perpendicular y el vector de la arista es positivo, el cruce se produce de derecha a izquierda y sumaremos 1 al número de vueltas. En caso contrario, la arista cruza nuestra línea de referencia de izquierda a derecha y restaremos 1 del número de vueltas. La regla del número de vueltas distinto de cero tiende a clasificar como interiores algunas áreas que la regla par-impar considera como exteriores, y puede ser más versátil en algunas aplicaciones. En general, las figuras planas pueden definirse mediante múltiples componentes disjuntas y la dirección especificada para cada conjunto de contornos disjuntos puede utilizarse para designar las regiones interior y exterior. Como ejemplos tendríamos los caracteres (como las letras del alfabeto y los símbolos de puntuación), los polígonos anidados y los círculos o elipses concéntricos. Para líneas curvas, la regla par-impar se aplica calculando las intersecciones con los trayectos

curvos. De forma similar, con la regla del número de vueltas distinto de cero necesitamos calcular vectores tangentes a las curvas en los puntos de cruce con la línea de referencia que parte de la posición **P**.

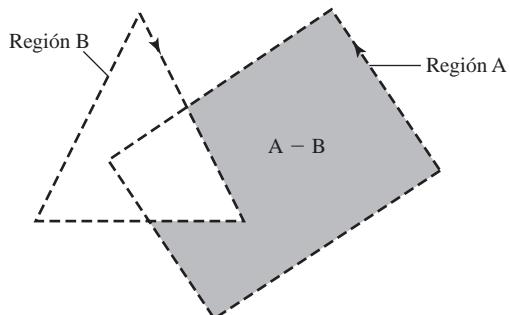
Pueden utilizarse algunas variaciones de la regla del número de vueltas distinto de cero para definir las regiones interiores de otras maneras. Por ejemplo, podríamos definir un punto como interior si su número de vueltas es positivo o si es negativo. O podríamos usar cualquier otra regla para generar una diversidad de formas de relleno. Algunas veces, se utilizan operaciones booleanas para especificar un área de relleno como combinación de dos regiones. Una forma de implementar operaciones booleanas consiste en utilizar una variación de la regla básica del número de vueltas. Con este esquema, primero definimos un contorno simple sin intersecciones para cada una de las dos regiones. Entonces, si consideramos que la dirección de cada contorno va en el sentido de las agujas del reloj, la unión de las dos regiones estará formada por todos aquellos puntos cuyo número de vueltas es positivo (Figura 3.47). De forma similar, la intersección de dos regiones con contornos que vayan en sentido contrario a las agujas del reloj contendrá aquellos puntos cuyo número de vueltas sea superior a 1, como se muestra en la Figura 3.48. Para definir un área de relleno que sea la diferencia entre las dos regiones,  $A - B$ , podemos encerrar la región A mediante un contorno que vaya en sentido contrario a las agujas del reloj y la región B con un contorno que vaya en el sentido de las agujas del reloj. Entonces, la región diferencia (Figura 3.49) será el conjunto de todos los puntos cuyo número de vueltas sea positivo.



**FIGURA 3.47.** Un área de relleno definida como una región que tiene un valor positivo para el número de vueltas. Este área de relleno es la unión, cada una de las cuales tiene un contorno que va en sentido contrario a las agujas del reloj.



**FIGURA 3.48.** Un área de relleno definida como una región con un número de vueltas superior a 1. Este área de relleno es la intersección entre dos regiones, cada una de las cuales tiene un contorno que va en sentido contrario a las agujas del reloj.



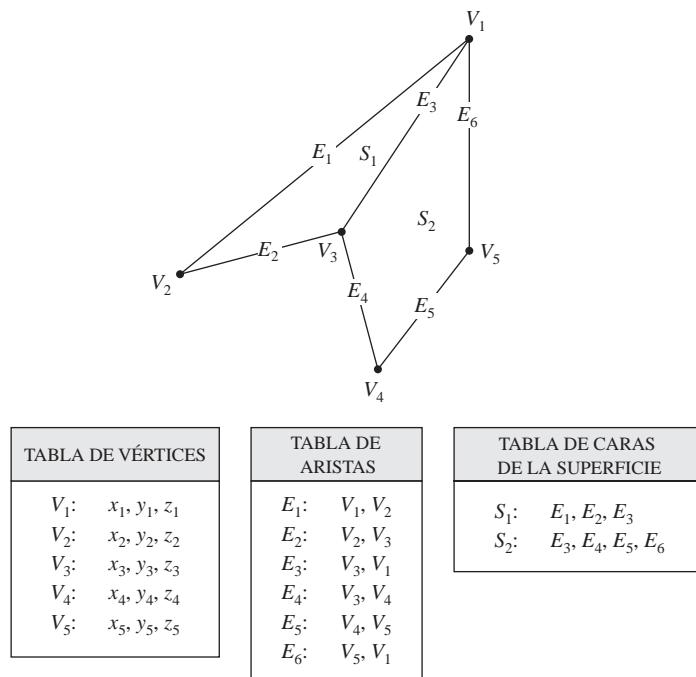
**FIGURA 3.49.** Un área de relleno definida como una región con un valor positivo para el número de vueltas. Este área de relleno es la diferencia,  $A - B$ , de dos regiones, teniendo la región A una dirección de contorno positiva (en sentido contrario a las agujas del reloj) y la región B una dirección de contorno negativa (en el sentido de las agujas del reloj).

## Tablas de polígonos

Normalmente, los objetos de una escena se describen como conjuntos de cara poligonales de superficie. De hecho, los paquetes gráficos suelen proporcionar funciones para definir la forma de una superficie en forma de una malla de parches poligonales. La descripción de cada objeto incluye la información de coordenadas que especifica la geometría de las caras poligonales y otros parámetros de la superficie como el color, la transparencia y las propiedades de reflexión de la luz. A medida que se introduce la información correspondiente a cada polígono, los datos se colocan en tablas que se utilizarán en el subsiguiente procesamiento como visualización y manipulación de los objetos de la escena. Estas tablas de datos de los polígonos pueden organizarse en dos grupos: tablas geométricas y tablas de atributos. Las tablas de datos geométricos contienen coordenadas de los vértices y parámetros para identificar la orientación espacial de las superficies poligonales. La información de atributos de un objeto incluye parámetros que especifican el grado de transparencia del objeto y la reflectividad y características de textura de su superficie.

Los datos geométricos de los objetos de una escena se pueden ordenar cómodamente en tres listas: una tabla de vértices, una tabla de aristas y una tabla de caras de la superficie. Los valores de coordenadas de cada vértice del objeto se almacenan en la tabla de vértices. La tabla de aristas contiene punteros que hacen referencia a la tabla de vértices y que permiten identificar los vértices de cada arista del polígono. Por su parte, la tabla de caras de la superficie contiene punteros que hacen referencia a la tabla de aristas, con el fin de identificar las aristas que definen cada polígono. Este esquema se ilustra en la Figura 3.50 para dos caras poligonales adyacentes de la superficie de un objeto. Además, se pueden asignar a los objetos individuales y a sus caras poligonales componentes unos identificadores de objeto y de cara para poder efectuar las referencias más fácilmente.

Enumerar los datos geométricos en tres tablas, como en la Figura 3.50, permite hacer referencia cómodamente a los componentes individuales (vértices, aristas y caras de la superficie) de cada objeto. Asimismo, el objeto puede visualizarse de manera eficiente utilizando los datos de la tabla de aristas para identificar los con-



**FIGURA 3.50.** Representación en tabla de los datos geométricos para dos caras poligonales adyacentes de una superficie, formadas por seis aristas y cinco vértices.

$E_1:$	$V_1, V_2, S_1$
$E_2:$	$V_2, V_3, S_1$
$E_3:$	$V_3, V_1, S_1, S_2$
$E_4:$	$V_3, V_4, S_2$
$E_5:$	$V_4, V_5, S_2$
$E_6:$	$V_5, V_1, S_2$

**FIGURA 3.51.** Tabla de aristas para las superficies de la Figura 3.50, expandida para incluir punteros que hagan referencia a la tabla de caras de la superficie.

tornos de los polígonos. Otra disposición alternativa consiste en utilizar simplemente dos tablas: una tabla de vértices y una tabla de caras de la superficie, pero este esquema resulta menos conveniente, y algunas de las aristas podrían llegar a dibujarse dos veces en una representación alámbrica. Otra posibilidad consiste en utilizar únicamente una tabla de caras de la superficie, pero con esto se duplica la información de coordenadas, ya que se tendrán que enumerar los valores explícitos de coordenada para cada vértice de cada cara poligonal. Asimismo, sería necesario reconstruir la relación entre aristas y caras a partir de la lista de vértices contenida en la tabla de caras de la superficie.

Podemos añadir información adicional a las tablas de datos de la Figura 3.50 para poder extraer más rápidamente la información. Por ejemplo, podríamos expandir la tabla de aristas para incluir retropunteros que hagan referencia a la tabla de caras de la superficie, con el fin de poder identificar más rápidamente las aristas comunes existentes entre los polígonos (Figura 3.51). Esto resulta particularmente útil para los procedimientos de renderización que necesitan variar con suavidad el sombreado de la superficie al cruzar una arista desde un polígono a otro. De forma similar, la tabla de vértices podría expandirse para hacer referencia a las aristas correspondientes, con el fin de extraer más rápidamente la información.

Entre la información geométrica adicional que suele almacenarse en las tablas de datos se incluyen la pendiente de cada arista y los recuadros de contorno de las aristas de los polígonos, de las caras poligonales y de cada objeto de la escena. A medida que se introducen vértices, podemos calcular las pendientes de las aristas y podemos analizar los valores de las coordenadas para identificar los valores  $x$ ,  $y$  y  $z$  mínimos y máximos para cada línea y polígono individual. Las pendientes de las aristas y la información de los recuadros de contorno son necesarias en el subsiguiente procesamiento, como por ejemplo en la renderización de las superficies y en los algoritmos de identificación de superficies visibles.

Puesto que las tablas de datos geométricos pueden contener listados muy extensos de vértices y aristas para los objetos y escenas más complejos, es importante que se compruebe la coherencia y exhaustividad de los datos. Cuando se especifican las definiciones de los vértices, aristas y polígonos, es posible, particularmente en las aplicaciones interactivas, que se cometan ciertos errores de entrada que pueden llegar a distorsionar la visualización de los objetos. Cuanta mayor información se incluya en las tablas de datos, más fácilmente se podrá comprobar si existen errores. Por tanto, la comprobación de errores es más fácil cuando se utilizan tres tablas de datos (vértices, aristas y caras de la superficie), ya que es este esquema el que proporciona la mayor cantidad de información. Entre las comprobaciones que podría realizar un paquete gráfico están (1) que todo vértice aparece como extremo de al menos dos aristas, (2) que toda arista forma parte de al menos un polígono, (3) que todos los polígonos son cerrados, (4) que cada polígono tiene al menos una arista compartida y (5) que si la tabla de aristas contiene punteros a los polígonos, toda arista referenciada por un puntero de un polígono tiene otro puntero inverso en la tabla de aristas que hace referencia al polígono.

## Ecuaciones de un plano

Para producir una imagen de una escena tridimensional, los sistemas gráficos procesan los datos de entrada, llevando a cabo diversos procedimientos. Entre estos procedimientos se incluyen la transformación de las descripciones de modelado y de las descripciones en coordenadas universales a través de la pipeline de visualización, la identificación de las superficies visibles y la aplicación de rutinas de renderización a cada una de las caras individuales de la superficie. Para algunos de estos procesos, hace falta disponer de información acerca de la orientación espacial de los componentes de la superficie. Esta información se obtiene a partir de

los valores de coordenadas de los vértices y a partir de las ecuaciones que describen las superficies de los polígonos.

Cada polígono en una escena está contenido dentro de un plano de extensión infinita. La ecuación general de un plano es:

$$Ax + By + Cz + D = 0 \quad (3.59)$$

donde  $(x, y, z)$  es cualquier punto del plano y los coeficientes  $A, B, C$  y  $D$  (denominados *parámetros del plano*) son constantes que describen las propiedades espaciales del plano. Podemos obtener los valores de  $A, B, C$  y  $D$  resolviendo un conjunto de tres ecuaciones del plano, utilizando los valores de coordenadas de tres puntos no colineales pertenecientes al plano. Para este propósito, podemos seleccionar tres vértices sucesivos de un polígono convexo,  $(x_1, y_1, z_1), (x_2, y_2, z_2)$  y  $(x_3, y_3, z_3)$ , en sentido contrario a las agujas del reloj y resolver el siguiente sistema de ecuaciones lineales del plano, con el fin de hallar los cocientes  $A/D, B/D$  y  $C/D$ :

$$(A/D)x_k + (B/D)y_k + (C/D)z_k = -1, \quad k = 1, 2, 3 \quad (3.60)$$

La solución de este sistema de ecuaciones puede obtenerse mediante determinantes utilizando la regla de Cramer:

$$\begin{aligned} A &= \begin{vmatrix} 1 & y_1 & z_1 \\ 1 & y_2 & z_2 \\ 1 & y_3 & z_3 \end{vmatrix} & B &= \begin{vmatrix} x_1 & 1 & z_1 \\ x_2 & 1 & z_2 \\ x_3 & 1 & z_3 \end{vmatrix} \\ C &= \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} & D &= -\begin{vmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{vmatrix} \end{aligned} \quad (3.61)$$

Expandiendo los determinantes, podemos escribir los cálculos necesarios para hallar los coeficientes del plano, que tendrán la forma:

$$\begin{aligned} A &= y_1(z_2 - z_3) + y_2(z_3 - z_1) + y_3(z_1 - z_2) \\ B &= z_1(x_2 - x_3) + z_2(x_3 - x_1) + z_3(x_1 - x_2) \\ C &= x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2) \\ D &= -x_1(y_1z_3 - y_3z_1) - x_2(y_3z_1 - y_1z_3) - x_3(y_1z_2 - y_2z_1) \end{aligned} \quad (3.62)$$

Estos cálculos son válidos para cualesquiera tres puntos, incluyendo aquellos para los que  $D = 0$ . Cuando se introducen las coordenadas de los vértices y otras informaciones en la estructura de datos del polígono, pueden calcularse los valores de  $A, B, C$  y  $D$  para cada cara poligonal y almacenarlos con el resto de los datos que definen el polígono.

Es posible que las coordenadas que definen una cara poligonal no estén contenidas dentro de un mismo plano. Podemos resolver este problema dividiendo dicha cara en un conjunto de triángulos, o bien podemos tratar de encontrar un plano aproximador para la lista de vértices. Un método para obtener un plano aproximador consiste en dividir la lista de vértices en subconjuntos de tres vértices y calcular los parámetros del plano  $A, B, C$  y  $D$  para cada subconjunto. Los parámetros del plano aproximador se obtendrán entonces calculando el valor medio de cada uno de los parámetros del plano calculado. Otra técnica consiste en proyectar la lista de vértices sobre los planos de coordenadas. Entonces, asignamos a  $A$  un valor proporcional al área de la proyección poligonal sobre el plano  $yz$ , asignamos al parámetro  $B$  un valor proporcional al área de proyec-

ción sobre el plano  $xz$  y asignamos al parámetro  $C$  un valor proporcional al área de proyección sobre el plano  $xy$ . Este método de proyección se utiliza a menudo en las aplicaciones de trazado de rayos.

## Caras poligonales anteriores y posteriores

Puesto que usualmente tratamos con caras poligonales que encierran un objeto interior, es necesario distinguir entre las dos caras de cada superficie. La cara de un polígono que apunta hacia el interior del objeto se denomina **cara posterior**, mientras que la cara visible es la **cara anterior**. La identificación de la posición de los puntos en el espacio con relación a las caras anterior y posterior de un polígono es una de las tareas básicas que deben llevarse a cabo en muchos algoritmos gráficos, como por ejemplo a la hora de determinar la visibilidad de los objetos. Todo polígono está contenido en un plano infinito que divide el espacio en dos regiones. Todo punto que no se encuentre en el plano y que esté situado del lado de la cara anterior de un polígono se considerará que está *delante* (o *fuera*) del plano y, por tanto, fuera del objeto. Cualquier punto que esté del lado de la cara posterior del polígono se encontrará *detrás* (o *dentro*) del plano. Un plano que esté detrás (dentro) de todos los planos correspondientes a los polígonos de la superficie estará dentro del objeto. Es necesario tener en cuenta que esta clasificación dentro/fuera es relativa al plano que contiene al polígono, mientras que nuestras anteriores comprobaciones de tipo dentro/fuera utilizando las reglas par-impar o del número de vueltas se referían al interior de algún tipo de contorno bidimensional.

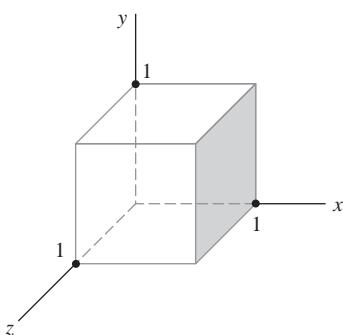
Pueden utilizarse las ecuaciones del plano para identificar la posición de los puntos en el espacio en relación con las caras poligonales de un objeto. Para cualquier punto  $(x, y, z)$  que no se encuentre sobre un cierto plano con parámetros  $A, B, C, D$ , tendremos:

$$Ax + By + Cz + D \neq 0$$

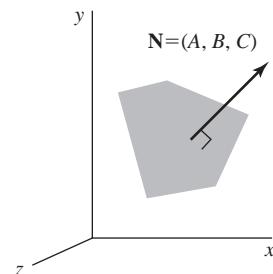
Así, podemos identificar el punto como situado delante o detrás de una superficie poligonal contenida en dicho plano sin más que utilizar el signo (negativo o positivo) de  $Ax + By + Cz + D$ :

- |                             |   |
|-----------------------------|---|
| si $Ax + By + Cz + D < 0$ , | el punto $(x, y, z)$ está detrás del plano  |
| si $Ax + By + Cz + D > 0$ , | el punto $(x, y, z)$ está delante del plano |

Estas comprobaciones de desigualdad son válidas en todo sistema cartesiano que cumpla la regla de la mano derecha, supuesto que los parámetros del plano  $A, B, C$  y  $D$  hayan sido calculados utilizando posiciones de coordenadas seleccionadas estrictamente en el sentido contrario a las agujas del reloj, cuando se mira la superficie desde la parte anterior a la posterior. Por ejemplo, en la Figura 3.52 cualquier punto situado fuera (delante) del plano correspondiente al polígono sombreado satisface la desigualdad  $x - 1 > 0$ , mientras que cualquier punto que esté dentro (detrás) del plano tendrá un valor de la coordenada  $x$  inferior a 1.



**FIGURA 3.52.** La superficie sombreada de este polígono del cubo unitario tiene como ecuación del plano  $x - 1 = 0$ .



**FIGURA 3.53.** El vector normal  $\mathbf{N}$  para un plano descrito por la ecuación  $Ax + By + Cz + D$  es perpendicular al plano y tiene como componentes cartesianas  $(A, B, C)$ .

La orientación de la superficie de un polígono en el espacio puede describirse mediante el **vector normal** del plano que contiene dicho polígono, como se muestra en la Figura 3.53. Este vector normal a la superficies perpendicular al plano y tiene como componentes cartesianas ( $A, B, C$ ), donde los parámetros  $A, B$  y  $C$  son los coeficientes del plano calculados en la Ecuación 3.62. El vector normal apunta en una dirección que va desde el interior del plano hacia el exterior, es decir, desde la cara trasera del polígono hacia la cara delantera.

Como ejemplo de cálculo de las componentes del vector normal de un polígono (lo que también nos da los parámetros del plano) vamos a elegir tres de los vértices de la cara sombreada del cubo unitario de la Figura 3.52. Estos puntos se seleccionan en sentido contrario a las agujas del reloj, según miramos al cubo desde el exterior y al origen de coordenadas. Las coordenadas de estos vértices, en el orden seleccionado, se utilizarán entonces en las Ecuaciones 3.62 para obtener los coeficientes del plano:  $A = 1, B = 0, C = 0, D = -1$ . Así, el vector normal para este plano es  $\mathbf{N} = (1, 0, 0)$  que está en la dirección del eje  $x$  positivo. Es decir, el vector normal apunta desde el interior hacia el exterior y es perpendicular al plano  $x = 1$ .

Los elementos del vector normal también pueden obtenerse utilizando el producto vectorial. Suponiendo que tengamos una cara superficial poligonal convexa y un sistema cartesiano que cumpla con la regla de la mano derecha, seleccionemos de nuevo cualesquiera tres posiciones de vértices,  $\mathbf{V}_1, \mathbf{V}_2$  y  $\mathbf{V}_3$ , tomadas en sentido contrario a las agujas del reloj, cuando se mira desde el exterior del objeto hacia el interior. Formando dos vectores, uno de  $\mathbf{V}_1$  a  $\mathbf{V}_2$  y el segundo de  $\mathbf{V}_1$  a  $\mathbf{V}_3$ , podemos calcular  $\mathbf{N}$  mediante el producto vectorial:

$$\mathbf{N} = (\mathbf{V}_2 - \mathbf{V}_1) \times (\mathbf{V}_3 - \mathbf{V}_1) \quad (3.63)$$

Esto genera los valores para los parámetros del plano  $A, B$  y  $C$ . Entonces podemos obtener el valor del parámetro  $D$  sustituyendo estos valores y las coordenadas de uno de los vértices del polígono en la Ecuación 3.59 del plano y despejando  $D$ . La ecuación del plano puede expresarse en forma vectorial utilizando la normal  $\mathbf{N}$  y el vector de posición  $\mathbf{P}$  de cualquier punto del plano, de la forma siguiente:

$$\mathbf{N} \cdot \mathbf{P} = -D \quad (3.64)$$

Para un polígono convexo, también podríamos obtener los parámetros del plano utilizando el producto vectorial de dos vectores representativos de aristas sucesivas. Y con un polígono cóncavo, podemos seleccionar los tres vértices de modo que los dos vectores que participen en el producto vectorial formen un ángulo inferior a  $180^\circ$ . En caso contrario, podemos tomar el negado de su producto vectorial para obtener la dirección correcta del vector normal correspondiente a la superficie poligonal.

## 3.16 FUNCIONES OpenGL DE RELLENO DE ÁREAS POLIGONALES

---

Con una sola excepción, los procedimientos OpenGL para especificar polígonos llenos son similares a los que se utilizan para describir un punto o una polilínea. Se emplea una función `glVertex` para introducir las coordenadas para un único vértice del polígono y el polígono completo se describe mediante una lista de vértices encerrada entre una pareja de funciones `glBegin/glEnd`. Sin embargo, hay una función adicional que podemos utilizar para mostrar un rectángulo y que tiene un formato completamente distinto.

De manera predeterminada, el interior de un polígono se muestra con un color homogéneo, que estará determinado por las configuraciones actuales de color. Como opciones (que se describen en el siguiente capítulo), podemos llenar un polígono con un patrón y mostrar las aristas del polígono mediante líneas que se dibujan alrededor del relleno interior. Hay seis constantes simbólicas diferentes que podemos utilizar como argumento de la función `glBegin` para describir las áreas de relleno poligonales. Estas seis constantes primitivas nos permiten mostrar un único polígono llenado, un conjunto de polígonos llenados desconectados o un conjunto de polígonos llenados conectados.

En OpenGL, las áreas de relleno deben especificarse como polígonos convexos. Por tanto, la lista de vértices para un polígono llenado deberá contener al menos tres vértices, no podrán existir aristas que se cor-

ten y todos los ángulos interiores del polígono deben ser inferiores a  $180^\circ$ . Además, cada área de relleno poligonal sólo puede definirse mediante una única lista de vértices, lo que prohíbe las especificaciones que contengan agujeros en el interior del polígono, como la que se muestra en la Figura 3.54. Para describir esta figura tendríamos que utilizar dos polígonos convexos solapados.

Cada polígono que especifiquemos tendrá dos caras: una cara anterior y una cara posterior. En OpenGL, el color de relleno y otros atributos pueden configurarse de manera independiente para cada cara y es necesario efectuar una identificación anterior/posterior tanto en las rutinas de visualización bidimensionales como en las tridimensionales. Por tanto, los vértices del polígono deben especificarse en sentido contrario a las agujas del reloj, según miramos hacia el polígono desde el “exterior”. Esto identifica la cara anterior de dicho polígono.

Puesto que las imágenes gráficas incluyen a menudo áreas rectangulares llenas, OpenGL proporciona una función especial de generación de rectángulos que acepta directamente especificaciones de vértices en el plano  $xy$ . En algunas implementaciones de OpenGL, la siguiente rutina puede ser más eficiente que generar un rectángulo lleno mediante especificaciones `glVertex`.

```
glRect* (x1, y1, x2, y2);
```

Una de las esquinas de este rectángulo se encuentra en las coordenadas  $(x1, y1)$  y la esquina opuesta en la posición  $(x2, y2)$ . Una serie de códigos de sufijo para `glRect` especifican el tipo de datos de las coordenadas y si éstas están expresadas como elementos de matriz. Dichos códigos son `i` (para integer), `s` (para short), `f` (para float), `d` (para double) y `v` (para vectores). El rectángulo se muestra con sus aristas paralelas a los ejes de coordenadas  $xy$ . Como ejemplo, la siguiente instrucción define el cuadrado que se muestra en la Figura 3.55.

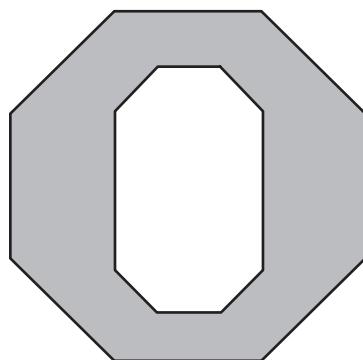
```
glRecti (200, 100, 50, 250);
```

Si introducimos los valores de coordenadas para este rectángulo en matrices, podemos generar el mismo cuadrado mediante el siguiente código.

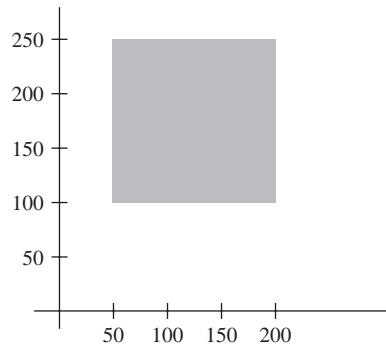
```
int vertex1 [ ] = {200, 100};
int vertex2 [ ] = {50, 250};

glRectiv (vertex1, vertex2);
```

Cuando se genera un rectángulo mediante la función `glRect`, las aristas del polígono se forman entre los vértices en el orden  $(x1, y1), (x2, y1), (x2, y2), (x1, y2)$ , volviéndose después al primer vértice. Así, en nuestro ejemplo, hemos generado una lista de vértices en el sentido de las agujas del reloj. En muchas aplicaciones bidimensionales, la determinación de las caras anterior y posterior no es importante, pero si queremos



**FIGURA 3.54.** Un polígono con un interior complejo, que no puede especificarse mediante una única lista de vértices.



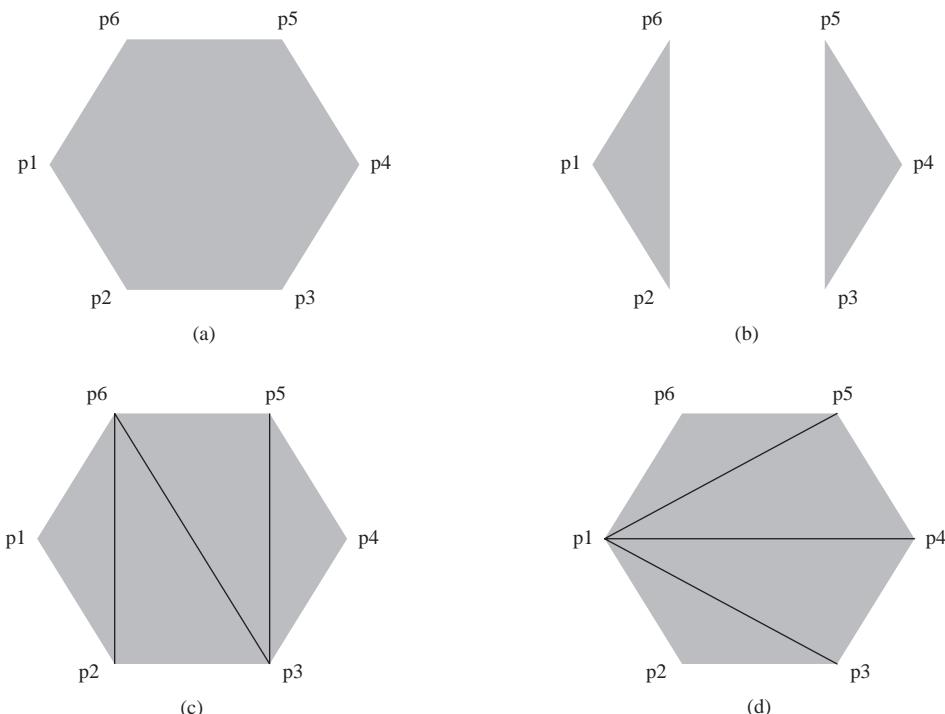
**FIGURA 3.55.** Visualización de un área cuadrada rellena utilizando la función `glRect`.

asignar propiedades diferentes a las caras anterior y posterior del rectángulo, entonces tendremos que invertir el orden de los dos vértices de este ejemplo, para obtener una ordenación de los vértices en sentido contrario a las agujas del reloj. En el Capítulo 4 veremos otra manera de invertir la especificación de las caras anterior y posterior del polígono.

Todas las otras seis primitivas OpenGL de relleno de polígonos se especifican utilizando una constante simbólica en la función `glBegin`, junto con una lista de comandos `glVertex`. Con la constante primitiva OpenGL `GL_POLYGON`, podemos mostrar una única área de relleno poligonal como la que se presenta en la Figura 3.56(a). Para este ejemplo, suponemos que tenemos una lista de seis puntos, etiquetados como  $p_1$  a  $p_6$ , los cuales especifican posiciones bidimensionales de los vértices del polígono en sentido contrario a las agujas del reloj. Cada uno de los puntos se representa como una matriz de valores de coordenadas ( $x, y$ ).

```
glBegin (GL_POLYGON);
    glVertex2iv (p1);
    glVertex2iv (p2);
    glVertex2iv (p3);
    glVertex2iv (p4);
    glVertex2iv (p5);
    glVertex2iv (p6);
glEnd ( );
```

La lista de vértices del polígono debe contener al menos tres vértices. En caso contrario, no se mostrará nada en la imagen.



**FIGURA 3.56.** Visualización de áreas de relleno poligonales utilizando una lista de seis posiciones de vértices. (a) Un área de relleno formada por un único polígono convexo generado por la constante primitiva `GL_POLYGON`. (b) Dos triángulos desconectados generados con `GL_TRIANGLES`. (c) Cuatro triángulos conectados generados con `GL_TRIANGLE_STRIP`. (d) Cuatro triángulos conectados generados con `GL_TRIANGLE_FAN`.

Si reordenamos la lista de vértices y cambiamos la constante primitiva del ejemplo de código anterior a `GL_TRIANGLES`, obtenemos dos áreas de relleno triangulares separadas, las cuales se muestran en la Figura 3.56(b).

```
glBegin (GL_TRIANGLES);
    glVertex2iv (p1);
    glVertex2iv (p2);
    glVertex2iv (p6);
    glVertex2iv (p3);
    glVertex2iv (p4);
    glVertex2iv (p5);
glEnd ( );
```

En este caso, los primeros tres puntos definen los vértices de un triángulo, los siguientes tres puntos definen el siguiente triángulo, etc. Para cada área de relleno triangular, especificamos las posiciones de los vértices en sentido contrario a las agujas del reloj. Con esta constante primitiva se mostrará un conjunto de triángulos desconectados, a menos que se repitan las coordenadas de algunos vértices. Si no incluimos al menos tres vértices, no se mostrará nada en la imagen, y si el número de vértices especificados no es un múltiplo de tres, el vértice o los dos vértices finales no se utilizarán.

Reordenando de nuevo la lista de vértices y cambiando la constante primitiva a `GL_TRIANGLE_STRIP`, podemos mostrar el conjunto de triángulos conectados que se presenta en la Figura 3.56(c).

```
glBegin (GL_TRIANGLE_STRIP);
    glVertex2iv (p1);
    glVertex2iv (p2);
    glVertex2iv (p6);
    glVertex2iv (p3);
    glVertex2iv (p5);
    glVertex2iv (p4);
glEnd ( );
```

Suponiendo que no se repita ninguno de los  $N$  vértices de la lista, obtendremos  $N - 2$  triángulos en la banda de triángulos dibujada por este comando. Obviamente, debemos tener  $N \geq 3$  o no se podrá mostrar ninguna imagen. En este ejemplo,  $N = 6$  y se obtienen cuatro triángulos. Cada uno de los sucesivos triángulos comparte una arista con el triángulo anteriormente definido, por lo que la ordenación de la lista de vértices debe ser la adecuada para garantizar una visualización coherente. Se definirá un triángulo para cada vértice enumerado después de los dos primeros vértices. Así, los primeros tres vértices deben enumerarse en el sentido contrario a las agujas del reloj, según se mira a la cara frontal (exterior) del triángulo. Después de eso, el conjunto de tres vértices para cada triángulo subsiguiente estará dispuesto en sentido contrario a las agujas del reloj dentro de la tabla de polígonos. Esto se lleva a cabo procesando cada posición  $n$  de la lista de vértices en el orden  $n = 1, n = 2, \dots, n = N - 2$  y disponiendo el orden del correspondiente conjunto de tres vértices según  $n$  sea un número par o impar. Si  $n$  es impar, la entrada de la tabla de polígonos para los vértices del triángulo estará en el orden  $n, n + 1, n + 2$ . Si  $n$  es par, los vértices del triángulo se enumerarán en el orden  $n + 1, n + 2, n$ . En el ejemplo anterior, nuestro primer triángulo ( $n = 1$ ) se describirá como compuesto por los vértices (p1, p2, p6). El segundo triángulo ( $n = 2$ ) tendrá la ordenación de vértices (p6, p2, p3). El orden de los vértices para el tercer triángulo ( $n = 3$ ) será (p6, p3, p5) y el cuarto triángulo ( $n = 4$ ) aparecerá en la tabla de polígonos con la ordenación de vértices (p5, p3, p4).

Otra forma de generar un conjunto de triángulos conectados consiste en utilizar la estructura de «ventilador» ilustrada en la Figura 3.56(d), donde todos los triángulos comparten un vértice común. Esta disposición de triángulos se obtiene utilizando la constante primitiva `GL_TRIANGLE_FAN` y la ordenación original de los seis vértices:

```

glBegin (GL_TRIANGLE_FAN);
    glVertex2iv (p1);
    glVertex2iv (p2);
    glVertex2iv (p3);
    glVertex2iv (p4);
    glVertex2iv (p5);
    glVertex2iv (p6);
glEnd ( );

```

Para  $N$  vértices, obtendremos de nuevo  $N - 2$ , supuesto que no se repita ninguna posición de vértice y que hayamos incluido al menos tres vértices. Además, los vértices deben especificarse en el orden correcto para poder definir adecuadamente las caras anterior y posterior de cada triángulo. El primer vértice enumerado (en este caso, p1) formará parte de cada uno de los triángulos de la estructura en ventilador. Si volvemos a enumerar los triángulos y los vértices como  $n = 1, n = 2, \dots, n = N - 2$ , entonces los vértices para el triángulo  $n$  aparecerán en la tabla de polígonos en el orden  $1, n + 1, n + 2$ . Por tanto, el triángulo 1 estará definido mediante la lista de vértices (p1, p2, p3); el triángulo 2 tendrá la ordenación de vértices (p1, p3, p4); el triángulo 3 tendrá sus vértices especificados en el orden (p1, p4, p5) y el triángulo 4 estará descrito mediante la ordenación de vértices (p1, p5, p6).

Además de las funciones primitivas para triángulos y para la visualización de un polígono general, OpenGL proporciona mecanismos para especificar dos tipos de cuadriláteros (polígonos de cuatro lados). Con la constante primitiva `GL_QUADS` y la siguiente lista de ocho vértices, especificados mediante matrices de coordenadas bidimensionales, podemos generar la imagen que se muestra en la Figura 3.57(a).

```

glBegin (GL_QUADS);
    glVertex2iv (p1);
    glVertex2iv (p2);
    glVertex2iv (p3);
    glVertex2iv (p4);
    glVertex2iv (p5);
    glVertex2iv (p6);
    glVertex2iv (p7);
    glVertex2iv (p8);
glEnd ( );

```

Los primeros cuatro vértices definen las coordenadas del primer cuadrilátero, mientras que los siguientes cuatro puntos definen el segundo cuadrilátero, etc. Para cada cuadrilátero relleno, se especifican las posiciones de los vértices en sentido contrario a las agujas del reloj. Si no se repiten las coordenadas de ningún vértice, se mostrará con esto un conjunto de áreas de relleno de cuatro lados no conectadas. Con esta primitiva, es necesario proporcionar al menos cuatro vértices, ya que de lo contrario no se mostrará ninguna imagen; además, si el número de vértices especificado no es múltiplo de cuatro, los vértices sobran se ignorarán.

Reordenando la lista de vértices del ejemplo de código anterior y cambiando la constante primitiva a `GL_QUAD_STRIP`, podemos obtener el conjunto de cuadriláteros conectados que se muestran en la Figura 3.57(b).

```

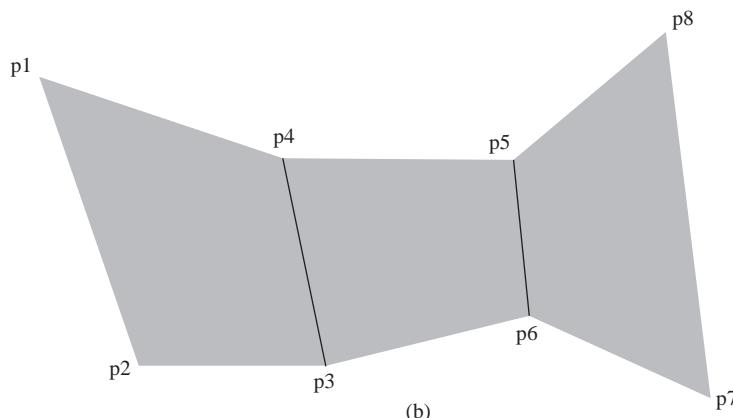
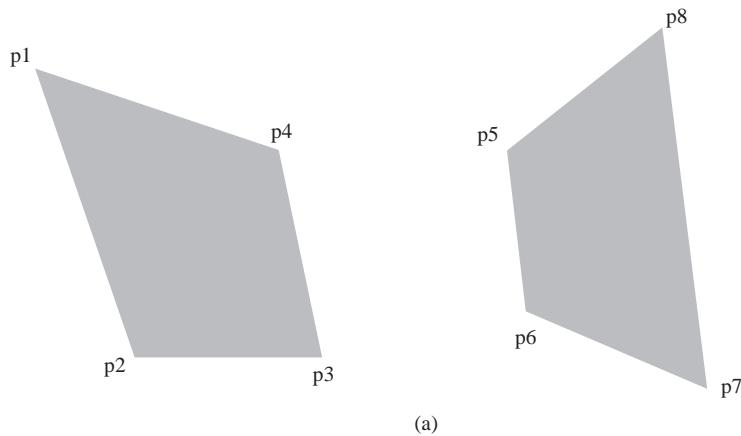
glBegin (GL_QUAD_STRIP);
    glVertex2iv (p1);
    glVertex2iv (p2);
    glVertex2iv (p4);
    glVertex2iv (p3);
    glVertex2iv (p5);
    glVertex2iv (p6);
    glVertex2iv (p8);

```

```
glVertex2iv (p7);
glEnd ( );
```

Se dibujará un cuadrilátero para cada par de vértices especificados después de los primeros dos vértices de la lista, y es necesario enumerar los vértices en el orden correcto para poder generar una ordenación de vértices en sentido contrario a las agujas del reloj para cada polígono. Para una lista de  $N$  vértices, obtendremos  $\frac{N}{2}-1$  cuadriláteros, supuesto que  $N \geq 4$ . Si  $N$  no es un múltiplo de cuatro, los vértices que sobren en la lista no se utilizarán. Podemos enumerar los polígonos rellenos y los vértices de la lista como  $n = 1, n = 2, \dots, n = \frac{N}{2}-1$ . Entonces las tablas de polígonos enumerarán los vértices para el cuadrilátero  $n$  con el orden  $2n-1, 2n, 2n+2, 2n+1$ . Para este ejemplo,  $N = 8$  y tendremos tres cuadriláteros en la banda de cuadriláteros. Así, nuestro primer cuadrilátero ( $n = 1$ ) tendrá la ordenación de vértices  $(p1, p2, p3, p4)$ . El segundo cuadrilátero ( $n = 2$ ) tendrá la ordenación de vértices  $(p4, p3, p6, p5)$  y la ordenación de vértices para el tercer cuadrilátero ( $n = 3$ ) será  $(p5, p6, p7, p8)$ .

La mayoría de los paquetes gráficos muestran las superficies curvadas mediante un conjunto de caras planas que las aproximan. Esto se debe a que las ecuaciones de los planos son lineales y el procesamiento de esas ecuaciones lineales es mucho más rápido que procesar cuádricas u otros tipos de ecuaciones de curvas. Por eso, OpenGL y otros paquetes proporcionan primitivas poligonales para facilitar la aproximación de una superficie curva. Los objetos se modelan mediante mallas de polígonos y se define una base de datos de información geométrica y de atributos para facilitar el procesamiento de las caras poligonales. En OpenGL, las



**FIGURA 3.57.** Visualización de cuadriláteros rellenos utilizando una lista de ocho vértices. (a) Dos cuadriláteros no conectados generados mediante `GL_QUADS`. (b) Tres cuadriláteros conectados generados mediante `GL_QUAD_STRIP`.

primitivas que podemos utilizar con este propósito son la *banda de triángulos*, el *ventilador de triángulos* y la *banda de cuadriláteros*. Los sistemas gráficos de alta calidad incorporan renderizadores rápidos de polígonos implementados en hardware que tienen la capacidad de mostrar un millón o más de polígonos sombreados por segundo (usualmente triángulos), incluyendo la aplicación de texturas superficiales y de efectos especiales de iluminación.

Aunque la biblioteca básica de OpenGL sólo permite utilizar polígonos convexos, la utilidad de OpenGL (GLU) proporciona funciones para procesar polígonos cóncavos y otros objetos no convexos con contornos lineales. Hay disponible una serie de rutinas de *teselado de polígonos* en GLU para convertir dichas formas en un conjunto de triángulos, mallas de triángulos, ventiladores de triángulos y segmentos de líneas rectas. Una vez descompuestos esos objetos, puede procesárselos mediante funciones OpenGL básicas.

## 3.17 MATRICES DE VÉRTICES OpenGL

---

Aunque los ejemplos que hemos utilizado hasta el momento contenían solamente unos pocos vértices, describir una escena que contenga numerosos objetos puede ser mucho más complicado. Como ilustración, vamos a considerar primero cómo podríamos describir un único objeto muy básico: el cubo de lado unidad que se muestra en la Figura 3.58, cuyas coordenadas se proporcionan como números enteros con el fin de simplificar las explicaciones. Un método directo para definir las coordenadas de los vértices consistiría en utilizar una matriz de dos dimensiones, como:

```
GLint points [8][3] = { {0, 0, 0}, {0, 1, 0}, {1, 0, 0}, {1, 1, 0},
                        {0, 0, 1}, {0, 1, 1}, {1, 0, 1}, {1, 1, 1} };
```

También podríamos definir primero un tipo de datos para las coordenadas tridimensionales de los vértices y luego proporcionar las coordenadas de cada vértice como un elemento de una matriz unidimensional, como por ejemplo:

```
typedef GLint vertex3 [3];
vertex3 pt [8] = { {0, 0, 0}, {0, 1, 0}, {1, 0, 0}, {1, 1, 0},
                   {0, 0, 1}, {0, 1, 1}, {1, 0, 1}, {1, 1, 1} };
```

A continuación, necesitamos definir cada una de las seis caras del objeto. Para esto, podemos hacer seis llamadas a `glBegin(GL_POLYGON)` o a `glBegin(GL_QUADS)`. En cualquiera de los casos, debemos asegurarnos de enumerar los vértices de cada cara en sentido contrario a las agujas del reloj, según se mira a la superficie desde el exterior del cubo. En el siguiente segmento de código, especificamos cada cara del cubo como un cuadrilátero y utilizamos una llamada a función para pasar los valores de subíndice de la matriz a las rutinas primitivas OpenGL. La Figura 3.59 muestra los valores de los subíndices de la matriz `pt` que define las posiciones de los vértices del cubo.

```
void quad (GLint n1, GLint n2, GLint n3, GLint n4)
{
    glBegin (GL_QUADS);
    glVertex3iv (pt [n1]);
    glVertex3iv (pt [n2]);
    glVertex3iv (pt [n3]);
    glVertex3iv (pt [n4]);
    glEnd ( );
}

void cube ( )
{
    quad (6, 2, 3, 7);
```

```

quad (5, 1, 0, 4);
quad (7, 3, 1, 5);
quad (4, 0, 2, 6);
quad (2, 0, 1, 3);
quad (7, 5, 4, 6);
}

```

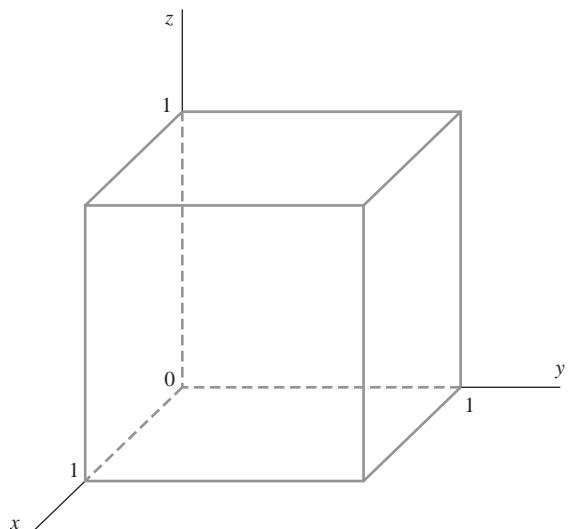
Así, la especificación de cada cara requiere seis funciones OpenGL y tenemos seis caras que especificar. Cuando se añaden las especificaciones de color y de otros parámetros, el programa de visualización del cubo puede alcanzar fácilmente el centenar de llamadas a funciones OpenGL, o incluso más. Y las escenas con muchos objetos complejos requerirán, obviamente, un número de llamadas muy superior.

Como podemos ver a partir del anterior ejemplo del cubo, la descripción completa de una escena podría requerir cientos o miles de especificaciones de coordenadas. Además, es necesario configurar diversos atributos y parámetros de visualización para cada objeto individual. Como consecuencia, las descripciones de los objetos y de la escena podrían requerir una cantidad enorme de llamadas a función, lo que impone una gran carga de procesamiento al sistema y puede ralentizar la ejecución de los programas gráficos. Un problema adicional que presentan las imágenes complejas es que las superficies de los objetos (como por ejemplo el cubo de la Figura 3.58) usualmente tienen coordenadas de vértices compartidas. Utilizando los métodos de los que hasta ahora hemos hablado, estas posiciones compartidas podrían tener que especificarse múltiples veces.

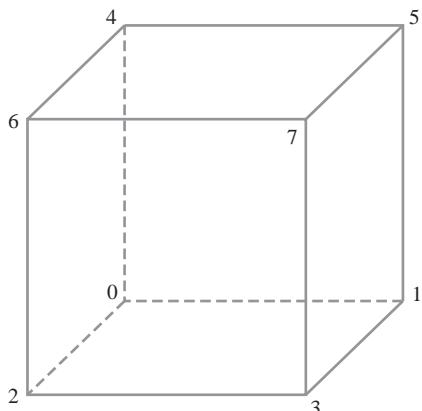
Para aliviar estos problemas, OpenGL proporciona un mecanismo para reducir el número de llamadas a función necesarias para procesar la información de coordenadas. Utilizando una **matriz de vértices** podemos disponer la información que describe una escena de modo que sólo sea necesario efectuar unas pocas llamadas a función. Los pasos necesarios son los siguientes:

- (1) Invocar la función `glEnableClientState (GL_VERTEX_ARRAY)` para activar la característica de matrices de vértices de OpenGL.
- (2) Utilizar la función `glVertexPointer` para especificar la ubicación y el formato de los datos que describen las coordenadas de los vértices.
- (3) Visualizar la escena utilizando una rutina tal como `glDrawElements`, que permite procesar múltiples primitivas con muy pocas llamadas a función.

Utilizando la matriz `pt` definida anteriormente para el cubo, implementaríamos estos tres pasos tal como se indica en el siguiente ejemplo de código:



**FIGURA 3.58.** Un cubo con arista de longitud igual a 1.



**FIGURA 3.59.** Valores de los subíndices para la matriz `pt` correspondiente a las coordenadas de los vértices del cubo mostrado en la Figura 3.58.

```
glEnableClientState (GL_VERTEX_ARRAY);
glVertexPointer (3, GL_INT, 0, pt);

GLubyte vertIndex [ ] = {6, 2, 3, 7, 5, 1, 0, 4, 7, 3, 1, 5,
4, 0, 2, 6, 2, 0, 1, 3, 7, 5, 4, 6};

glDrawElements (GL_QUADS, 24, GL_UNSIGNED_BYTE, vertIndex);
```

Con el primer comando, `glEnableClientState (GL_VERTEX_ARRAY)`, activamos una determinada característica (en este caso, una matriz de vértice) en el lado cliente de un sistema cliente-servidor. Puesto que el cliente (la máquina que está ejecutando el programa principal) es quien guarda los datos correspondientes a la imagen, la matriz de vértices también debe almacenarse allí. Como hemos indicado en el Capítulo 2, el servidor (nuestra estación de trabajo, por ejemplo) se encarga de generar los comandos y visualizar la imagen. Por supuesto, una misma máquina puede actuar a la vez como cliente y como servidor. La característica de matrices de vértices de OpenGL se desactiva mediante el comando:

```
glDisableClientState (GL_VERTEX_ARRAY);
```

A continuación indicamos la ubicación y el formato de las coordenadas de los vértices del objeto, en la función `glVertexPointer`. El primer parámetro de `glVertexPointer`, que es 3 en este ejemplo, especifica el número de coordenadas utilizadas para la descripción de cada vértice. El tipo de datos de las coordenadas de los vértices se designa utilizando una constante simbólica OpenGL como segundo parámetro de esta función. En nuestro ejemplo, el tipo de datos es `GL_INT`. El resto de los tipos de datos se especifican mediante las constantes simbólicas `GL_BYTE`, `GL_SHORT`, `GL_FLOAT` y `GL_DOUBLE`. Con el tercer parámetro indicamos el desplazamiento en bytes entre vértices consecutivos. El propósito de este argumento es permitir utilizar diversos tipos de datos, empaquetando en una única matriz información, por ejemplo, tanto de coordenadas como de colores. Puesto que sólo estamos proporcionando los datos de coordenadas, asignamos un valor de 0 al parámetro de desplazamiento. El último parámetro de la función `glVertexPointer` hace referencia a la matriz de vértices, que contiene los valores de las coordenadas.

Todos los índices de los vértices del cubo se almacenan en la matriz `vertIndex`. Cada uno de estos índices es el subíndice de la matriz `pt` correspondiente a los valores de coordenadas del respectivo vértice. Esta lista de índices se referencia como último parámetro en la función `glDrawElements` y es utilizada a continuación por la primitiva `GL_QUADS`, que es el primer parámetro, con el fin de mostrar el conjunto de cuadriláteros que componen el cubo. El segundo parámetro especifica el número de elementos de la matriz `vertIndex`. Puesto que un cuadrilátero requiere sólo cuatro vértices y hemos especificado 24, la función `glDrawElements` continuará mostrando caras del cubo adicionales para cada sucesivo conjunto de cuatro vértices, hasta haber procesado los 24 vértices disponibles. Con esto, conseguimos visualizar todas las caras

del cubo con una única llamada a función. El tercer parámetro de la función `glDrawElements` proporciona el tipo de los valores de índice. Puesto que nuestros índices son enteros de pequeño tamaño, hemos especificado el tipo `GL_UNSIGNED_BYTE`. Los otros dos tipos de índice que pueden utilizarse son `GL_UNSIGNED_SHORT` y `GL_UNSIGNED_INT`.

Puede combinarse información adicional con los valores de coordenadas en las matrices de vértices, con el fin de facilitar el procesamiento de la descripción de una escena. Podemos especificar valores de color y otros atributos para los objetos en una serie de matrices a las que se puede hacer referencia en la función `glDrawElements`. Y también podemos entrelazar las diversas matrices para aumentar la eficiencia. En el siguiente capítulo analizaremos los métodos disponibles para implementar estas matrices de atributos.

## 3.18 PRIMITIVAS DE MATRICES DE PÍXELES

---

Además de líneas rectas, polígonos, círculos y otras primitivas, los paquetes gráficos suministran a menudo rutinas para mostrar formas definidas mediante matrices rectangulares de valores de color. Podemos obtener esa cuadrícula rectangular digitalizando (escaneando) una fotografía u otra imagen, o generando una imagen con un programa gráfico. Cada valor de color de la matriz se asigna entonces a una o más posiciones de píxel en la pantalla. Como hemos indicado en el Capítulo 2, una matriz de píxeles con valores de color se denomina normalmente *mapa de pixeles*.

Los parámetros para una matriz de píxeles pueden incluir un puntero a la matriz de colores, el tamaño de la matriz y la posición y tamaño del área de la pantalla a la que hay que aplicar los valores de color. La Figura 3.60 proporciona un ejemplo de asignación de una matriz de colores de píxeles a un área de la pantalla.

Otro método para implementar una matriz de píxeles consiste en asignar el valor de bit 0 o el valor de bit 1 a cada elemento de la matriz. En este caso, la matriz es simplemente un *mapa de bits*, que también se denomina en ocasiones *máscara* y que indica si hay que asignar (o combinar) o no un cierto píxel con un color preestablecido.

## 3.19 FUNCIONES OpenGL PARA MATRICES DE PÍXELES

---

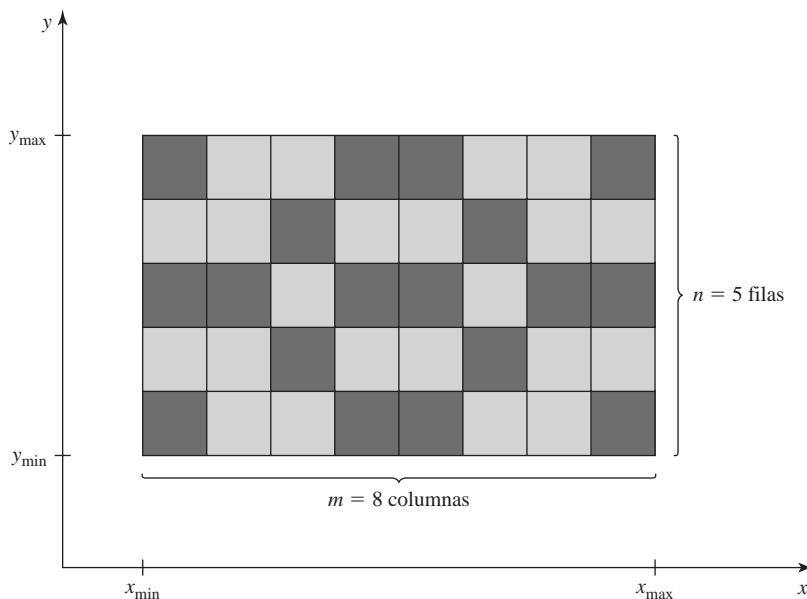
Hay dos funciones de OpenGL que podemos utilizar para definir una forma o patrón especificados mediante una matriz rectangular. Una de ellas sirve para procesar mapas de bits, mientras que la otra sirve para procesar mapas de píxeles. Asimismo, OpenGL proporciona diversas rutinas para guardar, copiar y manipular matrices de valores de píxeles.

### Función de mapa de bits de OpenGL

Un patrón de matriz binaria se define mediante la función:

```
glBitmap (width, height, x0, y0, xOffset, yOffset, bitShape);
```

Los parámetros `width` y `height` de esta función proporcionan el número de columnas y de filas, respectivamente, en la matriz `bitShape`. A cada elemento de `bitShape` se le asigna un 1 o un 0. Un valor de 1 indica que hay que mostrar el píxel correspondiente en un determinado color preestablecido. En caso contrario, el píxel no se verá afectado por el mapa de bits (como opción, podríamos utilizar un valor de 1 para indicar que hay que combinar un color especificado con el valor de color almacenado en dicha posición del búfer de refresco). Los parámetros `x0` e `y0` definen la posición que hay que considerar como «origen» de la matriz rectangular. Esta posición de origen se especifica en relación con la esquina inferior izquierda de `bitShape`, y los valores de `x0` e `y0` pueden ser positivos o negativos. Además, tenemos que designar una ubicación dentro del búfer de imagen a la que haya que aplicar el patrón. Esta ubicación se denomina **posición actual de visualización** y el mapa de bits se mostrará posicionando su origen (`x0`, `y0`) en la posición actual de visualización.



**FIGURA 3.60.** Asignación de una matriz de colores  $n$  por  $m$  a una región de las coordenadas de pantalla.

zación. Los valores asignados a los parámetros `xOffset` e `yOffset` se utilizan como desplazamiento de coordenadas para actualizar la posición actual de visualización del búfer de imagen después de mostrar el mapa de bits.

Los valores de coordenadas para `x0`, `y0`, `xOffset` e `yOffset`, así como la posición actual de visualización, se mantienen como valores en coma flotante. Por supuesto, los mapas de bit se aplicarán a posiciones de píxel enteras, pero las coordenadas en coma flotante permiten espaciar un conjunto de mapas de bits a intervalos arbitrarios, lo que resulta útil en algunas aplicaciones, como por ejemplo al formar cadenas de caracteres mediante patrones de mapas de bits.

Se utiliza la siguiente rutina para establecer las coordenadas de la posición actual de visualización:

```
glRasterPos* ( )
```

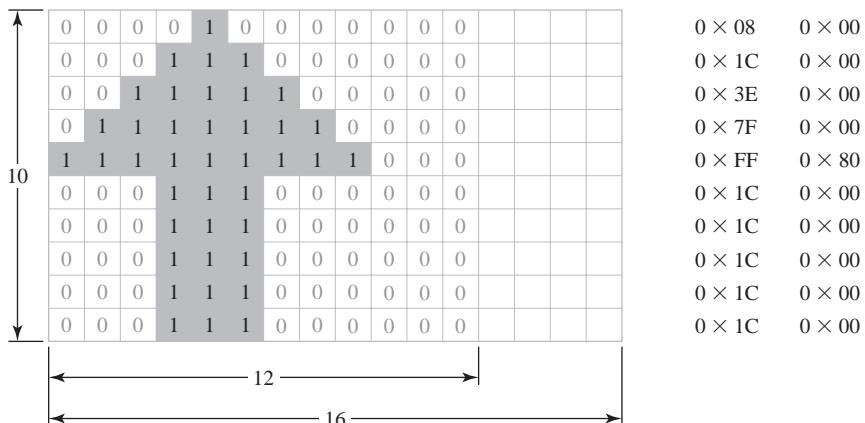
Los parámetros y códigos de sufijos son iguales que para la función `glVertex`. Así, la posición actual de visualización se proporciona en coordenadas universales y se transforma a coordenadas de pantalla mediante las transformaciones de visualización. Para nuestros ejemplos bidimensionales, podemos especificar coordenadas para la posición actual de visualización directamente en coordenadas enteras de pantalla. El valor predeterminado para la posición actual de visualización es el origen de las coordenadas universales (0, 0, 0).

El color para un mapa de bits es el color que esté activo en el momento de invocar el comando `glRasterPos`. Cualesquiera cambios de color subsiguientes no afectarán al mapa de bits.

Cada fila de una matriz de bits rectangular se almacena en múltiplos de 8 bits, disponiendo los datos binarios como conjuntos de caracteres de 8 bits sin signo. Pero podemos describir una forma geométrica utilizando cualquier tamaño de cuadrícula que nos resulte conveniente. Como ejemplo, la Figura 3.61 muestra un patrón de bits definido sobre una cuadrícula de 10 filas por 9 columnas, en la que los datos binarios se especifican utilizando 16 bits para cada fila. Cuando se aplica este patrón a los píxeles del búfer de imagen, todos los valores de bit situados más allá de la novena columna serán ignorados.

Para aplicar el patrón de bits de la Figura 3.61 a una ubicación del búfer de imagen, emplearíamos la siguiente sección de código:

```
GLubyte bitShape [20] = {
    0x1c, 0x00, 0x1c, 0x00, 0x1c, 0x00, 0x1c, 0x00, 0x1c, 0x00,
    0xff, 0x80, 0x7f, 0x00, 0x3e, 0x00, 0x1c, 0x00, 0x08, 0x00};
```



**FIGURA 3.61.** Un patrón de bits especificado como una matriz con 10 filas y 9 columnas y almacenado en bloques de 8 bits para las 10 filas, con 16 valores de bit por cada una de ellas.

```
glPixelStorei (GL_UNPACK_ALIGNMENT, 1); // Establece el modo de almacenamiento
                                         // de píxel.

glRasterPos2i (30, 40);
glBitmap (9, 10, 0.0, 0.0, 20.0, 15.0, bitShape);
```

Los valores de la matriz `bitShape` se especifican fila a fila, comenzando por la parte inferior del patrón de cuadrícula rectangular. A continuación, definimos el modo de almacenamiento para el mapa de bits mediante la rutina OpenGL `glPixelStorei`. El valor de parámetro de 1 en esta función indica que hay que alinear los valores de los datos en las fronteras de los bytes. Con `glRasterPos`, establecemos la posición actual de visualización en las coordenadas (30, 40). Finalmente, la función `glBitmap` especifica que el patrón de bits se proporciona en la matriz `bitShape`, y que esta matriz tiene 9 columnas y 10 filas. Las coordenadas del origen de este patrón son (0.0, 0.0) que se corresponden con la esquina inferior izquierda de la cuadrícula. Hemos ilustrado en el ejemplo un desplazamiento de coordenadas con los valores (20.0, 15.0), aunque en el ejemplo no se hace uso de dicho desplazamiento.

## Función OpenGL para mapas de píxeles

Para aplicar a un bloque del búfer de imagen un patrón definido mediante una matriz de valores de color se utiliza la función:

```
glDrawPixels (width, height, dataFormat, dataType, pixMap);
```

De nuevo los parámetros `width` y `height` proporcionan las dimensiones en columnas y filas, respectivamente, de la matriz `pixMap`. Al parámetro `dataFormat` se le asigna una constante OpenGL que indica cómo se especifican los valores de la matriz. Por ejemplo, podríamos especificar un mismo color azul para todos los píxeles mediante la constante `GL_BLUE`, o podríamos especificar tres componentes de color en el orden azul, verde, rojo mediante la constante `GL_BGR`. Pueden utilizarse varios otros tipos de especificaciones de color, y en el siguiente capítulo examinaremos las selecciones de color con más detalle. Al parámetro `dataType` se le asigna una constante OpenGL tal como `GL_BYTE`, `GL_INT` o `GL_FLOAT`, para designar el tipo de datos de los valores de color almacenados en la matriz. La esquina inferior izquierda de esta matriz de colores se asigna a la posición actual de visualización, definida mediante `glRasterPos`. Como ejemplo, la siguiente instrucción muestra un mapa de píxeles definido mediante una matriz de 128 por 128 valores de color RGB:

```
glDrawPixels (128, 128, GL_RGB, GL_UNSIGNED_BYTE, colorShape);
```

Puesto que OpenGL proporciona diversos búferes, podemos almacenar una matriz de valores en un búfer concreto seleccionando dicho búfer como objetivo de la rutina `glDrawPixels`. Algunos píxeles almacenan valores de color, mientras que otros almacenan otros tipos de datos relativos a los píxeles. Por ejemplo, un *búfer de profundidad* se utiliza para almacenar las distancias de los objetos (profundidades) con respecto a la posición de visualización, mientras que un *búfer de patrones* se utiliza para almacenar los patrones de los contornos correspondientes a una escena. Podemos seleccionar uno de estos dos búferes asignando al parámetro `dataFormat` de la rutina `glDrawPixels` los valores `GL_DEPTH_COMPONENT` o `GL_STENCIL_INDEX`. Para estos búferes, tendríamos que almacenar en la matriz de píxeles valores de profundidad o información sobre el patrón del contorno. Examinaremos más en detalle estos búferes en los capítulos posteriores.

En OpenGL hay disponibles cuatro *búferes de color* que pueden utilizar para refrescar la pantalla. Dos de los búferes de color constituyen una pareja de escenas izquierda-derecha para mostrar imágenes estereoscópicas. Para cada uno de los búferes estereoscópicos, hay una pareja primer plano/segundo plano para visualización de animación con doble búfer. En cada implementación concreta de OpenGL puede que no esté soportada la visualización estereoscópica o la visualización con doble búfer. Si no se soportan ni los efectos estereoscópicos ni los mecanismos de doble búfer, entonces habrá un único búfer de refresco, que se denominará **búfer de color izquierdo de primer plano**. Este es el búfer predeterminado de refresco cuando no hay disponible mecanismo de doble búfer o cuando éste no está activado. Si está activado el doble búfer, se utilizarán como búferes predeterminados el búfer izquierdo de segundo plano y el búfer derecho de segundo plano o únicamente el búfer izquierdo de segundo plano, dependiendo del estado actual de la visualización estereoscópica. Asimismo, se permite al usuario definir búferes auxiliares de color que pueden utilizarse para propósitos distintos del refresco de pantalla, como por ejemplo para guardar una imagen que haya que copiar más tarde o un búfer de refresco para su visualización.

Podemos seleccionar un único color, o un búfer auxiliar, o una combinación de búferes de color para almacenar un mapa de píxeles mediante el siguiente comando:

```
glDrawBuffer (buffer);
```

Al parámetro búfer se le pueden asignar diversas constantes simbólicas OpenGL con el fin de especificar uno o más búferes de “dibujo”. Por ejemplo, podemos seleccionar un único búfer mediante `GL_FRONT_LEFT`, `GL_FRONT_RIGHT`, `GL_BACK_LEFT` o `GL_BACK_RIGHT`. Podemos seleccionar ambos búferes de primer plano mediante `GL_FRONT` o ambos búferes de segundo plano mediante `GL_BACK`, suponiendo que esté activada la visualización estereoscópica. Si la visualización estereoscópica no está asignada, estas dos constantes simbólicas especifican un único búfer. De forma similar podemos seleccionar las parejas de búferes izquierdos o derechos mediante `GL_LEFT` o `GL_RIGHT`, y podemos seleccionar todos los búferes de color disponibles mediante `GL_FRONT_AND_BACK`. Los búferes auxiliares se seleccionan mediante las constantes `GL_AUXk`, donde `k` es un valor entero que va de 0 a 3, aunque puede que algunas implementaciones de OpenGL permitan más de cuatro búferes auxiliares.

## Operaciones OpenGL de manipulación de búferes

Además de almacenar una matriz de valores de píxeles en un búfer, podemos extraer un bloque de valores de un búfer o copiar el bloque en otra área del búfer. También podemos realizar otros varios tipos de operaciones sobre una matriz de píxeles. En general, utilizamos el término **operación de manipulación de búfer** u **operación de rasterización**. Para describir cualquier función que procese de alguna manera una matriz de píxeles. Una operación de manipulación de búfer que desplace una matriz de valores de píxeles desde un lugar a otro se denomina también **transferencia en bloque** de los valores de los píxeles. En un sistema monocromo, estas operaciones se denominan **transferencias bitblt** o **transferencias de bloques de bits**, particularmente cuando dichas funciones se implementan en hardware. En sistemas con más de dos niveles de color, se utiliza también el término **pixblt** para designar las transferencias de bloques.

Podemos utilizar la siguiente función para seleccionar un bloque rectangular de valores de píxeles en un conjunto especificado de búferes:

```
glReadPixels (xmin, ymin, width, height,
              dataFormat, dataType, array};
```

La esquina inferior izquierda del bloque rectangular que hay que extraer se encuentra en la posición de coordenadas de la pantalla (`xmin`, `ymin`). Los parámetros `width`, `height`, `dataFormat` y `dataType` son iguales que para la rutina `glDrawPixels`. El tipo de los datos que hay que guardar en el parámetro `array` dependen del búfer seleccionado. Podemos elegir el búfer de profundidad o el búfer de patrones asignando el valor `GL_DEPTH_COMPONENT` o `GL_STENCIL_INDEX` al parámetro `dataFormat`.

Para seleccionar una combinación concreta de búferes de color o un búfer auxiliar para la rutina `glReadPixels` se utiliza la función:

```
glReadBuffer (buffer);
```

Las constantes simbólicas para especificar uno o más búferes son iguales que para la rutina `glDrawBuffer`, salvo porque no podemos seleccionar los cuatro búferes de color a la vez. La selección pre-determinada de búfer es la pareja de búferes izquierdo-derecho de primer plano o simplemente el búfer izquierdo de primer plano, dependiendo del estado de la visualización estereoscópica.

También podemos copiar un bloque de datos de píxeles de una ubicación a otra dentro del conjunto de búferes OpenGL, para lo cual se utiliza la siguiente rutina:

```
glCopyPixels (xmin, ymin, width, height, pixelValues};
```

La esquina inferior izquierda del bloque se encontrará en las coordenadas de pantalla (`xmin`, `ymin`), y los parámetros `width` y `height` deberán tener valores enteros positivos para especificar el número de columnas y de filas, respectivamente, que hay que copiar. El parámetro `pixelValues` puede tener como valores las constantes `GL_COLOR`, `GL_DEPTH` o `GL_STENCIL` para indicar el tipo de datos que queremos copiar: valores de color, valores de profundidad o valores de patrón de contorno. Y el bloque de valores de píxeles será copiado desde un *búfer de origen* y el bloque de valores de píxel se copian desde un *búfer de origen* hasta un *búfer de destino*, asignándose la esquina inferior izquierda a la posición de visualización actual. El búfer de origen se selecciona mediante el comando `glReadBuffer`, mientras que para el búfer de destino se emplea el comando `glDrawBuffer`. Tanto la región que hay que copiar como el área de destino deben caer completamente dentro de los límites de las coordenadas de pantalla.

Para conseguir diferentes efectos a medida que se almacena un bloque de valores de píxel en un búfer mediante `glDrawPixels` o `glCopyPixels`, podemos combinar de diversas formas los valores entrantes con los anteriores valores del búfer. Como ejemplo, podemos aplicar operaciones lógicas, tales como *and*, *or* y *exclusive or*. En OpenGL, puede seleccionarse una operación lógica aplicable bit a bit para combinar los valores de color de los píxeles entrantes y de destino mediante las funciones:

```
 glEnable (GL_COLOR_LOGIC_OP);
 glLogicOp (logicOp);
```

Pueden asignarse diversas constantes simbólicas al parámetro `logicOp`, incluyendo `GL_AND`, `GL_OR` y `GL_XOR`. Además, tanto los valores de bit entrantes como los valores de bit de destino pueden invertirse (intercambiando los valores 0 y 1). Para invertir los valores entrantes de los bits de color se utiliza la constante `GL_COPY_INVERTED`, con la que se sustituyen los valores de destino por los valores entrantes invertidos. Y también podemos simplemente invertir los valores de bit de destino sin reemplazarlos por los valores entrantes, utilizando `GL_INVERT`. Las diversas operaciones de inversión pueden también combinarse mediante las operaciones lógicas *and*, *or* y *exclusive or*. Otras opciones incluyen borrar todos los bits de destino, asignándoles el valor 0 (`GL_CLEAR`), o asignar a todos los bits de destino el valor 1 (`GL_SET`). El valor predeterminado para la rutina `glLogicOp` es `GL_COPY`, que simplemente reemplaza los valores de destino por los valores entrantes.

Hay disponibles otras rutinas OpenGL adicionales para manipular matrices de píxeles procesadas por las funciones `glDrawPixels`, `glReadPixels` y `glCopyPixels`. Por ejemplo, las rutinas `glPixelTransfer`

y `glPixelMap` pueden utilizarse para efectuar un desplazamiento lógico o un ajuste de los valores de color, de los valores de profundidad o de los patrones de contorno. Volveremos a tratar de las operaciones de píxeles en capítulos posteriores, cuando pasemos a analizar otras facetas de los paquetes de gráficos por computadora.

## 3.20 PRIMITIVAS DE CARACTERES

---

Las imágenes gráficas incluyen a menudo información textual, como por ejemplo etiquetas, gráficas, carteles en edificios o vehículos e información general de identificación para aplicaciones de simulación y visualización. La mayoría de los paquetes gráficos incluyen rutinas para generar primitivas de caracteres gráficos. Algunos sistemas proporcionan un amplio conjunto de funciones de caracteres, mientras que otros sólo ofrecen un soporte de generación de caracteres mínimo.

Las letras, los números y otros caracteres pueden mostrarse con diversos tamaños y estilos. El estilo global de diseño para un conjunto (o familia) de caracteres se denomina **tipo de letra**. Hoy en día, hay disponibles miles de tipos de letra distintos para aplicaciones informáticas, como por ejemplo Courier, Helvetica, New York, Palatino o Zapf Chancery. Originalmente, el término **fuente** hacía referencia a un conjunto de caracteres extruídos en metal con un tamaño y formato concretos, como por ejemplo Courier Itálica de 10 puntos o Palatino Negrita de 12 puntos. Una fuente de 14 puntos tiene una altura total de los caracteres de unos 0,5 centímetros; en otras palabras 72 puntos es aproximadamente equivalente a 2,54 centímetros (una pulgada). Hoy en día, se usan los términos fuente y tipo de letra de manera intercambiable, ya que las tareas modernas de impresión raramente se realizan con caracteres extruídos en metal.

Los tipos de letra (o fuentes) pueden dividirse en dos amplios grupos: *serif* y *sans serif*. Un tipo serif tiene pequeñas líneas o acentos en los extremos de los principales trazos de los caracteres, mientras que los tipos sans-serif no tiene este tipo de acentos. Por ejemplo, el texto de este libro está realizado con una fuente serif. Pero esta frase está impresa en una fuente sans-serif (*Univers*). Los tipos serif son generalmente más *legibles*; es decir, es más fácil leer grandes bloques de texto. Por otra parte, los caracteres individuales de los tipos sans-serif son más fáciles de reconocer, por lo que este tipo de letra resulta adecuada para etiquetas y cortos encabezados.

Las fuentes también se clasifican dependiendo de si son *monoespaciadas* o *proporcionales*. Todos los caracteres de una fuente monoespaciada tienen la misma anchura, mientras que en una fuente proporcional la anchura de los caracteres varía.

Se utilizan dos representaciones diferentes para almacenar fuentes en una computadora. Un método simple para representar las formas de los caracteres en un tipo de letra concreto consiste en definir un patrón de valores binarios sobre una cuadrícula rectangular. Entonces, ese conjunto de caracteres se denomina **fuente de mapa de bits**. Un conjunto de caracteres de mapa de bits también se denomina en ocasiones **fuente raster** o **fuente digitalizada**.

Otro esquema más flexible consiste en describir las formas de los caracteres utilizando secciones de línea rectas y de curvas, como por ejemplo en PostScript. En este caso, el conjunto de caracteres se denomina **fuente de contorno** o **fuente de trazos**. La Figura 3.62 ilustra los dos métodos para la representación de caracteres. Cuando se aplica el patrón de la Figura 3.62(a) a un área del búfer de imagen, los bits con valor 1 designan qué posiciones de píxel hay que mostrar en un color especificado. Para visualizar la forma del carácter de la Figura 3.62(b), el interior del contorno de carácter se trata como un área de relleno.

Las fuentes de mapa de bits son las más simples de definir y de visualizar: basta con asignar las cuadrículas de caracteres a una posición dentro del búfer de imagen. Sin embargo, en general, las fuentes de mapa de bits requieren más espacio de almacenamiento, ya que es necesario guardar cada variante (tamaño y formato) dentro de una *caché de fuentes*. Se pueden generar diferentes tamaños y otras variaciones, como por ejemplo negrita e itálica a partir de un conjunto de fuentes en mapa de bits, pero los resultados que se obtienen no suelen ser buenos. Podemos incrementar o decrementar el tamaño del mapa de bits de un carácter solamente en

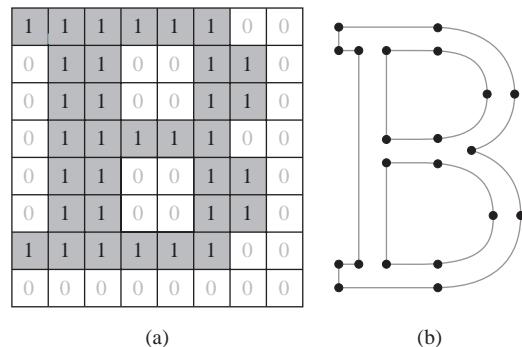


FIGURA 3.62. La letra «B» representada con un patrón de mapa de bits 8 por 8 (a) y con una forma de contorno definida mediante segmentos de líneas rectas y de curvas (b).

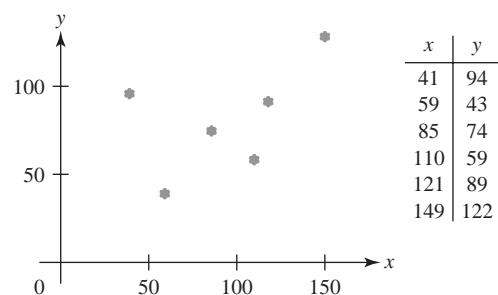


FIGURA 3.63. Una gráfica de tipo polimarcador para un conjunto de valores de datos.

múltiplos enteros del tamaño del píxel. Si queremos doblar el tamaño de un carácter, tenemos que doblar el número de píxeles del mapa de bits, con lo que los bordes del carácter tendrán una apariencia escalonada.

Por contraste con las fuentes de mapa de bits, las fuentes de contorno pueden incrementarse de tamaño sin distorsionar la forma de los caracteres. Además, estas fuentes requieren menos espacio de almacenamiento, porque no hace falta una caché de fuente separada para cada una de las variantes del tipo de caracteres. Podemos generar caracteres en negrita, en itálica o de diferentes tamaños manipulando las definiciones de las curvas que especifican el contorno de los caracteres. Pero obviamente hace falta más tiempo para procesar las fuentes de contorno, ya que es necesario digitalizar los caracteres para almacenarlos en el búfer de imagen.

Hay una diversidad de posibles funciones para visualizar caracteres. Algunos paquetes gráficos proporcionan una función que acepta cualquier cadena de caracteres y una indicación de cuál debe ser la posición de inicio de la cadena dentro del búfer de imagen. Otro tipo de funciones son aquellas que muestran un único carácter en una o más posiciones seleccionadas. Puesto que esta rutina de caracteres resulta muy útil para mostrar caracteres en una imagen, por ejemplo, de una red de computadoras o a la hora de mostrar una gráfica con los puntos correspondientes a un conjunto de datos discretos, el carácter mostrado por esta rutina se denomina en ocasiones **símbolo marcador** o **polimarcador**, por analogía con la primitiva de polilínea. Además de los caracteres estándar, otras formas geométricas especiales como los puntos, los círculos y las cruces suelen estar disponibles como símbolos marcadores. La Figura 3.63 muestra una gráfica de un conjunto de datos discreto donde se utiliza un asterisco como símbolo marcador.

Las descripciones geométricas de los caracteres se proporcionan en coordenadas universales, al igual que con las otras primitivas, y esta información se hace corresponder con las coordenadas de pantalla mediante las transformaciones de visualización. Un carácter de mapa de bits se describe mediante una cuadrícula rectangular de valores binarios y una posición de referencia para la cuadrícula. Esta posición de referencia se asigna entonces a una ubicación específica dentro del búfer de imagen. Un carácter de contorno está definido por un conjunto de puntos que hay que conectar mediante una serie de segmentos curvos y rectos, junto con una posición de referencia que hay que hacer corresponder con una ubicación determinada del búfer de imagen. La posición de referencia puede especificarse para un único carácter de contorno o para una cadena de caracteres. En general, las rutinas de caracteres permiten generar imágenes con caracteres tanto bidimensionales como tridimensionales.

## 3.21 FUNCIONES OpenGL DE CARACTERES

---

La biblioteca OpenGL básica sólo proporciona soporte de bajo nivel para la visualización de caracteres individuales o cadenas de texto. Podemos definir explícitamente cualquier carácter como un mapa de bits, como en la forma de ejemplo mostrada en la Figura 3.61, y podemos almacenar un conjunto de mapas de bits caracteres como listado de fuentes. Entonces, se podrá visualizar una cadena de texto asignando una secuencia seleccionada de mapa de bits de la lista de fuentes a un conjunto de posiciones adyacentes dentro del búfer de imagen.

Sin embargo, GLUT (OpenGL Utility Toolkit) proporciona algunos conjuntos de caracteres predefinidos, por lo que no es necesario que creamos nuestras propias fuentes como mapas de bits, a menos que queramos mostrar una fuente que no esté disponible en GLUT. La biblioteca de GLUT contiene rutinas para visualizar tanto fuentes de mapa de bits como fuentes de contorno. Las fuentes de mapa de bits de GLUT se visualizan utilizando la función `glBitmap` de OpenGL, mientras que las fuentes de contorno se generan mediante contornos de tipo polilínea (`GL_LINE_STRIP`).

Podemos visualizar un carácter GLUT de mapa de bits mediante:

```
glutBitmapCharacter (font, character);
```

donde al parámetro `font` se le asigna una constante simbólica de GLUT que identifica un conjunto concreto de tipos de letra, mientras que al parámetro `character` se le asigna el código ASCII o el carácter específico que queramos visualizar. Así, para mostrar la letra mayúscula “A”, podemos utilizar el valor ASCII 65 o introducir directamente el carácter ‘A’. De forma similar, un valor de código de 66 será el equivalente a ‘B’, el código 97 se corresponderá con la letra minúscula ‘a’, el código 98 se corresponderá con la ‘b’, etc. Hay disponibles tanto fuentes de anchura fija como de espaciado proporcional. Se puede seleccionar una fuente de anchura fija asignando las constantes `GLUT_BITMAP_8_BY_13` o `GLUT_BITMAP_9_BY_15` al parámetro `font`. Y podemos seleccionar una fuente de 10 puntos con espaciado proporcional utilizando `GLUT_BITMAP_TIMES_ROMAN_10` o `GLUT_BITMAP_HELVETICA_10`. También hay disponible una fuente Times-Roman de 12 puntos, así como fuentes de tipo Helvetica de 12 puntos y 18 puntos.

Cada carácter generado por `glutBitmapCharacter` se visualiza de modo que el origen (esquina inferior izquierda) del mapa de bits se encuentre en la posición de visualización actual. Después de cargar el mapa de bits del carácter en el búfer de refresco, se añade a la coordenada `x` de la posición actual de visualización un desplazamiento igual a la anchura del carácter. Como ejemplo, podríamos visualizar una cadena de texto que contuviera 36 caracteres de mapa de bits mediante el código siguiente:

```
glRasterPosition2i (x, y);
for (k = 0; k < 36; k++)
    glutBitmapCharacter (GLUT_BITMAP_9_BY_15, text [k]);
```

Los caracteres se muestran en el color que haya sido especificado antes de ejecutar la función `glutBitmapCharacter`.

Un carácter de contorno se muestra mediante la siguiente llamada a función:

```
glutStrokeCharacter (font, character);
```

Para esta función, podemos asignar al parámetro `font` el valor `GLUT_STROKE_ROMAN`, que muestra una fuente con espaciado proporcional, o el valor `GLUT_STROKE_MONO_ROMAN`, que muestra una fuente con espaciado constante. Podemos controlar el tamaño y la posición de estos caracteres especificando una serie de operaciones de transformación (Capítulo 5) antes de ejecutar la rutina `glutStrokeCharacter`. Después de visualizar cada carácter, se aplica un desplazamiento automático de coordenadas para que la posición de visualización del siguiente carácter se encuentre a la derecha del carácter actual. Las cadenas de texto generadas mediante fuentes de contorno son parte de la descripción geométrica de una escena bidimensional o tridimensional, porque se construyen mediante segmentos de líneas. Así, pueden visualizarse desde diversas direcciones y podemos reducirlas o ampliarlas sin que se distorsionen, o incluso transformarlas de otras maneras. Sin embargo, son más lentas de visualizar, comparadas con las fuentes de mapa de bits.

## 3.22 PARTICIONAMIENTO DE IMÁGENES

---

Algunas bibliotecas gráficas incluyen rutinas para describir una imagen como una colección de secciones nominadas y para manipular las secciones individuales de una imagen. Utilizando estas funciones, podemos crear, editar, borrar o mover una parte de una imagen independientemente de los demás componentes de la misma. Y también podemos utilizar esta característica de los paquetes gráficos para el modelado jerárquico (Capítulo 14), con el cual la descripción de un objeto se proporciona en forma de una estructura de árbol compuesta por una serie de niveles que especifican las subpartes del objeto.

Se utilizan diversos nombres para las subsecciones de una imagen. Algunos paquetes gráficos se refieren a ellos denominándolos estructuras (*structures*), mientras que otros paquetes los denominan segmentos (*segments*) u objetos (*objects*). Asimismo, las operaciones disponibles para cada subsección varían enormemente de un paquete a otro. Los paquetes de modelado, por ejemplo, proporcionan un amplio rango de operaciones que pueden usarse para describir y manipular los elementos de una imagen. Por el contrario, en las bibliotecas gráficas siempre se pueden estructurar y gestionar los componentes de una imagen utilizando elementos procedimentales disponibles en algún lenguaje de alto nivel, como por ejemplo C++.

## 3.23 LISTAS DE VISUALIZACIÓN DE OpenGL

---

A menudo, puede resultar conveniente o más eficiente almacenar la descripción de un objeto (o cualquier otro conjunto de comandos OpenGL) como una secuencia nominada de instrucciones. En OpenGL, podemos hacer esto utilizando una estructura denominada **lista de visualización**. Una vez creada una lista de visualización, podemos hacer referencia a la lista múltiples veces, utilizando operaciones de visualización diferentes. En una red, una lista de visualización que describa una escena puede almacenarse en una máquina servidora, lo que elimina la necesidad de transmitir los comandos de la lista cada vez que haya que visualizar la escena. También podemos configurar una lista de visualización con el fin de guardarla para su ejecución posterior, o podemos especificar que los comandos de la lista se ejecuten inmediatamente. Además, las listas de visualización son particularmente útiles para el modelador jerárquico, en el que los objetos complejos se describen mediante un conjunto de subpartes más simples.

### Creación y denominación de una lista de visualización OpenGL

Podemos transformar un conjunto de comandos OpenGL en una lista de visualización encerrando los comandos dentro de la pareja de funciones `glNewList/glEndList`. Por ejemplo,

```
glNewList (listID, listMode);
.
.
.
glEndList ( );
```

Esta estructura forma una lista de visualización asignando un valor entero positivo al parámetro `listID` como nombre de la lista. Al parámetro `listMode` se le asigna una constante simbólica OpenGL que puede ser `GL_COMPILE` o `GL_COMPILE_AND_EXECUTE`. Si queremos guardar la lista para su ejecución posterior, utilizaremos `GL_COMPILE`. En caso contrario, los comandos se ejecutan a medida que se los introduce en la lista, además de poder ejecutar la lista de nuevo en un instante posterior.

A medida que se crea una lista de visualización, se evalúan las expresiones donde intervienen parámetros tales como las posiciones de coordenadas y las componentes de color, de modo que en la lista de visualización sólo se almacenan los valores de los parámetros. Cualesquiera cambios posteriores que se realicen a estos parámetros no tendrán ningún efecto sobre la lista. Puesto que los valores de las listas de visualización no pue-

den modificarse, no podemos incluir en una lista de visualización determinados comandos OpenGL, como los punteros a listas de vértices.

Podemos crear todas las listas de visualización que queramos, y ejecutar una lista concreta de comandos sin más que efectuar una llamada a su identificador. Además, puede incrustarse una lista de visualización dentro de otra lista. Pero si se asigna a una lista un identificador que ya haya sido utilizado, la nueva lista sustituirá a la lista anterior a la que se la hubiera asignado dicho identificador. Por tanto, para evitar perder una lista por la reutilización accidental de su identificador, lo mejor es dejar que OpenGL genere un identificador por nosotros:

```
listID = glGenLists (1);
```

Esta instrucción devuelve un (1) identificador entero positivo no utilizado, asignándolo a la variable `listID`. Podemos obtener un rango de identificadores de lista no utilizados sin más que utilizar el argumento de `glGenLists`, sustituyendo el valor 1 por algún otro entero positivo. Por ejemplo, si ejecutamos el comando `glGenLists (6)`, se reservará una secuencia de seis valores enteros positivos contiguos y el primer valor de esta lista de identificadores se devolverá a la variable `listID`. La función `glGenLists` devuelve un valor 0 si se produce un error o si el sistema no puede generar el rango solicitado de enteros contiguos. Por tanto, antes de utilizar un identificador obtenido mediante la rutina `glGenLists`, conviene comprobar que éste es distinto de 0.

Aunque pueden generarse identificadores de lista no utilizados mediante la función `glGenLists`, también podemos consultar independientemente al sistema para determinar si un cierto valor entero específico ha sido ya utilizado como nombre de una lista. La función para efectuar esta consulta es:

```
glIsList (listID);
```

Esta función devolverá un valor `GL_TRUE` si el valor de `listID` es un entero que ya haya sido utilizado como nombre de una lista de visualización. Si el valor entero no ha sido utilizado como nombre de una lista, la función `glIsList` devolverá el valor `GL_FALSE`.

## Ejecución de listas de visualización OpenGL

Podemos ejecutar una lista de visualización mediante la instrucción:

```
glCallList (listID);
```

El siguiente segmento de código ilustra la creación y ejecución de una lista de visualización. Primero definimos una lista de visualización que contiene la descripción de un hexágono regular, definido en el plano *xy* utilizando un conjunto de seis vértices equiespaciados dispuestos a lo largo de una circunferencia y que tienen su centro en las coordenadas (200, 200), siendo el radio igual a 150. Después, realizamos una llamada a la función `glCallList`, que muestra el hexágono.

```
const double TWO_PI = 6.2831853;
GLuint regHex;

GLdouble theta;
GLint x, y, k;

/* Especificar una lista de visualización para un hexágono regular.
 * Los vértices del hexágono son seis puntos
 * equiespaciados dispuestos sobre una circunferencia.*/
regHex = glGenLists (1);    // Obtener un identificador para la
                           // lista de visualización.
```

```

glNewList (regHex, GL_COMPILE);
glBegin (GL_POLYGON);
for (k = 0; k < 6; k++) {
    theta = TWO_PI * k / 6.0;
    x = 200 + 150 * cos (theta);
    y = 200 + 150 * sin (theta);
    glVertex2i (x, y);
}
glEnd ();
glEndList ();
glCallList (regHex);

```

Podemos ejecutar varias listas de visualización de una vez utilizando las siguientes dos instrucciones:

```

glListBase (offsetValue);
glCallLists (nLists, arrayDataType, listIDArray);

```

El número entero de listas que queremos ejecutar se asigna al parámetro `nLists`, mientras que el parámetro `listIDArray` es una matriz de identificadores de listas de visualización. En general, `listIDArray` puede contener cualquier número de elementos, ignorándose los identificadores de listas de visualización que no sean válidos. Asimismo, los elementos de `listIDArray` pueden especificarse en diversos formatos de datos y el parámetro `arrayDataType` se utiliza para indicar un tipo de datos, como por ejemplo `GL_BYTE`, `GL_INT`, `GL_FLOAT`, `GL_3_BYTE` o `GL_4_BYTES`. Los identificadores de las listas de visualización se calculan sumando el valor de un elemento de `listIDArray` al valor entero de `offsetValue` que se proporciona en la función `glListBase`. El valor predeterminado de `offsetValue` es 0.

Este mecanismo para especificar una secuencia de listas de visualización que haya que ejecutar nos permite definir grupos de listas de visualización relacionadas, cuyos identificadores se formarán a partir de códigos o nombres simbólicos. Un ejemplo típico sería un conjunto de caracteres, en el que el identificador de cada lista de visualización sería el valor ASCII de un carácter. Cuando haya definidos varios conjuntos de fuentes, puede utilizarse el parámetro `offsetValue` de la función `glListBase` para obtener una fuente concreta que esté descrita dentro de la matriz `listIDArray`.

## Borrado de listas de visualización OpenGL

Para eliminar un conjunto contiguo de listas de visualización, podemos utilizar la llamada a función:

```
glDeleteLists (startID, nLists);
```

El parámetro `startID` indica el identificador de lista de visualización inicial, mientras que el parámetro `nLists` especifica el número de listas que hay que borrar. Por ejemplo, la instrucción:

```
glDeleteLists (5, 4);
```

elimina las cuatro listas de visualización con los identificadores 5, 6, 7 y 8. Los valores de identificadores que hagan referencia a listas de visualización no existentes se ignorarán.

## 3.24 FUNCIÓN OpenGL DE REDIMENSIONAMIENTO DE LA VENTANA DE VISUALIZACIÓN

En nuestro programa OpenGL introductorio (Sección 2.9), hemos explicado las funciones para establecer una ventana de visualización inicial. Pero después de la generación de una imagen, a menudo nos surge la nece-

sidad de utilizar el puntero del ratón para arrastrar la ventana de visualización a otra posición de la pantalla o para cambiar su tamaño. Cambiar el tamaño de una ventana de visualización puede hacer que se modifique su relación de aspecto y que, por tanto, los objetos se distorsionen con respecto a su forma original.

Para poder compensar los cambios realizados en las dimensiones de la ventana de visualización, la biblioteca GLUT proporciona la siguiente rutina:

```
glutReshapeFunc (winReshapeFcn);
```

Podemos incluir esta función en el procedimiento `main` de nuestro programa, junto con las otras rutinas GLUT, y dicha función se activará cada vez que se altere el tamaño de la ventana de visualización. El argumento de esta función GLUT es el nombre de un procedimiento que será el que reciba como parámetros la nueva anchura y la nueva altura de la ventana de visualización. Entonces, podemos utilizar las nuevas dimensiones para reinicializar los parámetros de proyección y realizar cualesquiera otras operaciones, como por ejemplo modificar el color de la ventana de visualización. Además, podemos guardar los nuevos valores de anchura y altura para poderlos utilizar en otros procedimientos de nuestro programa.

Como ejemplo, el siguiente programa ilustra cómo podríamos estructurar el procedimiento `winReshapeFcn`. Incluimos el comando `glLoadIdentity` en la función de redimensionamiento para que los valores previos de los parámetros de proyección no afecten a la nueva configuración de proyección. Este programa muestra el hexágono regular del que hemos hablado en la Sección 3.23. Aunque el centro del hexágono (en la posición del centro del círculo) de este ejemplo se especifica en términos de los parámetros de la ventana de visualización, la posición del hexágono no se verá afectada por los cambios que se realicen en el tamaño de la ventana de visualización. Esto se debe a que el hexágono está definido con una lista de visualización, y en dicha lista sólo se almacenan las coordenadas originales del centro. Si queremos que la posición del hexágono cambie cuando se modifique el tamaño de la ventana de visualización, tendremos que definir el hexágono de otra manera o modificar la referencia de coordenadas de la ventana de visualización. La salida de este programa se muestra en la Figura 3.64.

```
#include <GL/glut.h>
#include <math.h>
#include <stdlib.h>

const double TWO_PI = 6.2831853;

/* Tamaño inicial de la ventana de visualización. */
GLsizei winWidth = 400, winHeight = 400;
GLuint regHex;

class screenPt
{
private:
    GLint x, y;

public:
    /* Constructor predeterminado: inicialización la
     * posición de coordenadas a (0, 0). */
    screenPt () {
        x = y = 0;
    }

    void setCoords (GLint xCoord, GLint yCoord) {
        x = xCoord;
        y = yCoord;
    }
}
```

```

        GLint getx ( ) const {
            return x;
        }

        GLint gety ( ) const {
            return y;
        }
    };

    static void init (void)
    {
        screenPt hexVertex, circCtr;
        GLdouble theta;
        GLint k;

        /* Establecer coordenadas del centro del círculo. */
        circCtr.setCoords (winWidth / 2, winHeight / 2);
        glClearColor (1.0, 1.0, 1.0, 0.0); // Color ventana visualización = blanco.

        /* Definir una lista de visualización para un hexágono regular rojo.
         * Los vértices del hexágono son seis puntos
         * equiespaciados situados sobre una circunferencia.
         */
        regHex = glGenLists (1); // Obtener un identificador para la vista
                                // de visualización.

        glNewList (regHex, GL_COMPILE);
        glColor3f (1.0, 0.0, 0.0); // Establecer rojo como color de relleno
                                // para el hexágono.

        glBegin (GL_POLYGON);
        for (k = 0; k < 6; k++) {
            theta = TWO_PI * k / 6.0;
            hexVertex.setCoords (circCtr.getx ( ) + 150 * cos (theta),
                                circCtr.gety ( ) + 150 * sin (theta));
            glVertex2i (hexVertex.getx ( ), hexVertex.gety ( ));
        }
        glEnd ( );
        glEndList ( );
    }

    void regHexagon (void)
    {
        glClear (GL_COLOR_BUFFER_BIT);
        glCallList (regHex);
        glFlush ( );
    }

    void winReshapeFcn (int newWidth, int newHeight)
    {
        glMatrixMode (GL_PROJECTION);
        glLoadIdentity ( );
        gluOrtho2D (0.0, (GLdouble) newWidth, 0.0, (GLdouble) newHeight);
    }
}

```

```

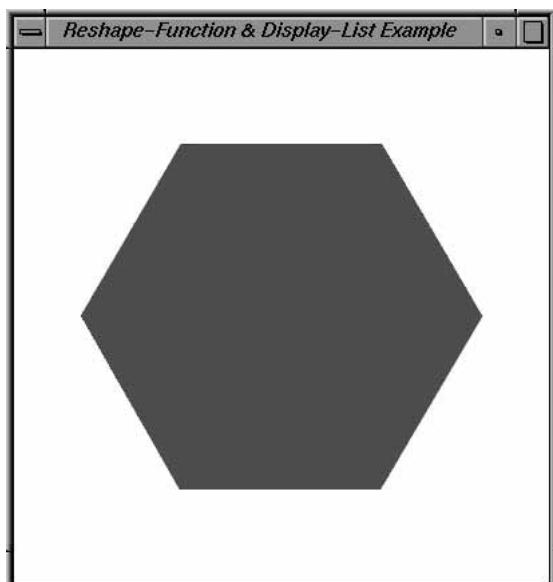
    glClear (GL_COLOR_BUFFER_BIT);
}

void main (int argc, char** argv)
{
    glutInit (&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    glutInitWindowPosition (100, 100);
    glutInitWindowSize (winWidth, winHeight);
    glutCreateWindow («Reshape-Function & Display-List Example»);

    init ( );
    glutDisplayFunc (regHexagon);
    glutReshapeFunc (winReshapeFcn);

    glutMainLoop ( );
}

```

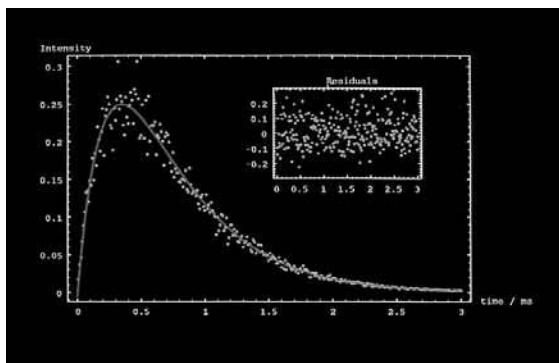


**FIGURA 3.64.** Ventana de visualización generada por el programa de ejemplo que ilustra el uso de la función de redimensionamiento.

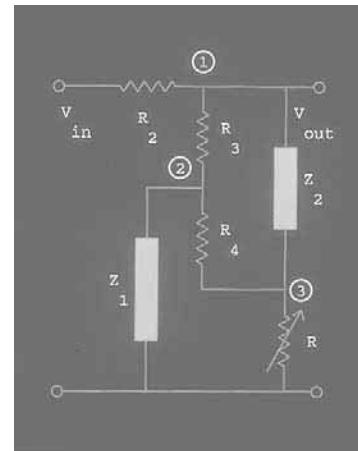
## 3.25 RESUMEN

Las primitivas de salida analizadas en este capítulo proporcionan las herramientas básicas para construir imágenes con puntos individuales, líneas rectas, curvas, áreas de color llenas, patrones matriciales y textos. Las primitivas se especifican proporcionando su descripción geométrica en un sistema de referencia cartesiano en coordenadas universales. Las Figuras 3.65 y 3.66 proporcionan ejemplos de imágenes generadas mediante primitivas de salida.

Tres métodos que pueden utilizarse para situar las posiciones de los píxeles a lo largo de una trayectoria recta son el algoritmo DDA, el algoritmo de Bresenham y el método del punto medio. El algoritmo de Bresenham y el método del punto medio son equivalentes en el caso de las líneas y son también los más



**FIGURA 3.65.** Una gráfica de datos generada mediante segmentos lineales, curvas, símbolos de caracteres marcadores y texto (Cortesía de Wolfram Research, Inc., fabricante del paquete software Mathematica.)



**FIGURA 3.66.** Un diagrama eléctrico dibujado mediante secciones rectas, círculos, rectángulos y texto. (Cortesía de Wolfram Research, Inc., fabricante del paquete software Mathematica.)

eficientes. Los valores de color correspondientes a las posiciones de los píxeles a lo largo del trayecto lineal se pueden almacenar de manera eficiente en el búfer de imagen calculando incrementalmente las direcciones de memoria. Cualquiera de los algoritmos de generación de líneas puede adaptarse para su implementación paralela, particionando los segmentos lineales y distribuyendo las particiones entre los procesadores disponibles.

Los círculos y elipses pueden digitalizarse de manera eficiente y precisa utilizando el método del punto medio y teniendo en cuenta la simetría de las curvas. Otras secciones cónicas (paráolas e hipérbolas) pueden dibujarse utilizando métodos similares. Las curvas de tipo *spline*, que son polinomios continuos por partes, se utilizan ampliamente en las aplicaciones de animación y de diseño asistido por computadora. Pueden conseguirse implementaciones paralelas para la generación de imágenes de curvas utilizando métodos similares a los que se emplean para el procesamiento paralelo de líneas.

Para tener en cuenta el hecho de que las líneas y curvas visualizadas piden anchuras finitas, podemos ajustar las dimensiones de los objetos en píxeles con el fin de que coincidan con las dimensiones geométricas especificadas. Esto puede llevarse a cabo empleando un esquema de direccionamiento que haga referencia a las posiciones de los píxeles utilizando la esquina inferior izquierda, o bien ajustando las longitudes de las líneas.

Un área de relleno es una región plana que hay que dibujar utilizando un color homogéneo o un patrón de colores. Las primitivas de áreas de relleno en la mayoría de los paquetes gráficos son polígonos, pero en general podríamos especificar una región de relleno con cualquier contorno arbitrario. A menudo, los sistemas gráficos sólo permiten áreas de relleno poligonales convexas. En ese caso, puede visualizarse un área de relleno poligonal cóncava dividiéndola en una serie de polígonos convexos. Los triángulos son los polígonos más fáciles de llenar ya que cada línea de exploración que cruza un triángulo intersecta exactamente dos aristas del polígono (suponiendo que la línea de exploración no pase por ninguno de los vértices).

Puede emplearse la regla par-impar para determinar los puntos interiores de una región plana. También son útiles otros métodos para definir el interior de los objetos, particularmente en el caso de objetos irregulares que se auto-intersecan. Un ejemplo común es la regla del número de vueltas distinto de cero. Esta regla es más flexible que la regla par-impar a la hora de procesar objetos definidos mediante múltiples contornos. También podemos utilizar variantes de la regla del número de vueltas distinto de cero para combinar áreas planas utilizando operaciones booleanas.

**TABLA 3.1.** RESUMEN DE FUNCIONES PRIMITIVAS OpenGL DE SALIDA Y RUTINAS RELACIONADAS.

<i>Función</i>	<i>Descripción</i>
<code>gluOrtho2D</code>	Especifica una referencia 2D de coordenadas universales.
<code>glVertex*</code>	Selecciona unas determinadas coordenadas. Esta función debe incluirse dentro de una pareja <code>glBegin/glEnd</code> .
<code>glBegin (GL_POINTS);</code>	Dibuja uno o más puntos, cada uno de ellos especificado mediante una función <code>glVertex</code> . La lista de puntos se cierra mediante una instrucción <code>glEnd</code> .
<code>glBegin (GL_LINES);</code>	Muestra un conjunto de segmentos lineales rectos, cuyos extremos se especifican mediante funciones <code>glVertex</code> . La lista de puntos se termina mediante una instrucción <code>glEnd</code> .
<code>glBegin (GL_LINE_STRIP);</code>	Muestra una polilínea, especificada utilizando la misma estructura que <code>GL_LINES</code> .
<code>glBegin (GL_LINE_LOOP);</code>	Muestra una polilínea cerrada, especificada utilizando la misma estructura <code>GL_LINES</code> .
<code>glRect*</code>	Muestra un rectángulo lleno en el plano <i>xy</i> .
<code>glBegin (GL_POLYGON);</code>	Muestra un polígono lleno, cuyos vértices se proporcionan mediante funciones <code>glVertex</code> y se terminan mediante una instrucción <code>glEnd</code> .
<code>glBegin (GL_TRIANGLES);</code>	Muestra un conjunto de triángulos llenos utilizando la misma estructura que <code>GL_POLYGON</code> .
<code>glBegin (GL_TRIANGLE_STRIP);</code>	Muestra una malla de triángulos llenos, especificada mediante la misma estructura que <code>GL_POLYGON</code> .
<code>glBegin (GL_TRIANGLE_FAN);</code>	Muestra una malla de triángulos llenos con forma de ventilador, en la que todos los triángulos se conectan al primer vértice, que se especifica con la misma estructura que <code>GL_POLYGON</code> .
<code>glBegin (GL_QUADS);</code>	Muestra un conjunto de cuadriláteros llenos, que se especifican mediante la misma estructura que <code>GL_POLYGON</code> .
<code>glBegin (GL_QUAD_STRIP);</code>	Muestra una malla de cuadriláteros llenos, que se especifica con la misma estructura que <code>GL_POLYGON</code> .
<code>glEnableClientState (GL_VERTEX_ARRAY);</code>	Activa las características de matrices de vértices de OpenGL.
<code>glVertexPointer (size, type, stride, array);</code>	Especifica una matriz de valores de coordenadas.
<code>glDrawElements (prim, num, type, array);</code>	Muestra un tipo de primitiva especificado a partir de los datos de una matriz.
<code>glNewList (listID, listMode)</code>	Define un conjunto de comandos como una lista de visualización, que se termina mediante una instrucción <code>glEndList</code> .

(Continúa)

**TABLA 3.1.** RESUMEN DE FUNCIONES PRIMITIVAS OpenGL DE SALIDA Y RUTINAS RELACIONADAS. (*Cont.*)

<i>Función</i>	<i>Descripción</i>
<code>glGenLists</code>	Genera uno o más identificadores para listas de visualización.
<code>glIsList</code>	Función de consulta para determinar si un cierto identificador de lista de visualización está siendo utilizado.
<code>glCallList</code>	Ejecuta una única lista de visualización.
<code>glListBase</code>	Especifica un valor de desplazamiento para una matriz de identificadores de listas de visualización.
<code>glCallLists</code>	Ejecuta múltiples listas de visualización.
<code>glDeleteLists</code>	Elimina una secuencia especificada de listas de visualización.
<code>glRasterPos*</code>	Especifica una posición actual bidimensional o tridimensional para el búfer de imagen. Esta posición se utiliza como referencia para los patrones de mapa de bits y de mapa de píxeles.
<code>glBitmap (w, h, x0, y0, xShift, yShift, pattern);</code>	Especifica un patrón binario que hay que asignar a una serie de posiciones de píxeles, de forma relativa a la posición actual.
<code>glDrawPixels (w, h, type, format, pattern);</code>	Especifica un patrón de colores que hay que asignar a una serie de posiciones de píxeles, de forma relativa a la posición actual.
<code>glDrawBuffer</code>	Selecciona uno o más búferes para almacenar un mapa de píxeles.
<code>glReadPixels</code>	Guarda un bloque de píxeles en una matriz seleccionada.
<code>glCopyPixels</code>	Copia un bloque de píxeles de una posición de búfer a otra.
<code>glLogicOp</code>	Selecciona una operación lógica para combinar dos matrices de píxeles, después de habilitar esta característica con la constante <code>GL_COLOR_LOGIC_OP</code> .
<code>glutBitmapCharacter (font, char);</code>	Especifica una fuente y un carácter de mapa de bits para su visualización.
<code>glutStrokeCharacter (font, char);</code>	Especifica una fuente y un carácter de contorno para su visualización.
<code>glutReshapeFunc</code>	Especifica las acciones que hay que tomar cuando se modifican las dimensiones de la ventana de visualización.

Cada polígono tiene una cara anterior y una cara posterior, las cuales determinan la orientación espacial del plano del polígono. Esta orientación espacial puede determinarse a partir del vector normal, que es perpendicular al plano del polígono y apunta en la dirección que va de la cara posterior a la cara anterior. Podemos determinar las componentes del vector normal a partir de la ecuación del plano del polígono o realizando un producto vectorial utilizando tres puntos en el plano, tomándose los tres puntos en sentido contrario.

rio a las agujas del reloj y asegurándose de que el ángulo formado por los tres punto sea inferior a  $180^\circ$ . Todos los valores de coordenadas, las orientaciones espaciales y otros datos geométricos para una escena se introducen en tres tablas: tabla de vértices, tabla de aristas y tabla de caras de la superficie.

Entre las primitivas adicionales disponibles en los paquetes gráficos se incluyen las relativas a las matrices de patrones de bits o píxeles y a las cadenas de caracteres. Las matrices de patrones pueden utilizarse para especificar formas geométricas bidimensionales, incluyendo un conjunto de caracteres, utilizando un conjunto rectangular de valores binarios o un conjunto de valores de color. Las cadenas de caracteres se emplean para etiquetar las imágenes y los gráficos.

Utilizando las funciones primitivas disponibles en la biblioteca OpenGL básica, podemos generar puntos, segmentos de línea recta, áreas de relleno poligonales convexas y matrices de patrones de mapas de bits o mapas de píxeles. GLUT dispone de rutinas, para visualizar cadenas de caracteres. Otros tipos de primitivas, como círculos, elipses y áreas de relleno poligonales cóncavas, pueden construirse o aproximarse utilizando estas funciones, o bien pueden generarse empleando las rutinas disponibles en GLU y GLUT. Todos los valores de coordenadas se expresan en coordenadas absolutas dentro de un sistema de referencia cartesiano que cumpla la regla de la mano derecha. Las coordenadas que describen una escena pueden proporcionarse dentro de un marco de referencia bidimensional o tridimensional. Podemos utilizar valores enteros o de coma flotante para especificar unas coordenadas, y también podemos hacer referencia a una posición utilizando un puntero a una matriz de valores de coordenadas. La descripción de una escena se transforma entonces mediante una serie de funciones de visualización, para obtener una imagen bidimensional sobre un dispositivo de salida, como por ejemplo un monitor de vídeo. Excepto por la función `glRect`, las coordenadas para un conjunto de puntos, líneas o polígonos se especifican mediante la función `glVertex`, y el conjunto de funciones `glVertex` que definen cada primitiva se incluye entre una pareja de instrucciones `glBegin/glEnd`, donde el tipo de la primitiva se identifica mediante una constante simbólica que se introduce como argumento de la función `glBegin`. A la hora de describir una escena que contenga muchas superficies de relleno poligonales, podemos generar de manera eficiente la imagen utilizando matrices de vértices OpenGL para especificar los datos geométricos y de otro tipo.

En la Tabla 3.1 se enumeran las funciones básicas para generar primitivas de salida en OpenGL. También se incluyen en la tabla algunas rutinas relacionadas.

## **PROGRAMAS DE EJEMPLO**

Vamos a presentar aquí unos cuantos ejemplos de programas OpenGL que ilustran el uso de las primitivas de salida. Cada programa emplea una o más de las funciones enumeradas en la Tabla 3.1. En cada programa se configura una ventana de visualización para la salida utilizando las rutinas GLUT de las que hemos hablado en el Capítulo 2.

El primer programa ilustra el uso de una polilínea, un conjunto de polimarcadores y de etiquetas basadas en caracteres de mapa de bits para generar un gráfico lineal que representa una serie de datos mensuales a lo largo de un período de un año. Se ilustra el uso de una fuente con espaciado proporcional, aunque las fuentes de anchura fija suelen ser más fáciles de alinear con las posiciones correspondientes de las gráficas. Puesto que los mapas de bits son referenciados con respecto a la esquina inferior izquierda por la función de posición actual de visualización, debemos desplazar la posición de referencia para alinear el centro de las cadenas de texto con las posiciones dibujadas de los datos. La Figura 3.67 muestra la salida del programa de generación de la gráfica lineal.

```

GLint xRaster = 25, yRaster = 150;           // Inicializar posición de visualización.

GLubyte label [36] = {'J', 'a', 'n', 'F', 'e', 'b', 'M', 'a', 'r',
                     'A', 'p', 'r', 'M', 'a', 'y', 'J', 'u', 'n',
                     'J', 'u', 'l', 'A', 'u', 'g', 'S', 'e', 'p',
                     'O', 'c', 't', 'N', 'o', 'v', 'D', 'e', 'c'};

GLint dataValue [12] = {420, 342, 324, 310, 262, 185,
                       190, 196, 217, 240, 312, 438};

void init (void)
{
    glClearColor (1.0, 1.0, 1.0, 1.0);      // Ventana de visualización blanca.
    glMatrixMode (GL_PROJECTION);
    gluOrtho2D (0.0, 600.0, 0.0, 500.0);
}

void lineGraph (void)
{
    GLint month, k;
    GLint x = 30;                         // Inicializar posición x para la gráfica.

    glClear (GL_COLOR_BUFFER_BIT);          // Borrar ventana de visualización.

    glColor3f (0.0, 0.0, 1.0);             // Seleccionar azul como color de línea.
    glBegin (GL_LINE_STRIP);              // Dibujar los datos como una polilínea.
    for (k = 0; k < 12; k++)
        glVertex2i (x + k*50, dataValue [k]);
    glEnd ( );

    glColor3f (1.0, 0.0, 0.0);             // Establecer el rojo como color de marcador.
    for (k = 0; k < 12; k++) {            // Dibujar los datos mediante polimarcadores
        // de asterisco.
        glRasterPos2i (xRaster + k*50, dataValue [k] - 4);
        glutBitmapCharacter (GLUT_BITMAP_9_BY_15, '*');
    }

    glColor3f (0.0, 0.0, 0.0);             // Establecer el negro como color del texto.
    xRaster = 20;                         // Mostrar etiquetas de la gráfica.

    for (month = 0; month < 12; month++) {
        glRasterPos2i (xRaster, yRaster);
        for (k = 3*month; k < 3*month + 3; k++)
            glutBitmapCharacter (GLUT_BITMAP_HELVETICA_12, label [k]);
        xRaster += 50;
    }
    glFlush ( );
}

void winReshapeFcn (GLint newWidth, GLint newHeight)
{
    glMatrixMode (GL_PROJECTION);
}

```

```

glLoadIdentity ( );
glOrtho2D (0.0, GLdouble (newWidth), 0.0, GLdouble (newHeight));

glClear (GL_COLOR_BUFFER_BIT);
}

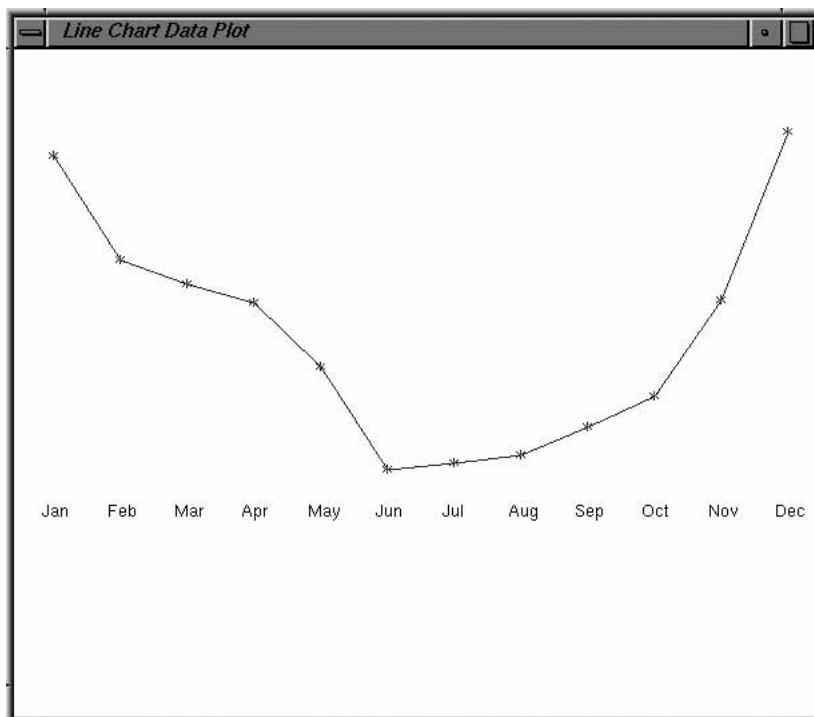
void main (int argc, char** argv)
{
    glutInit (&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    glutInitWindowPosition (100, 100);
    glutInitWindowSize (winWidth, winHeight);
    glutCreateWindow («Line Chart Data Plot»);

    init ();
    glutDisplayFunc (lineGraph);
    glutReshapeFunc (winReshapeFcn);

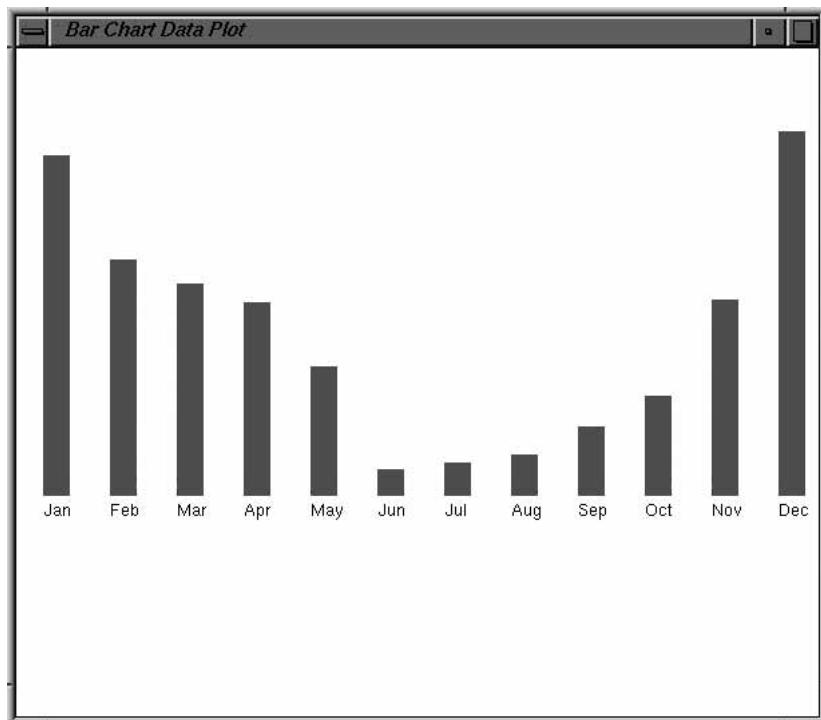
    glutMainLoop ();
}

```

Vamos a utilizar el mismo conjunto de datos en el segundo programa con el fin de generar la gráfica de barras de la Figura 3.68. El programa ilustra la aplicación de las áreas de relleno rectangulares, así como de las etiquetas de caracteres basadas en mapas de bits.



**FIGURA 3.67.** Una representación de puntos de datos mediante polilíneas y polimarcadores, generada mediante la rutina lineGraph.



**FIGURA 3.68.** Una gráfica de barras generada mediante el procedimiento barChart.

```

void barChart (void)
{
    GLint month, k;

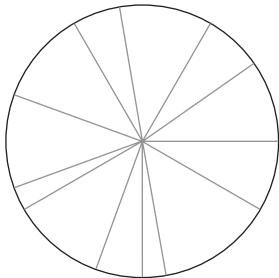
    glClear (GL_COLOR_BUFFER_BIT); // Borrar ventana de visualización.

    glColor3f (1.0, 0.0, 0.0);      // Color rojo para las barras.
    for (k = 0; k < 12; k++)
        glRecti (20 + k*50, 165, 40 + k*50, dataValue [k]);

    glColor3f (0.0, 0.0, 0.0);      // Color negro para el texto.
    xRaster = 20;                  // Mostrar etiquetas de la gráfica.
    for (month = 0; month < 12; month++) {
        glRasterPos2i (xRaster, yRaster);
        for (k = 3*month; k < 3*month + 3; k++)
            glutBitmapCharacter (GLUT_BITMAP_HELVETICA_12,
                                  label [h]);
        xRaster += 50;
    }
    glFlush ( );
}

```

Las gráficas de sectores circulares se utilizan para mostrar la contribución porcentual de una serie de partes individuales a un todo. El siguiente programa construye una gráfica de sectores circulares, utilizando la rutina del punto medio para generar el círculo. Se utilizan valores de ejemplo para el número y los tamaños relativos de los distintos sectores, y la salida del programa se muestra en la Figura 3.69.



**FIGURA 3.69.** Salida generada mediante el procedimiento pieChart.

```

        for (k = 0; k < nSlices; k++)
            dataSum += dataValues[k];

        for (k = 0; k < nSlices; k++) {
            sliceAngle = twoPi * dataValues[k] / dataSum + previousSliceAngle;
            piePt.x = circCtr.x + radius * cos (sliceAngle);
            piePt.y = circCtr.y + radius * sin (sliceAngle);
            glBegin (GL_LINES);
                glVertex2i (circCtr.x, circCtr.y);
                glVertex2i (piePt.x, piePt.y);
            glEnd ( );
            previousSliceAngle = sliceAngle;
        }
    }

    void displayFcn (void)
    {
        glClear (GL_COLOR_BUFFER_BIT); // Borrar ventana de visualización.

        glColor3f (0.0, 0.0, 1.0); // Seleccionar azul como color del círculo.

        pieChart ( );
        glFlush ( );
    }

    void winReshapeFcn (GLint newWidth, GLint newHeight)
    {
        glMatrixMode (GL_PROJECTION);
        glLoadIdentity ( );
        gluOrtho2D (0.0, GLdouble (newWidth), 0.0, GLdouble (newHeight));

        glClear (GL_COLOR_BUFFER_BIT);

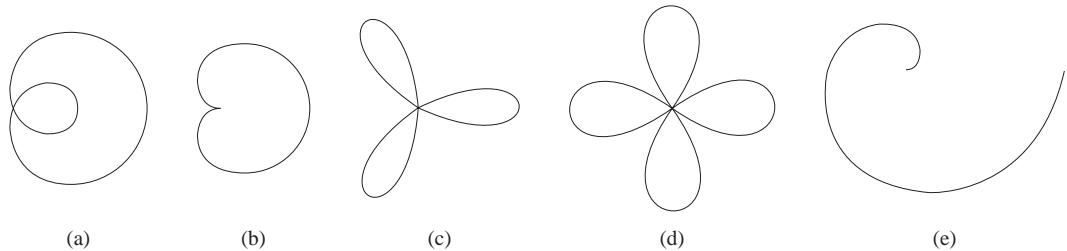
        /* Reinicializar parámetros de tamaño de la ventana de visualización. */
        winWidth = newWidth;
        winHeight = newHeight;
    }

    void main (int argc, char** argv)
    {
        glutInit (&argc, argv);
        glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
        glutInitWindowPosition (100, 100);
        glutInitWindowSize (winWidth, winHeight);
        glutCreateWindow («Pie Chart»);

        init ( );
        glutDisplayFunc (displayFcn);
        glutReshapeFunc (winReshapeFcn);

        glutMainLoop ( );
    }
}

```



**FIGURA 3.70.** Figuras curvas generadas por el procedimiento `drawCurve`: (a) caracola, (b) cardioide, (c) trébol de tres hojas, (d) trébol de cuatro hojas y (e) espiral.

Nuestro último programa de ejemplo genera algunas variantes de las ecuaciones del círculo, utilizando las ecuaciones paramétricas polares (3.28) para calcular los puntos a lo largo de las trayectorias curvas. Estos puntos se emplean entonces como extremos de una serie de secciones lineales, mostrándose las curvas mediante polilíneas que las aproximan. Las curvas mostradas en la Figura 3.70 se pueden generar haciendo variar el radio  $r$  de un círculo. Dependiendo del modo en que varíemos  $r$ , podemos generar una cardioide, una espiral, un trébol y otras figuras similares.

```
#include <GL/glut.h>
#include <stdlib.h>
#include <math.h>

#include <iostream.h>

struct screenPt
{
    GLint x;
    GLint y;
};

typedef enum { limacon = 1, cardioid, threeLeaf, fourLeaf, spiral } curveName;

GLsizei winWidth = 600, winHeight = 500; // Tamaño inicial de la ventana de
                                         // visualización.

void init (void)
{
    glClearColor (1.0, 1.0, 1.0, 1.0);
    glMatrixMode (GL_PROJECTION);
    gluOrtho2D (0.0, 200.0, 0.0, 150.0);
}

void lineSegment (screenPt pt1, screenPt pt2)
{
    glBegin (GL_LINES);
        glVertex2i (pt1.x, pt1.y);
        glVertex2i (pt2.x, pt2.y);
    glEnd ( );
}

void drawCurve (GLint curveNum)
{
```

```

/*
 * La caracola de Pascal es una modificación de la ecuación del círculo
 * en la que el radio varía con la fórmula  $r = a * \cos(\thetaeta) + b$ , donde
 * a y b son constantes. Una cardioide es una caracola con a = b.
 * Los tréboles de tres y cuatro hojas se generan cuando
 *  $r = a * \cos(n * \thetaeta)$ , con n = 3 y n = 2, respectivamente.
 * La espiral se muestra cuando r es un múltiplo de thetaeta.
 */

const GLdouble twoPi = 6.283185;
const GLint a = 175, b = 60;

GLfloat r, theta, dtheta = 1.0 / float (a);
GLint x0 = 200, y0 = 250; // Establecer posición inicial en la pantalla.
screenPt curvePt[2];

	glColor3f (0.0, 0.0, 0.0); // Seleccionar negro como color de curva.

curvePt[0].x = x0; // Inicializar posición de la curva.
curvePt[0].y = y0;

switch (curveNum) {
    case limacon:      curvePt[0].x += a + b; break;
    case cardioid:     curvePt[0].x += a + a; break;
    case threeLeaf:   curvePt[0].x += a; break;
    case fourLeaf:    curvePt[0].x += a; break;
    case spiral:       break;
    default:           break;
}

theta = dtheta;
while (theta < two_Pi) {
    switch (curveNum) {
        case limacon:
            r = a * cos (theta) + b;      break;
        case cardioid:
            r = a * (1 + cos (theta));  break;
        case threeLeaf:
            r = a * cos (3 * theta);   break;
        case fourLeaf:
            r = a * cos (2 * theta);   break;
        case spiral:
            r = (a / 4.0) * theta;    break;
        default:                  break;
    }

    curvePt[1].x = x0 + r * cos (theta);
    curvePt[1].y = y0 + r * sin (theta);
    lineSegment (curvePt[0], curvePt[1]);

    curvePt[0].x = curvePt[1].x;
    curvePt[0].y = curvePt[1].y;
    theta += dtheta;
}

```

```

        }
    }

void displayFcn (void)
{
    GLint curveNum;

    glClear (GL_COLOR_BUFFER_BIT); // Borrar ventana de visualización.

    cout << "\nEnter the integer value corresponding to\n";
    cout << "one of the following curve names.\n";
    cout << "Press any other key to exit.\n";
    cout << "\nl-limacon, 2-cardioid, 3-threeLeaf, 4-fourLeaf, 5-spiral: ";
    cin >> curveNum;

    if (curveNum == 1 || curveNum == 2 || curveNum == 3 || curveNum == 4
        || curveNum == 5)
        drawCurve (curveNum);
    else
        exit (0);
    glFlush ( );
}

void winReshapeFcn (GLint newWidth, GLint newHeight)
{
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ( );
    gluOrtho2D (0.0, (GLdouble) newWidth, 0.0, (GLdouble) newHeight);

    glClear (GL_COLOR_BUFFER_BIT);
}

void main (int argc, char** argv)
{
    glutInit (&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    glutInitWindowPosition (100, 100);
    glutInitWindowSize (winWidth, winHeight);
    glutCreateWindow («Draw Curves»);

    init ( );
    glutDisplayFunc (displayFcn);
    glutReshapeFunc (winReshapeFcn);

    glutMainLoop ( );
}

```

## REFERENCIAS

Puede encontrar información básica sobre los algoritmos de Bresenham en Bresenham (1965 y 1977). Para los métodos del punto medio, consulte Kappel (1985). Los métodos paralelos para la generación de líneas y círculos se tratan en Pang (1990) y en Wright (1990). En tres Glassner (1990), Arvo (1991), Kirk (1992),

Heckbert (1994) y Paeth (1995) se explican muchos otros métodos para la generación y el procesamiento de primitivas gráficas.

En Woo, Neider, Davis y Shreiner (1999) se proporcionan ejemplos adicionales de programación en los que se utilizan las funciones primitivas de OpenGL. Puede encontrar un listado de todas las funciones primitivas de OpenGL en Shreiner (2000). Si necesita consultar una referencia completa de GLUT, le recomendamos Kilgard (1996).

## EJERCICIOS

---

- 3.1 Implemente una función de polilínea utilizando el algoritmo DDA, dado un número cualquiera  $n$  de puntos de entrada. Cuando  $n = 1$  se debe mostrar un único punto.
- 3.2 Amplíe el algoritmo de generación de líneas de Bresenham para generar líneas de cualquier pendiente, teniendo en cuenta la simetría entre cuadrantes.
- 3.3 Implemente una función de polilínea utilizando el algoritmo del ejercicio anterior, con el fin de mostrar un conjunto de líneas rectas que conectan una lista de  $n$  puntos de entrada. Para  $n = 1$ , la rutina debe mostrar un único punto.
- 3.4 Utilice el método del punto medio para calcular los parámetros de decisión necesarios para generar los puntos a lo largo de una trayectoria recta cuya pendiente esté comprendida en el rango  $0 < m < 1$ . Demuestre que los parámetros de decisión para el algoritmo del punto medio son iguales que para el algoritmo de generación de líneas de Bresenham.
- 3.5 Utilice el método del punto medio para calcular los parámetros de decisión que pueden emplearse para generar segmentos de línea recta con cualquier pendiente.
- 3.6 Diseñe una versión paralela del algoritmo de generación de líneas de Bresenham para pendientes comprendidas en el rango  $0 < m < 1$ .
- 3.7 Diseñe una versión paralela del algoritmo de Bresenham para generación de líneas rectas con pendiente arbitraria.
- 3.8 Suponga que dispone de un sistema con un monitor de vídeo de 8 por 10 pulgadas que puede mostrar 100 píxeles por pulgada. Si la memoria está organizada en palabras de un byte, si la dirección de inicio del búfer de imagen es 0 y si a cada píxel se le asigna un byte de almacenamiento, ¿cuál será la dirección del búfer de imagen correspondiente al píxel con coordenadas de pantalla  $(x, y)$ ?
- 3.9 Suponga que tiene un sistema con un monitor de vídeo de 8 por 10 pulgadas que puede mostrar 100 píxeles por pulgada. Si la memoria está organizada en palabras de un byte, si la dirección de inicio del búfer de imagen es 0 y si a cada píxel se le asignan 6 bits de almacenamiento, ¿cuál será la dirección (o direcciones) del búfer de imagen correspondiente al píxel con coordenadas de pantalla  $(x, y)$ ?
- 3.10 Incorpore las técnicas iterativas de cálculo de las direcciones del búfer de imagen (Sección 3.7) al algoritmo de generación de líneas de Bresenham.
- 3.11 Revise el algoritmo del punto medio para generación de círculos con el fin de mostrar círculos en donde se preserven las magnitudes geométricas de entrada (Sección 3.13).
- 3.12 Diseñe un procedimiento para la implementación paralela del algoritmo del punto medio para generación de círculos.
- 3.13 Calcule los parámetros de decisión para el algoritmo del punto medio de generación de elipses, suponiendo que la posición de inicio sea  $(r_x, 0)$  y que los puntos deben generarse a lo largo de la trayectoria curva en sentido contrario a las agujas del reloj.
- 3.14 Diseñe un procedimiento para la implementación paralela del algoritmo del punto medio para generación de elipses.

- 3.15 Diseñe un algoritmo eficiente que aproveche las propiedades de simetría para mostrar un ciclo completo de una función seno.
- 3.16 Modifique el algoritmo del ejercicio anterior para mostrar la función seno para cualquier intervalo angular especificado.
- 3.17 Diseñe un algoritmo eficiente, teniendo en cuenta la simetría de la función, para mostrar una gráfica del movimiento armónico amortiguado:

$$y = Ae^{-kx} \sin(\omega x + \theta)$$

donde  $\omega$  es la frecuencia angular y  $\theta$  es la fase de la función seno. Dibuje  $y$  en función de  $x$  para varios ciclos de la función seno, o hasta que la amplitud máxima se reduzca a  $\frac{A}{10}$ .

- 3.18 Utilizando el método del punto medio y teniendo en cuenta la simetría, diseñe un algoritmo eficiente para la digitalización de la siguiente curva en el intervalo  $-10 \leq x \leq 10$ .

$$y = \frac{1}{12}x^3$$

- 3.19 Utilice el método del punto medio y las consideraciones de simetría para digitalizar la parábola:

$$y = 100 - x^2$$

en el intervalo  $-10 \leq x \leq 10$ .

- 3.20 Utilice el método del punto medio y las consideraciones de simetría para digitalizar la parábola:

$$x = y^2$$

en el intervalo  $-10 \leq y \leq 10$ .

- 3.21 Diseñe un algoritmo del punto medio, teniendo presentes las consideraciones de simetría, con el fin de digitalizar cualquier parábola de la forma:

$$y = ax^2 + b$$

con valores de entrada que especifique los parámetros  $a$ ,  $b$  y el rango de  $x$ .

- 3.22 Defina unas tablas de datos geométricos como las de la Figura 3.50 para un cubo de lado unidad.
- 3.23 Defina las tablas de datos geométricos para un cubo de lado unidad utilizando simplemente una tabla de vértices y una tabla de caras de la superficie, y luego almacene la información empleando únicamente la tabla de caras de la superficie. Compare los dos métodos de representación del cubo de lado unidad con una representación que utilice las tres tablas del Ejercicio 3.22. Estime los requisitos de almacenamiento de cada una de las soluciones.
- 3.24 Defina una representación eficiente en forma de malla de polígonos para un cilindro y explique por qué ha elegido dicha representación.
- 3.25 Defina un procedimiento para establecer las tablas de datos geométricos correspondientes a cualquier conjunto de puntos de entrada que definen las caras poligonales correspondientes a la superficie de un objeto tridimensional.
- 3.26 Diseñe las rutinas necesarias para comprobar las tres tablas de datos geométricos de la Figura 3.50 con el fin de garantizar su coherencia y exhaustividad.
- 3.27 Escriba un programa para calcular los parámetros  $A$ ,  $B$ ,  $C$  y  $D$  para una malla de entrada que define las caras poligonales de una superficie.
- 3.28 Escriba un procedimiento para determinar si un determinado punto suministrado como entrada se encuentra delante o detrás de una superficie poligonal, dados los parámetros del plano correspondiente al polígono,  $A$ ,  $B$ ,  $C$  y  $D$ .
- 3.29 Si la referencia de coordenadas de una escena se cambia de modo que en lugar de cumplir la regla de la mano derecha se cumple la regla de la mano izquierda, ¿qué cambios podríamos hacer en los valores de los parámetros  $A$ ,  $B$ ,  $C$  y  $D$  de un plano de la superficie para garantizar que la orientación del plano esté descrita correctamente?
- 3.30 Diseñe un procedimiento para identificar una lista de vértices no planar para un cuadrilátero.

- 3.31 Amplíe el algoritmo del ejercicio anterior para identificar una lista de vértices no planares que contengan más de cuatro puntos.
- 3.32 Escriba un procedimiento para dividir un conjunto de cuatro vértices de un polígono en un conjunto de triángulos.
- 3.33 Diseñe un algoritmo para dividir un conjunto de  $n$  vértices de un polígono, con  $n > 4$ , en un conjunto de triángulos.
- 3.34 Diseñe un algoritmo para identificar una lista de vértices de un polígono degenerado que pueda contener vértices repetidos o vértices colineales.
- 3.35 Diseñe un algoritmo para identificar una lista de vértices de un polígono que contenga aristas que se intersecten.
- 3.36 Escriba una rutina para la identificación de polígonos cóncavos calculando el producto vectorial de parejas de vectores de las aristas.
- 3.37 Escriba una rutina para dividir un polígono cóncavo, utilizando el método vectorial.
- 3.38 Escriba una rutina para dividir un polígono cóncavo, utilizando el método rotacional.
- 3.39 Diseñe un algoritmo para determinar las regiones interiores correspondientes a cualquier conjunto de vértices de entrada, utilizando la regla del número de vueltas distinto de cero y los cálculos de productos vectoriales para identificar la dirección en los cortes de las aristas.
- 3.40 Diseñe un algoritmo para determinar las regiones interiores correspondientes a cualquier conjunto de vértices de entrada, utilizando la regla del número de vueltas distinto de cero y los cálculos de producto escalar para identificar la dirección en los cortes de las aristas.
- 3.41 ¿Qué regiones de la polilínea auto-intersectante mostrada en la Figura 3.46 tienen un número de vueltas positivo? ¿Cuáles son las regiones que tienen un número de vueltas negativo? ¿Qué regiones tienen un número de vueltas superior a 1?
- 3.42 Escriba una rutina para implementar una función de generación de cadenas de texto que tenga dos parámetros: uno de ellos especifica una posición en coordenadas universales y el otro especifica una cadena de texto.
- 3.43 Escriba una rutina para implementar una función de generación de polimarcadores que tenga dos parámetros: uno de los parámetros es el carácter que hay que mostrar y el otro es una lista de posiciones en coordenadas universales.
- 3.44 Modifique el programa de ejemplo de la Sección 3.24 para que el hexágono mostrado esté siempre en el centro de la ventana de visualización, independientemente de cómo se cambie el tamaño de ésta.
- 3.45 Escriba un programa completo para mostrar una gráfica de barras. Los datos de entrada al programa incluirán los puntos de datos y las etiquetas requeridas para los ejes  $x$  e  $y$ . El programa deberá cambiar la escala de los puntos de datos para mostrar la gráfica de modo que ocupe toda el área de la ventana de visualización.
- 3.46 Escriba un programa para mostrar una gráfica de barras en cualquier área seleccionada de una ventana de visualización.
- 3.47 Escriba un procedimiento para mostrar una gráfica de líneas para cualquier conjunto de puntos de datos de entrada en cualquier área seleccionada de la pantalla, cambiando la escala del conjunto de datos de entrada con el fin de que encaje en el área de pantalla elegida. Los puntos de datos deben mostrarse mediante asteriscos unidos por segmentos de línea recta, y los ejes  $x$  e  $y$  deben etiquetarse de acuerdo con las especificaciones de entrada (en lugar de asteriscos, puede utilizar pequeños círculos o algún otro símbolo para dibujar los puntos de datos).
- 3.48 Utilizando una función de generación de círculos, escriba una rutina para mostrar una gráfica de sectores circulares con las etiquetas apropiadas. Los datos de entrada a la rutina incluirán un conjunto de datos que indiquen la distribución de éstos a lo largo de un conjunto de intervalos, el nombre de la gráfica de sectores circulares y los nombres de cada uno de los intervalos. La etiqueta de cada sección deberá mostrarse fuera del contorno de la gráfica de sectores circulares, en las proximidades del sector correspondiente.



# Atributos de las primitivas gráficas



Una película de dibujos animados generada por computadora que muestra varios objetos con color y otros atributos. (Cortesía de Softimage, Inc.)

- |   |  |
|---|--|
| <ul style="list-style-type: none"><li>4.1 Variables de estado de OpenGL</li><li>4.2 Color y escala de grises</li><li>4.3 Funciones de color de OpenGL</li><li>4.4 Atributos de los puntos</li><li>4.5 Atributos de las líneas</li><li>4.6 Atributos de las curvas</li><li>4.7 Funciones OpenGL para los atributos del punto</li><li>4.8 Funciones OpenGL para los atributos de las líneas</li><li>4.9 Atributos de relleno de áreas</li><li>4.10 Algoritmo general de relleno de polígonos mediante líneas de barrido</li><li>4.11 Relleno de polígonos convexos mediante líneas de barrido</li></ul> | <ul style="list-style-type: none"><li>4.12 Relleno de regiones con límites curvos mediante líneas de barrido</li><li>4.13 Métodos de relleno de áreas con límites irregulares</li><li>4.14 Funciones OpenGL para los atributos de relleno de áreas</li><li>4.15 Atributos de los caracteres</li><li>4.16 Funciones OpenGL para los atributos de los caracteres</li><li>4.17 Suavizado (<i>antialiasing</i>)</li><li>4.18 Funciones de suavizado de OpenGL</li><li>4.19 Funciones de consulta de OpenGL</li><li>4.20 Grupos de atributos de OpenGL</li><li>4.21 Resumen</li></ul> |
|---|--|

Por lo general, un parámetro que afecta a la forma en que una primitiva se va a visualizar se denomina **parámetro de atributo**. Algunos parámetros de atributo, tales como el color y el tamaño, determinan características fundamentales de una primitiva. Otros atributos especifican cómo se visualizará la primitiva en condiciones especiales. Las opciones tales como la visibilidad o la detectabilidad en un programa de selección de objetos interactivo son ejemplos de atributos de condiciones especiales. Los atributos de condiciones especiales se estudiarán en capítulos posteriores. En este capítulo, sólo estudiaremos aquellos atributos que controlan las propiedades básicas de visualización de las primitivas gráficas, sin tener en cuenta las situaciones especiales. Por ejemplo, las líneas pueden ser punteadas o discontinuas, gruesas o finas, y azules o naranjas. Las áreas se pueden llenar con un color o con un patrón multicolor. El texto puede parecer con un sentido de lectura de izquierda a derecha, inclinado según la diagonal de la pantalla, o en columnas verticales. Los caracteres individuales se pueden visualizar con diferentes tipos de letra, colores y tamaños. Y podemos aplicar variaciones de la intensidad a los bordes de los objetos para suavizar el efecto de escalonamiento en la representación.

Un modo de incorporar parámetros de atributos en un paquete gráfico consiste en ampliar la lista de parámetros asociados a cada función de primitiva gráfica para incluir los valores de los atributos apropiados. Por ejemplo, una función para dibujar líneas, podría contener parámetros adicionales para establecer el color, el grosor y otras propiedades de la línea. Otro enfoque consiste en mantener una lista de sistema de los valores actuales de los atributos. Se incluyen funciones independientes en el paquete gráfico para establecer los valores actuales en la lista de atributos. Para generar una primitiva, el sistema comprueba los atributos relevantes e invoca la subrutina de visualización para dicha primitiva utilizando los valores actuales de los atributos. Algunos paquetes gráficos utilizan una combinación de métodos para establecer los colores de los atributos, y otras bibliotecas, entre las que se incluye OpenGL, asignan los atributos utilizando funciones independientes que actualizan la lista de atributos del sistema.

Un sistema gráfico que mantiene una lista de los valores actuales de los atributos y otros parámetros se denomina **sistema de estados** o **máquina de estados**. Los atributos de las primitivas de salida y algunos otros parámetros, tales como la posición actual dentro del búfer de imagen se denominan **variables de estado** o

**parámetros de estado.** Cuando asignamos un valor a uno o más parámetros de estado, ponemos el sistema en un estado determinado. Y el efecto de este estado permanece hasta que cambiamos el valor de los parámetros de estado.

## 4.1 VARIABLES DE ESTADO DE OpenGL

---

Los valores de los atributos y otros parámetros se especifican con funciones independientes que definen el estado actual de OpenGL. Entre los parámetros de estado de OpenGL se incluye el color y otros atributos de primitivas, el modo actual de matriz, los elementos de la matriz *modelview* (vista de modelo), la posición actual del búfer de imagen y los parámetros para los efectos de iluminación de una escena. Todos los parámetros de estado de OpenGL tienen valores predeterminados, que se mantienen hasta que se especifican nuevos valores. En cualquier momento, podemos preguntar al sistema por el valor actual de un parámetro de estado. En las siguientes secciones de este capítulo, estudiaremos sólo los valores de los atributos para las primitivas de salida. Otros parámetros de estado se estudian en capítulos posteriores.

Todas las primitivas gráficas en OpenGL se visualizan con los atributos definidos en la lista de estados actual. El cambio de uno o más de los valores de los atributos afecta sólo a aquellas primitivas que se especifican después de que el estado de OpenGL haya cambiado. Las primitivas que se definieron antes del cambio de estado mantienen sus atributos. Por tanto, podemos visualizar una línea verde, cambiar el color actual a rojo y definir otro segmento de línea. Ambas líneas, la verde y la roja, se visualizarán. También, algunos valores de estado de OpenGL se pueden especificar en pares `glBegin/glEnd` junto con los valores de las coordenadas, para que los valores de los parámetros puedan variar de unas coordenadas a otras.

## 4.2 COLOR Y ESCALA DE GRISES

---

Un atributo básico de todas las primitivas es el color. Se puede hacer que varias opciones de color estén disponibles para el usuario, dependiendo de las capacidades y de los objetivos de diseño de un sistema particular. Las opciones de color se pueden especificar numéricamente, seleccionar en un menú o visualizar en una escala con un cursor. En un monitor de vídeo, estos códigos de color se convierten en valores de nivel de intensidad de los haces de electrones. En los trazadores de color, los códigos permiten controlar los depósitos de inyección de tinta o seleccionar las plumillas.

### Las componentes de color RGB

En un sistema de barrido a color, el número de las alternativas de color disponibles depende de la cantidad de almacenamiento proporcionado por píxel en el búfer de imagen. Además, la formación sobre el color se puede almacenar en el búfer de imagen de dos modos: podemos almacenar directamente en el búfer de imagen los códigos de color RGB, o podemos poner los códigos de color en una tabla independiente y usar las posiciones de los píxeles para almacenar los valores de los índices que hacen referencia a las entradas de la tabla de color. En el esquema de almacenamiento directo, cada vez que un código de color concreto se especifica en un programa de aplicación, la información de color se coloca en el búfer de imagen en la posición de cada píxel componente en las primitivas para mostrar dicho color. Se puede proporcionar un número mínimo de colores en este esquema con tres bits de almacenamiento por píxel, como se muestra en la Tabla 4.1. Cada uno de estos tres bits se utiliza para controlar el nivel de intensidad (encendido o apagado, en este caso) del cañón de electrones correspondiente en un monitor RGB. El bit situado más a la izquierda controla el cañón rojo, el bit central controla el cañón verde y el bit situado más a la derecha el cañón azul. Podemos aumentar el número de alternativas de color disponibles añadiendo más bits por píxel al búfer de imagen. Con seis bits por píxel, se pueden utilizar dos bits para cada cañón. Esto permite cuatro valores de intensidad diferentes para cada uno de los tres cañones de color, y un total de 64 opciones de color disponibles para cada píxel de pantalla. A medida que se proporcionan más opciones de color, el almacenamiento

**TABLA 4.1.** LOS OCHO CÓDIGOS DE COLOR RGB DE UN BÚFER DE IMAGEN DE 3 BITS POR PÍXEL.

<i>Valores de color almacenados en el búfer de imagen</i>				
<i>Código de color</i>	<i>ROJO</i>	<i>VERDE</i>	<i>AZUL</i>	<i>Color mostrado</i>
0	0	0	0	Negro
1	0	0	1	Azul
2	0	1	0	Verde
3	0	1	1	Cíán
4	1	0	0	Rojo
5	1	0	1	Magenta
6	1	1	0	Amarillo
7	1	1	1	Blanco

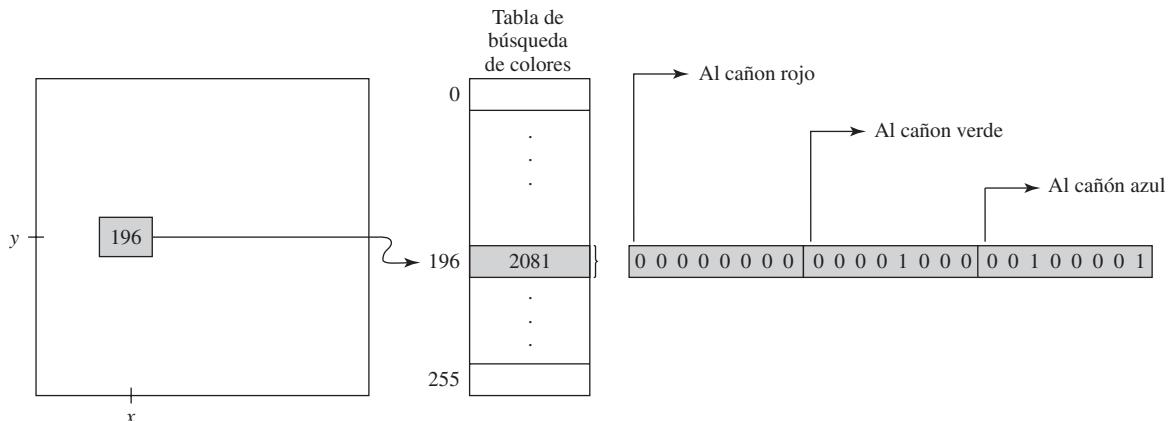
requerido por el búfer de imagen también aumenta. Con una resolución de 1024 por 1024, un sistema de color total (24 bits por píxel) RGB necesita 3 megabytes de almacenamiento para el búfer de imagen.

Las tablas de color son un medio alternativo para proporcionar a un usuario capacidades de color ampliadas sin requerir búferes de imagen grandes. Hace tiempo, era importante tener esto en consideración. Pero hoy en día, los costes del hardware se han reducido drásticamente y las capacidades de color ampliadas son bastante comunes, incluso en computadoras personales. Por tanto, en la mayoría de los ejemplos se asume que los códigos de color RGB se almacenan directamente en el búfer de imagen.

## Tablas de color

La Figura 4.1 muestra un posible esquema para almacenar los valores de color en una **tabla de búsqueda de color** (o **mapa de color**). A veces una tabla de color se denomina **tabla de búsqueda de vídeo**. Los valores que se almacenan en el búfer de imagen se utilizan ahora como índices en la tabla de color. Por ejemplo, cada píxel puede referenciar cualquiera de las 256 posiciones de la tabla, y cada entrada de la tabla utiliza 24 bits para especificar un color RGB. Para el código de color en hexadecimal 0x0821, se muestra una combinación de los colores verde y azul en la posición de píxel  $(x, y)$ . Los sistemas que emplean esta tabla de búsqueda concreta permiten a un usuario seleccionar de una paleta de, aproximadamente, 17 millones de colores 256 colores cualesquier para su visualización simultánea. En comparación con un sistema de color total, este diseño reduce el número de colores simultáneos que se pueden visualizar, pero también reduce los requisitos de almacenamiento del búfer de imagen a un megabyte. A veces se dispone de múltiples tablas de color para la manipulación de aplicaciones de sombreado específicas, tales como el suavizado (*antialiasing*) y aquellas que se utilizan con sistemas que contienen más de un dispositivo de salida de color.

Una tabla de color puede ser útil para un gran número de aplicaciones, y puede proporcionar un «razonable» número de colores simultáneos sin requerir búferes de imagen grandes. Para la mayoría de las aplicaciones, 256 ó 512 colores diferentes son suficientes para una única imagen. También, se pueden cambiar las entradas de la tabla en cualquier momento, permitiendo a un usuario ser capaz de experimentar fácilmente con diferentes combinaciones de color en un diseño, escena o gráfico sin cambiar la configuración de los atributos de la estructura de datos para los gráficos. Cuando se cambia el valor de un color en la tabla de color, todos los píxeles con ese índice de color se cambian inmediatamente al nuevo color. Cuando no se dispone de una tabla de color, sólo podemos cambiar el color de un píxel almacenando el nuevo color en su posición dentro del búfer de imagen. Del mismo modo, las aplicaciones para visualización de datos pueden almacenar valo-



**FIGURA 4.1.** Una tabla de búsqueda de color con 24 bits por entrada a la que se accede desde de un búfer de imagen con 8 bits por píxel. Un valor de 196 almacenado en la posición de píxel  $(x, y)$  hace referencia a la posición de esta tabla que contiene el valor hexadecimal 0x0821 (el valor decimal 2081). Cada segmento de 8 bits de esta entrada controla el nivel de intensidad de uno de los tres cañones de electrones de un monitor RGB.

res para una magnitud física, tal como la energía, en el búfer de imagen y utilizar una tabla de búsqueda para experimentar con varias combinaciones de color sin cambiar los valores del píxel. Y en aplicaciones de procesamiento de imágenes y visualización, las tablas de color son un medio conveniente para cambiar los umbrales de color para que todos los valores de píxel por encima o por debajo de un umbral específico se puedan cambiar al mismo color. Por estos motivos, algunos sistemas proporciona ambas capacidades de almacenamiento de la información del color. Un usuario puede entonces elegir entre usar las tablas de color o almacenar directamente los códigos de color en el búfer de imagen.

## Escala de grises

Ya que las capacidades de color son en la actualidad habituales en los sistemas de gráficos por computadora, utilizamos las funciones de color RGB para cambiar los niveles de gris, o **escala de grises**, en un programa de aplicación. Cuando en una configuración de color RGB se especifica una cantidad igual de los colores rojo, verde, y azul, el resultado es un nivel de gris. Valores próximos a 0 para las componentes de color producen gris oscuro, y valores más altos próximos a 1.0 producen gris claro. Entre las aplicaciones de los métodos de visualización en escala de grises se pueden incluir la mejora de fotogramas en blanco y negro y la generación de efectos de visualización.

## Otros parámetros de color

Además de la especificación RGB, son útiles en aplicaciones de gráficos por computadora otras representaciones de color de tres componentes. Por ejemplo, la salida de color en las impresoras se describe con las componentes de color cian, magenta y amarillo, y a veces las interfaces de color utilizan parámetros tales como el brillo o la oscuridad para elegir un color. También, el color, y la luz en general, es un asunto complejo, por lo que se han ideado en el campo de la óptica, la radiometría y la psicología muchos términos y conceptos para describir los múltiples aspectos de las fuentes de luz y los efectos de iluminación. Físicamente, podemos describir un color como una radiación electromagnética con un margen de frecuencias y una distribución de energía concretos, pero también influyen las características de la percepción de color de que se disponga. Por tanto, utilizamos el término físico *intensidad* para cuantificar la cantidad de energía luminosa radiante en una dirección concreta durante un período de tiempo, y utilizamos el término psicológico *luminancia* para caracterizar el brillo de la luz percibido. Estudiaremos estos términos y otros conceptos relacionados con el color

con un mayor grado de detalle cuando tengamos en consideración los métodos para el modelado de efectos de iluminación (Capítulo 10) y los múltiples modelos para la descripción del color (Capítulo 12).

## 4.3 FUNCIONES DE COLOR DE OpenGL

---

En el programa de ejemplo de la parte final del Capítulo 2, presentamos brevemente las subrutinas de color de OpenGL. Utilizamos una función para establecer el color de la ventana de visualización y otra función para especificar el color del segmento de línea recta. También establecemos el **modo de visualización de color** como RGB con la línea:

```
glutInitDisplayMode ( GLUT_SINGLE | GLUT_RGB );
```

El primer parámetro de la lista de argumentos establece que estamos utilizando un único búfer como búfer de imagen, y el segundo parámetro establece el modo RGB (o RGBA), que es el modo de color predeterminado. Podemos utilizar GLUT\_RGB o GLUT\_RGBA para seleccionar este modo de color. Si quisieramos especificar los colores mediante un índice en la tabla de color, reemplazaríamos la constante de OpenGL GLUT\_RGB por GLUT\_INDEX.

### Los modos de color RGB y RGBA de OpenGL

La mayor parte de los parámetros de configuración de color de las primitivas de OpenGL pertenecen al **modo RGB**, que es básicamente el mismo que el **modo RGBA**. La única diferencia entre RGB y RGBA consiste en la utilización del valor alfa para realizar el fundido de color. Cuando especificamos un conjunto concreto de valores de color, definimos el *estado de color* de OpenGL. El color actual se aplica a las primitivas posteriores hasta que cambiemos la configuración del color. Una especificación nueva de color afecta únicamente a los objetos que definamos con posterioridad al cambio de color.

En el modo RGB, especificamos valores para las componentes roja, verde y azul de un color. Como indicamos en el Sección 2.9, el cuarto parámetro del color, el **coeficiente alfa**, es opcional y una especificación de color de cuatro dimensiones se denomina color RGBA. Este cuarto parámetro del color se puede utilizar para controlar el fundido de color en las primitivas de superposición. Una aplicación importante del fundido de color es la simulación de los efectos de transparencia. En estos cálculos, el valor alfa se corresponde con la transparencia (u opacidad). En el modo RGB (o RGBA), seleccionamos las componentes de color actuales con la función:

```
	glColor* (colorComponents);
```

Los sufijos son similares a los de la función `glVertex`. Utilizamos un sufijo de 3 o 4 para especificar el modo RGB o RGBA junto con el sufijo para el tipo de dato numérico y un sufijo opcional para los vectores. El sufijo para los tipos de datos numérico son `b` (byte), `i` (entero), `s` (entero corto), `f` (decimal en punto flotante) y `d` (decimal en punto flotante de doble precisión), así como valores numéricos sin signo. Los valores en punto flotante de las componentes del color están comprendidos dentro del rango que va desde 0.0 hasta 1.0, y las componentes del color predeterminado de  `glColor`, incluyendo el valor alfa, son (1.0, 1.0, 1.0, 1.0), que establecen el color RGB en blanco y el valor alfa en 1.0. A modo de ejemplo, la siguiente línea utiliza valores en punto flotante en el modo RGB para establecer el color actual para las primitivas en cian (una combinación de las intensidades más altas de verde y azul).

```
	glColor3f ( 0.0, 1.0, 1.0 );
```

Empleando una especificación de matriz de las tres componentes de color, podríamos establecer el color del ejemplo anterior del siguiente modo:

```
	glColor3fv ( colorArray );
```

Una selección de color en OpenGL se puede asignar a posiciones de puntos individuales dentro de los pares `glBegin/glEnd`.

Las especificaciones utilizando valores enteros para las componentes de color dependen de las capacidades del sistema. En un sistema de color total, se asignan 8 bits por píxel (256 niveles para cada componente de color), los valores de color enteros varían desde 0 a 255. Los valores correspondientes en punto flotante de las componentes de color serían entonces  $0.0, 1.0/255.0, 2.0/255.0, \dots, 255.0/255.0 = 1.0$ . En un sistema de color total, podemos especificar el color cian del ejemplo anterior empleando valores enteros en las componentes de color de este modo:

```
glColor3i (0, 255, 255);
```

Las posiciones del búfer de imagen se almacenan realmente como valores enteros, por lo que la especificación de los valores de color con enteros evita las conversiones necesarias cuando se proporcionan valores en punto flotante. Cuando se especifica un valor de color en cualquier formato, éste se transforma a un valor entero dentro del rango de número de bits disponibles en un sistema concreto.

## Modo de color indexado de OpenGL

Las especificaciones de color en OpenGL también se pueden proporcionar en el **modo de color indexado**, el cual hace referencia a los valores de una tabla de color. Utilizando este modo, establecemos el color actual al especificar un índice dentro de una tabla de color:

```
glIndex* (colorIndex);
```

Al argumento `colorIndex` se le asigna un valor entero no negativo. Este valor del índice se almacena entonces en las posiciones del búfer de imagen para las primitivas especificadas posteriormente. Podemos especificar el índice de color con cualquiera de los tipos de datos siguientes: byte sin signo, entero o punto flotante. El tipo de dato del argumento `colorIndex` se indica con un sufijo `ub`, `s`, `i`, `d`, o `f`, y el número de posiciones del índice en una tabla de color siempre es una potencia de 2, tal como 256 o 1024. El número de bits disponible en cada posición de la tabla depende de las características del hardware del sistema. Como ejemplo de la especificación de color en el modo indexado, la siguiente línea establece el índice actual de color en el valor 196.

```
glIndexi (196);
```

A todas las primitivas definidas después de esta línea se les asignará el color almacenado en dicha posición de la tabla de color, hasta que el color actual se cambie.

La biblioteca de núcleo de OpenGL no proporciona funciones para cargar valores en la tabla de búsqueda de color, porque las subrutinas de procesamiento de la tabla forman parte del sistema de ventanas. Además, algunos sistemas de ventanas proporcionan múltiples tablas de color y color total, mientras que otros sistemas puede que sólo dispongan de una única tabla de color y elecciones de color limitadas. Sin embargo, disponemos de una subrutina de GLUT que interactúa con el sistema de ventanas para establecer especificaciones de color en la tabla en una posición del índice dada:

```
glutSetColor (índice, rojo, verde, azul);
```

A los argumentos de color `rojo`, `verde` y `azul` se les asignan valores en punto flotante dentro de un rango que varía desde 0,0 hasta 1,0. Este color se carga entonces en la tabla en la posición especificada por el valor del argumento `índice`.

Las subrutinas para el procesamiento de otras tres tablas de color se proporcionan como extensiones de la biblioteca de núcleo de OpenGL. Esta subrutinas forman parte del **subconjunto de procesamiento de imágenes** (Imaging Subset) de OpenGL. Los valores de color almacenados en estas tablas se pueden utilizar para modificar los valores de los píxeles cuando éstos se procesan mediante varios búferes. Algunos ejemplos de utilización de estas tablas son los siguientes: efectos de enfoque de la cámara, filtrado de ciertos colores de una imagen, mejora de ciertas intensidades o ajustes de brillo, conversión a escala de grises de un fotograma en color, y suavizado de una representación. También podemos utilizar estas tablas para cambiar los modelos

de color; es decir, podemos cambiar los colores RGB a otra especificación utilizando otros tres colores «primarios» (tales como cian, magenta y amarillo).

Una tabla de color concreta del subconjunto de procesamiento de imágenes de OpenGL se activa con la función `glEnable`, empleando como argumento uno de los nombres de la tabla: `GL_COLOR_TABLE`, `GL_POST_CONVOLUTION_COLOR_TABLE` o `GL_POST_COLOR_MATRIX_COLOR_TABLE`. Entonces podemos utilizar las subrutinas del subconjunto de procesamiento de imágenes para seleccionar una tabla de color concreta, establecer valores de la tabla de color, copiar valores de la tabla o especificar las componentes de color de un píxel que queremos cambiar y cómo queremos cambiarlo.

## Fundido de color en OpenGL

En muchas aplicaciones, es conveniente ser capaz de combinar los colores de objetos superpuestos o fundir un objeto con el fondo. Algunos ejemplos son los siguientes: simulación del efecto de una brocha, creación de una imagen compuesta de dos o más imágenes, modelado de efectos de transparencia y suavizado de los objetos de una escena. La mayoría de los paquetes gráficos proporcionan métodos para producir varios efectos de mezcla de color, y estos procedimientos se denominan **funciones de fundido de color** o **funciones de composición de imágenes**. En OpenGL, los colores de dos objetos se pueden fundir cargando en primer lugar un objeto en el búfer de imagen, para después combinar el color del segundo objeto con el color del búfer de imagen. El color actual del búfer de imagen se denomina *color de destino* de OpenGL y el color del segundo objeto *color de origen* de OpenGL. Los métodos de fundido se pueden realizar sólo en el modo RGB o RGBA. Para aplicar el fundido de color en una aplicación, necesitamos en primer lugar activar esta característica de OpenGL con la siguiente función.

```
 glEnable ( GL_BLEND );
```

Y desactivamos las subrutinas de fundido de color en OpenGL con:

```
 glDisable ( GL_BLEND );
```

Si el fundido de color no está activado, el color de un objeto simplemente reemplaza el contenido del búfer de imagen en la posición del objeto.

Los colores se pueden fundir de modos diferentes, dependiendo de los efectos que queramos lograr, y generaremos diferentes efectos de color mediante la especificación de dos conjuntos de *factores de fundido*. Un conjunto de factores de fundido es para el objeto actual en el búfer de imagen («objeto destino»), y el otro conjunto de factores de fundido es para el objeto entrante («origen»). El nuevo color que resulta del fundido y que se carga entonces en el búfer de imagen se calcula como:

$$(S_r R_s + D_r R_d, S_g G_s + D_g G_d, S_b B_s + D_b B_d, S_a A_s + D_a A_d) \quad (4.1)$$

donde las componentes de color RGBA del origen son  $(R_s, G_s, B_s, A_s)$ , las componentes de color del destino son  $(R_d, G_d, B_d, A_d)$ , los factores de fundido del origen son  $(S_r, S_g, S_b, S_a)$  y los factores de fundido del destino son  $(D_r, D_g, D_b, D_a)$ . Los valores calculados de las componentes del color de fundido se transforman al rango que varía desde 0.0 a 1.0. Es decir, cualquier suma mayor que 1.0 se cambia al valor 1.0 y cualquier suma menor que 0.0 se cambia a 0.0.

Seleccionamos los valores de los factores de fundido con la función de OpenGL:

```
 glBlendFunc ( sFactor, dFactor );
```

A los argumentos `sFactor` y `dFactor`, factores del origen y del destino, se les asigna a cada uno una constante simbólica de OpenGL que especifica un conjunto predefinido de cuatro coeficientes de fundido. Por ejemplo, la constante `GL_ZERO` proporciona los factores de fundido (0.0, 0.0, 0.0, 0.0) y `GL_ONE` (1.0, 1.0, 1.0, 1.0). Podríamos cambiar los cuatro factores de fundido del valor alfa de destino y del valor alfa del origen mediante el empleo de `GL_DST_ALPHA` y `GL_SRC_ALPHA`, respectivamente. Otras constantes de OpenGL disponibles para cambiar los factores de fundido son las siguientes: `GL_ONE_MINUS_DST_ALPHA`,

`GL_ONE_MINUS_SRC_ALPHA`, `GL_DST_COLOR` y `GL_SRC_COLOR`. Estos factores de fundido se utilizan a menudo para la simulación de transparencias, y se estudian con un mayor grado de detalle en el Sección 10.19. El valor predeterminado del argumento `sFactor` es `GL_ONE`, mientras que el valor predeterminado del argumento `dFactor` es `GL_ZERO`. Por tanto, los valores predeterminados de los factores de fundido provocan que los valores de color entrantes sustituyan a los valores actuales que hay en el búfer de imagen.

Se han incluido funciones adicionales en la extensión de OpenGL denominada Imaging Subset. Entre estas funciones se incluye una subrutina para establecer el color de fundido y otra subrutina para especificar la ecuación de fundido.

## Matrices de color en OpenGL

También podemos especificar los valores de color de una escena en combinación con los valores de las coordenadas en una matriz de vértices (Sección 3.17). Esto se puede hacer en modo RGB o en el modo de color indexado. Como en el caso de las matrices de vértices, debemos activar en primer lugar las características de la matriz de color de OpenGL:

```
glEnableClientState (GL_COLOR_ARRAY);
```

A continuación, en el caso del modo de color RGB, especificamos la localización y el formato de los componentes de color con:

```
glColorPointer (nColorComponents, dataType, offset, colorArray);
```

Al argumento `nColorComponents` se le asigna un valor de 3 o 4, dependiendo de si estamos enumerando las componentes de color RGB o RGBA en la matriz `colorArray`. Se asigna una constante simbólica de OpenGL tal como `GL_INT` o `GL_FLOAT` al argumento `dataType` para indicar el tipo de dato de los valores del color. En una matriz de color independiente, podemos asignar el valor 0 al argumento `offset`. Pero si combinamos los datos de color con los datos de los vértices en la misma matriz, el valor `offset` es el número de bytes entre cada conjunto de componentes de color en la matriz.

Como ejemplo de utilización de las matrices de color, podemos modificar el ejemplo de la matriz de vértices del Sección 3.17 para incluir una matriz de color. El siguiente fragmento de código establece el color de todos los vértices de la cara frontal del cubo en azul y a todos los vértices de la cara trasera se les asigna el color rojo.

```
typedef GLint vertex3 [3], color3 [3];

vertex3 pt [8] = { {0, 0, 0}, {0, 1, 0}, {1, 0, 0},
                  {1, 1, 0}, {0, 0, 1}, {0, 1, 1}, {1, 0, 1}, {1, 1, 1} };
color3 hue [8] = { {1, 0, 0}, {1, 0, 0}, {0, 0, 1},
                  {0, 0, 1}, {1, 0, 0}, {1, 0, 0}, {0, 0, 1}, {0, 0, 1} };

glEnableClientState (GL_VERTEX_ARRAY);
glEnableClientState (GL_COLOR_ARRAY);

glVertexPointer (3, GL_INT, 0, pt);
glColorPointer (3, GL_INT, 0, hue);
```

Incluso podemos empaquetar tanto los colores como las coordenadas de los vértices en una **matriz entrelazada**. Cada uno de los punteros harán referencia entonces a la única matriz entrelazada, con un valor apropiado de `offset`. Por ejemplo,

```
static GLint hueAndPt [ ] =
{1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0,
 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0,
```

```

1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1,
0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1};

glVertexPointer (3, GL_INT, 6*sizeof(GLint), hueAndPt [3]);
glColorPointer (3, GL_INT, 6*sizeof(GLint), hueAndPt [0]);

```

Los tres primeros elementos de esta matriz especifican un valor de color RGB, los tres siguientes elementos especifican un conjunto de coordenadas de vértice ( $x, y, z$ ), y este patrón continúa hasta la última especificación color-vértice. Establecemos el argumento `offset` con el número de bytes entre valores sucesivos de color o vértice, el cual es `6*sizeof(GLint)` para ambos. Los valores de color comienzan en el primer elemento de la matriz entrelazada, el cual es `hueAndPt [0]`, y los valores de los vértices comienzan en el cuarto elemento, que es `hueAndPt [3]`.

Ya que una escena contiene generalmente varios objetos y cada uno con múltiples superficies planas, OpenGL proporciona una función en la que podemos especificar todas las matrices de vértices y de color de una vez, así como otros tipos de información. Si cambiamos los valores de los colores y de los vértices en el ejemplo anterior a punto flotante, utilizamos esta función en la forma:

```
glInterleavedArrays (GL_C3F_V3F, 0, hueAndPt);
```

El primer argumento es una constante de OpenGL que indica especificaciones en punto flotante de tres elementos tanto para el color (C) como para las coordenadas de los vértices (V). Los tres elementos de la matriz `hueAndPt` están entrelazados con el color de cada vértice enumerado antes de las coordenadas. Esta función también habilita automáticamente ambas matrices de vértices y de color.

En el modo de color indexado, definimos una matriz de índices de color con:

```
glIndexPointer ( type, stride, colorIndex);
```

Los índices de color se enumeran en la matriz `colorIndex` y los argumentos `type` y `stride` son los mismos de `glColorPointer`. No se necesita el argumento `size`, ya que los índices de la tabla de color se especifican con un único valor.

## Otras funciones de color en OpenGL

En el primer ejemplo de programación de la Sección 2.9, presentamos la siguiente función que selecciona las componentes de color RGB para una ventana de visualización.

```
glClearColor ( rojo, verde, azul, alfa);
```

A cada componente de color en la designación (rojo, verde, y azul), así como el parámetro `alfa`, se le asigna un valor en punto flotante dentro del rango que varía desde 0.0 hasta 1.0. El valor predeterminado de los cuatro parámetros es 0.0, que produce el color negro. Si cada componente de color se establece en 1.0, el color de borrado es blanco. Los niveles de gris se obtienen con valores idénticos en las componentes de color entre 0.0 y 1.0. El cuarto argumento, `alpha`, proporciona una opción para el fundido del color anterior con el color actual. Esto puede ocurrir sólo si activamos la característica de fundido de OpenGL; el fundido de color se puede realizar cuando se especifican los valores en una tabla de color.

Como indicamos en el Sección 3.19, existen varios *búferes de color* en OpenGL, que se pueden utilizar como el búfer de refresco actual para visualizar una escena, y la función `glClearColor` especifica el color de todos los búferes de color. Entonces aplicamos el color de borrado a los búferes de color con el comando:

```
glClear ( GL_CLEAR_BUFFER_BIT);
```

También podemos utilizar la función `glClear` para establecer los valores iniciales de otros búferes disponibles en OpenGL. Existe el *búfer de acumulación*, el cual almacena la información de fundido de color, el *búfer de profundidad*, que almacena los valores de profundidad (distancias desde la posición de visualización) de los objetos de una escena y el *búfer de patrones*, que almacena información para definir los límites de una imagen.

En el modo de color indexado, utilizamos la siguiente función (en lugar de `glClearColor`) para definir el color de la ventana de visualización.

```
glClearIndex (index);
```

Al color del fondo de las ventanas se le asigna entonces el color almacenado en la posición `index` de la tabla de color. Y la ventana se muestra en ese color cuando se invoca la función `glClear (GL_CLEAR_BUFFER_BIT)`.

Hay disponibles muchas otras funciones de color en la biblioteca OpenGL para tratar una gran variedad de tareas, tales como el cambio de los modelos de color, el establecimiento de efectos de iluminación en las escenas, la especificación de efectos de cámara, y la representación de las superficies de un objeto. Examinaremos otras funciones de color cuando se exploren cada uno de los procesos componentes de un sistema de gráficos por computadora. Por ahora, limitaremos este estudio a aquellas funciones relacionadas con la especificación de color de las primitivas gráficas.

## 4.4 ATRIBUTOS DE LOS PUNTOS

---

Básicamente, podemos establecer dos atributos para los puntos: el color y el tamaño. En un sistema de estados, el color y el tamaño mostrados de un punto están determinados por los valores actuales almacenados en la lista de atributos. Las componentes de color se establecen con valores RGB o con un índice de una tabla de color. En un sistema de gráficos digitalizados, el tamaño del punto es un múltiplo del tamaño del píxel, por lo que un punto grande se muestra como un bloque cuadrado de píxeles.

## 4.5 ATRIBUTOS DE LAS LÍNEAS

---

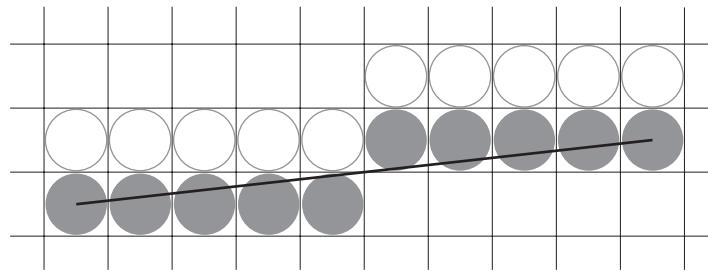
Un segmento de línea recta se puede mostrar con tres atributos básicos: el color, el grosor y el estilo. El color de la línea se establece habitualmente con la misma función de todas las primitivas gráficas, mientras que el grosor y el estilo de la línea se seleccionan con funciones independientes para las líneas. Adicionalmente, las líneas se pueden generar con otros efectos, tales como los trazos de un lapicero y una brocha.

### El grosor de las líneas

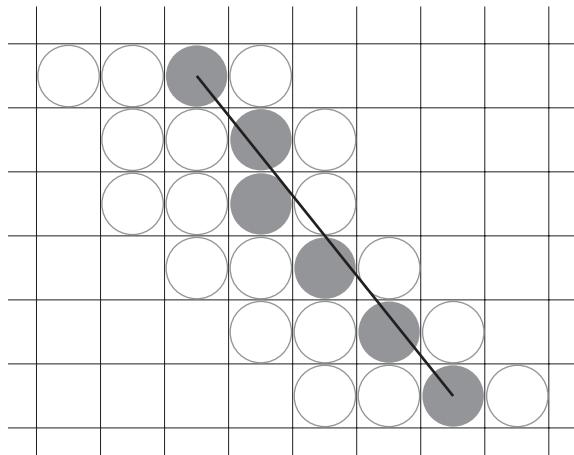
La implementación de las opciones del grosor de las líneas depende de las capacidades del dispositivo de salida. Una línea gruesa se podría mostrar en un monitor de vídeo como líneas adyacentes paralelas, mientras que un trazador de plumillas podría precisar el cambio de la plumilla para dibujar una línea gruesa.

En implementaciones de gráficos digitalizados, una línea de grosor estándar se genera con un único píxel en cada posición muestreada, como en el algoritmo de Bresenham. Las líneas más gruesas se visualizan como múltiplos enteros positivos de la línea estándar mediante el dibujo de píxeles adicionales a lo largo de líneas paralelas adyacentes. Si una línea tiene una pendiente menor o igual que 1.0, podemos modificar la subrutina de dibujo de líneas para mostrar líneas gruesas mediante el dibujo de píxeles de extensión en cada columna (posición  $x$ ) a lo largo de la línea. El número de píxeles que se van a mostrar en cada columna se define como el valor entero del grosor de la línea. En la Figura 4.2 se muestra una línea de grosor doble mediante la generación de una línea paralela encima de la trayectoria de la línea original. En cada posición  $x$  muestreada, calculamos la coordenada correspondiente  $y$  y se dibujan píxeles en las coordenadas de pantalla  $(x, y)$  y  $(x, y + 1)$ . Podríamos mostrar líneas con un grosor igual a 3 o mayor, dibujando alternativamente píxeles por encima y por debajo de la línea de grosor simple.

En una línea de pendiente mayor que 1.0, podemos mostrar líneas gruesas empleando extensiones horizontales, añadiendo píxeles alternativamente a la derecha y a la izquierda de la trayectoria de la línea. Este



**FIGURA 4.2.** Una línea digitalizada de doble grosor con pendiente  $|m| < 1.0$  generada con extensiones verticales de píxeles.



**FIGURA 4.3.** Una línea digitalizada con pendiente  $|m| > 1.0$  y un grosor de línea de 4 dibujada empleando extensiones horizontales de píxeles.

planteamiento se muestra en la Figura 4.3, donde un segmento de línea con un grosor de 4 se dibuja empleando múltiples píxeles transversales a cada línea de barrido. De forma similar, una línea gruesa con una pendiente menor o igual que 1.0 se puede mostrar empleando extensiones verticales de píxeles. Podemos implementar este procedimiento mediante la comparación de las magnitudes de las separaciones horizontal y vertical ( $\Delta x$  y  $\Delta y$ ) de los puntos extremos de la línea. Si  $|\Delta x| \geq |\Delta y|$ , los píxeles se repiten a lo largo de las columnas. En caso contrario, múltiples píxeles se dibujan de forma transversal a las filas.

Aunque las líneas gruesas se generan fácilmente mediante el dibujo de extensiones de píxeles horizontales o verticales, el grosor visualizado de una línea (medido perpendicularmente a la trayectoria de la línea) depende de su pendiente. Una línea de  $45^\circ$  se visualizará más fina debido al factor  $1/\sqrt{2}$  comparada con una línea horizontal o vertical dibujada con la misma longitud de extensiones de píxeles.

Otro problema con la implementación de las opciones de grosor utilizando extensiones de píxeles horizontales y verticales es que el método produce líneas cuyos extremos son horizontales o verticales independientemente de la pendiente de la línea. Este efecto es más apreciable con líneas muy gruesas. Podemos ajustar la forma de las terminaciones de la línea para darles una mejor apariencia añadiendo **extremos de línea** (Figura 4.4). Una tipo de extremo de línea es el *extremo abrupto*, que presenta terminaciones cuadradas que son perpendiculares a la trayectoria de la línea. Si la línea tiene una pendiente  $m$ , las terminaciones cuadradas de la línea gruesa tienen una pendiente  $-1/m$ . Cada una de las líneas paralelas componentes se visualiza entonces entre las dos líneas perpendiculares a cada terminación de la trayectoria de la línea. Otro tipo de extremo de línea es el *redondeado* que se obtiene añadiendo un semicírculo relleno a cada extremo abrupto. Los arcos cir-

culares tienen su centro en la zona media de la línea gruesa y un diámetro igual al grosor de la línea. Un tercer tipo de extremo de línea es el *cuadrado*. En este caso, simplemente extendemos la línea y añadimos extremos abruptos que se posicionan en la mitad del grosor de la línea más allá de las terminaciones.

Entre otros métodos para producir líneas gruesas se pueden incluir la visualización de la línea como un rectángulo relleno o la generación de la línea con un patrón de plumilla o brocha determinado, como se estudia en la sección siguiente. Para obtener una representación con forma de rectángulo para el límite de la línea, calculamos la posición de los vértices del rectángulo a lo largo de las perpendiculares a la trayectoria de la línea, para que las coordenadas de los vértices del rectángulo se desplacen de las posiciones originales de los extremos de la línea la mitad del grosor de la misma. La línea rectangular tiene entonces el aspecto de la Figura 4.4(a). Podríamos añadir extremos redondeados al rectángulo relleno, o podemos extender su longitud para mostrar extremos cuadrados.

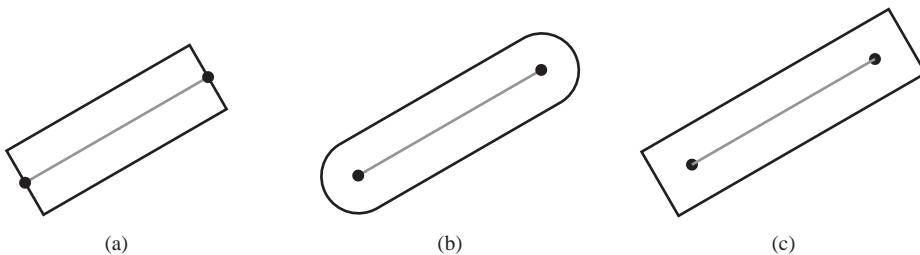
La generación de polilíneas gruesas requiere algunas consideraciones adicionales. Por lo general, los métodos que hemos considerado para mostrar segmentos de línea no producirán series de segmentos de línea conectados suavemente. La representación de polilíneas gruesas empleando extensiones de píxeles horizontales y verticales, por ejemplo, producen huecos en los píxeles en los límites entre los segmentos de línea con diferentes pendientes donde hay un cambio de extensiones de píxeles horizontales a extensiones verticales. Podemos generar polilíneas gruesas que estén unidas suavemente a costa de un procesamiento adicional en los extremos de los segmentos. La Figura 4.5 muestra tres posibles métodos para unir suavemente dos segmentos de línea. Una *unión en punta* se consigue extendiendo las fronteras de cada uno de los dos segmentos de línea hasta que se encuentran. Una *unión redondeada* se produce tapando la conexión entre los dos segmentos con una frontera circular cuyo diámetro es igual al grosor de la línea. Y una *unión biselada* se genera visualizando los segmentos de línea con extremos abruptos y llenando el hueco triangular donde los elementos se encuentran. Si el ángulo entre los dos segmentos de línea conectados es muy pequeño, una unión en punta puede generar una larga púa que distorsiona la apariencia de la polilínea. Un paquete gráfico puede evitar este efecto cambiando de unión en punta a unión biselada cuando, por ejemplo, el ángulo entre dos segmentos cualesquiera consecutivos es pequeño.

## Estilo de las líneas

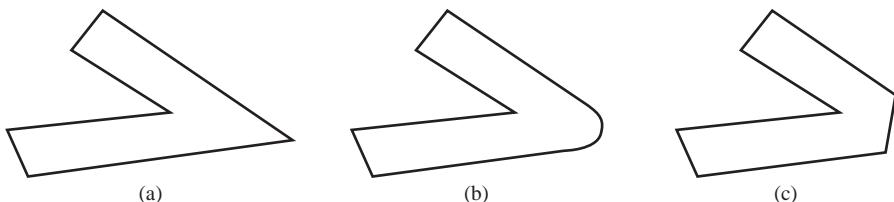
Entre las posibles elecciones del atributo de estilo de línea se incluyen las líneas continuas, las líneas discontinuas y las líneas punteadas. Modificamos el algoritmo de dibujo de líneas para generar tales líneas mediante el cambio de la longitud y del espaciado de las secciones continuas mostradas a lo largo de la trayectoria de la línea. En muchos paquetes gráficos, podemos seleccionar tanto la longitud de los trazos discontinuos como el espacio entre dichos trazos.

Los algoritmos de líneas en sistemas digitalizados muestran los atributos de estilo de las líneas dibujando extensiones de píxeles. En patrones discontinuos, de puntos o ambas a la vez, el procedimiento de dibujo de líneas produce como salida secciones continuas de píxeles a lo largo de la trayectoria de la línea, saltando un número de píxeles entre las extensiones continuas. La cantidad de píxeles utilizada en la longitud de las extensiones y el espacio entre las mismas se puede especificar mediante una **máscara de píxel**, la cual es un patrón de dígitos binarios que indica qué posiciones se han de dibujar a lo largo de la trayectoria de la línea. La máscara lineal 11111000, por ejemplo, se podría utilizar para mostrar una línea discontinua con una longitud de trazo de cinco píxeles y un espacio entre trazos de tres píxeles. A las posiciones de píxel correspondientes al bit 1 se les asigna el color actual, y a las posiciones de píxel correspondientes al bit 0 se les asigna el color de fondo.

El dibujo de trazos con un número fijo de píxeles provoca trazos de longitud desigual según las diferentes orientaciones de la línea, como se muestra en la Figura 4.6. Ambos trazos se dibujan con cuatro píxeles pero el trazo diagonal es más largo en un factor de  $\sqrt{2}$ . En dibujos en los que se necesita precisión, las longitudes de los trazos deberían permanecer aproximadamente constantes para cualquier orientación de la línea. Para conseguir esto, podríamos ajustar la cantidad de píxeles de las extensiones continuas y el espacio entre



**FIGURA 4.4.** Líneas gruesas dibujadas con extremos (a) abruptos, (b) redondeados y (c) cuadrados .



**FIGURA 4.5.** Segmentos de línea gruesa conectados con una unión en punta (a), redondeada (b) y biselada (c).

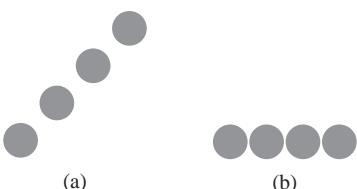
dichas extensiones de acuerdo con la pendiente de la línea. En la Figura 4.6, podemos visualizar trazos de longitud aproximadamente igual mediante la reducción del trazo diagonal a tres píxeles.

Otro método para mantener la longitud del trazo consiste en tratar los trazos como segmentos individuales de línea. Las coordenadas de los puntos extremos de cada trazo se localizan y se pasan a la subrutina de linea, la cual entonces calcula las posiciones de los píxeles a lo largo de la trayectoria del trazo.

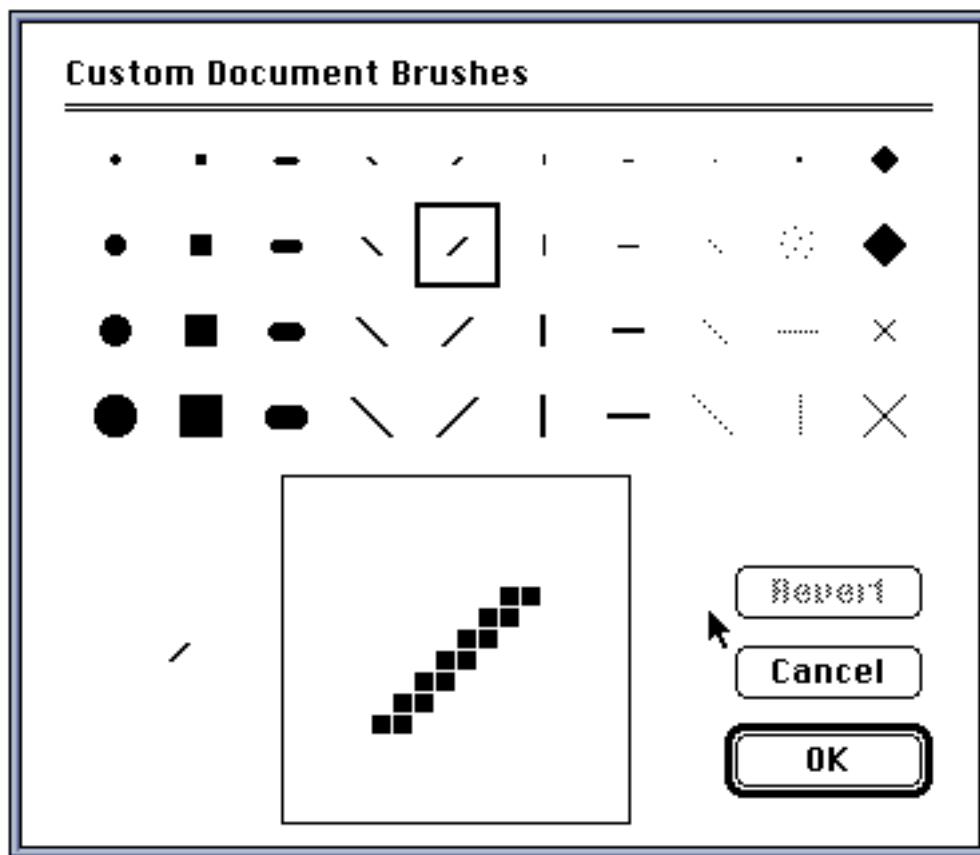
### Opciones de plumilla y brocha

En algunos paquetes, particularmente en los sistemas de dibujo y pintura, podemos seleccionar directamente diferentes estilos de plumilla y brocha. Entre las opciones de esta categoría se incluyen la forma, el tamaño y el patrón de la plumilla o brocha. Algunos ejemplos de formas de plumilla y brocha se muestran en la Figura 4.7. Esta forma se puede almacenar en una máscara de píxel que identifica la matriz de posiciones de píxel que se deben cambiar a lo largo de la trayectoria de la línea. Por ejemplo, una plumilla rectangular se podría implementar con la máscara que se muestra en la Figura 4.8 mediante el movimiento del centro (o una esquina) de la máscara a lo largo de la trayectoria de la línea, como en la Figura 4.9. Para evitar cambiar los píxeles más de una vez en el búfer de imagen, podemos simplemente acumular las extensiones horizontales generadas en cada posición de la máscara y mantener la pista del comienzo y el final de las posiciones  $x$  de las extensiones a lo largo de cada línea de barrido.

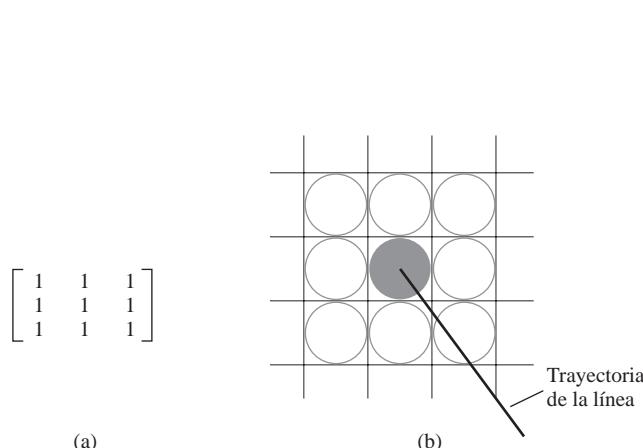
Las líneas generadas con plumilla (o brocha) se pueden visualizar con varios grosores mediante el cambio del tamaño de la máscara. Por ejemplo, la línea generada con una plumilla rectangular de la Figura 4.9 se podría estrechar con una máscara rectangular de tamaño 2 por 2 o ensanchar con una máscara de tamaño 4 por 4. También, las líneas se pueden representar con patrones creados mediante la superposición de varios patrones en una máscara de plumilla o brocha.



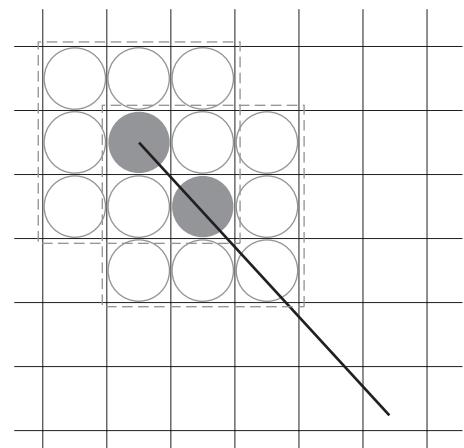
**FIGURA 4.6.** Trazos de distinta longitud visualizados con el mismo número de píxeles.



**FIGURA 4.7.** Formas de plumillas y brochas para la visualización de líneas.



**FIGURA 4.8.** Una máscara de píxel (a) de una plumilla rectangular, y la matriz asociada de píxeles (b) mostrada mediante el centrado de la máscara sobre una posición de píxel específica.



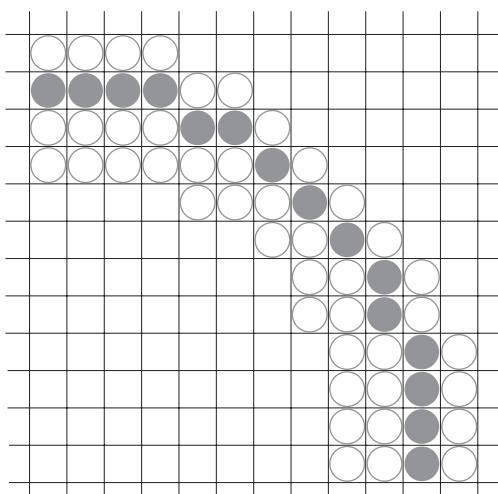
**FIGURA 4.9.** Generación de una línea con la forma de la plumilla de la Figura 4.8.

## 4.6 ATRIBUTOS DE LAS CURVAS

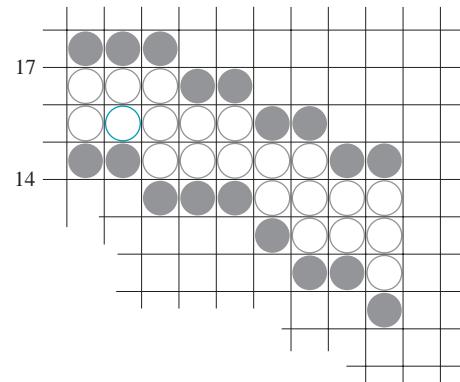
Los parámetros de los atributos de las curvas son los mismos que los de los segmentos de línea recta. Podemos mostrar curvas variando los colores, los grosores, utilizando patrones de punto y guión, y opciones disponibles de plumillas o brochas. Los métodos para la adaptación de algoritmos de dibujo de curvas para acomodarlos a las selecciones de los atributos son similares a aquellos empleados para el dibujo de líneas.

Las curvas digitalizadas de varios grosores se pueden representar empleando el método de las extensiones verticales y horizontales de píxeles. Donde la magnitud de la pendiente de la curva sea menor o igual que 1.0, dibujaremos extensiones verticales; donde la magnitud de la pendiente sea mayor que 1.0, dibujaremos extensiones horizontales. La Figura 4.10 muestra este método para la representación de un arco circular de grosor 4 en el primer cuadrante. Utilizando la simetría del círculo, podemos generar una trayectoria circular con extensiones verticales en el octante que va desde  $x = 0$  a  $x = y$ , y después reflejar las posiciones de los píxeles sobre la línea  $y=x$  para obtener el reflejo de la curva mostrada. Las secciones circulares de otros cuadrantes se obtienen mediante reflexión de las posiciones de los píxeles del primer cuadrante según los ejes de coordenadas. El grosor de las curvas representadas con este método es de nuevo función de la pendiente de la curva. Los círculos, elipses y otras curvas aparecerán más finas donde la pendiente tenga una magnitud de 1.

Otro método para representar curvas gruesas consiste en llenar el área entre dos trayectorias de curva paralelas, cuya distancia de separación sea igual al grosor deseado. Podríamos hacer esto empleando la trayectoria de la curva especificada como una frontera y establecer la segunda frontera dentro o fuera de la trayectoria de la curva original. Esta técnica, sin embargo, cambia la trayectoria de la curva original hacia dentro o hacia afuera, dependiendo de la dirección que elijamos para la segunda frontera. Podemos mantener la posición de la curva original estableciendo dos curvas frontera a una distancia de medio grosor a cada lado de la trayectoria de la curva especificada. Un ejemplo de esta técnica se muestra en la Figura 4.11 para un segmento circular con radio 16 y grosor 4. Los arcos límite se establecen entonces con una distancia de separación de 2 a cada lado del radio de valor 16. Para mantener las dimensiones adecuadas del arco circular, como se estudió en el Sección 3.13, podemos establecer el radio para los arcos límite concéntricos en  $r = 14$  y  $r = 17$ . Aunque este método es preciso para la generación de círculos gruesos, proporciona en general, sólo una aproximación al área verdadera de otras curvas gruesas. Por ejemplo, los límites internos y externos de una elipse gruesa generada con este método no tienen el mismo foco.



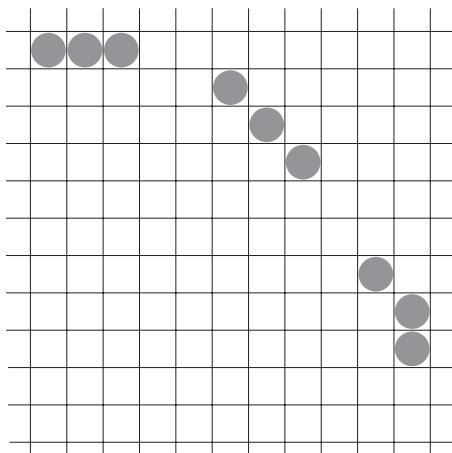
**FIGURA 4.10.** Un arco circular de grosor 4 dibujado con extensiones de píxeles verticales u horizontales, dependiendo de la pendiente.



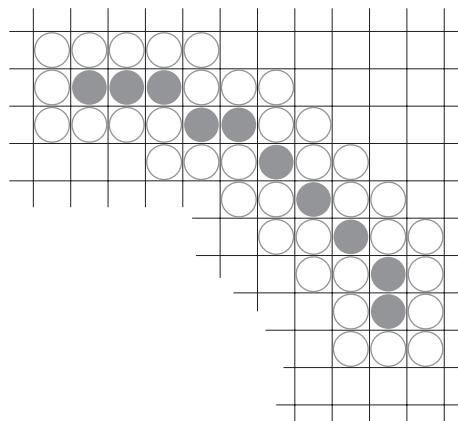
**FIGURA 4.11.** Un arco circular de grosor 4 y radio 16 representado llenando la región entre dos arcos concéntricos.

Las máscaras de píxeles estudiadas para la implementación de las opciones de los estilos de línea podrían también utilizarse en los algoritmos de curvas digitalizadas para generar patrones de línea discontinua o de puntos. Por ejemplo, la máscara 11100 produce el trazo de línea discontinua mostrado en la Figura 4.12. Podemos generar los trazos discontinuos en varios octantes utilizando la simetría del círculo, pero debemos cambiar las posiciones de los píxeles para mantener la secuencia correcta de trazos discontinuos y espacios a medida que cambiamos de un octante al siguiente. También, como en los algoritmos para las líneas rectas, las máscaras de píxeles muestran trazos discontinuos y espacios entre los mismos que varían en longitud de acuerdo a la pendiente de la curva. Si queremos representar trazos discontinuos de longitud constante, necesitamos ajustar el número de píxeles dibujados en cada trazo a medida que avanzamos sobre la circunferencia del círculo. En lugar de aplicar una máscara de píxeles con extensiones constantes, dibujamos píxeles a lo largo de arcos de igual ángulo para producir trazos discontinuos de igual longitud.

Las visualizaciones de curvas con una plumilla (o con una brocha) se generan utilizando las mismas técnicas estudiadas para los segmentos de línea recta. Reproducimos una forma de plumilla a lo largo de la trayectoria de la línea, como se muestra de la Figura 4.13 para el caso de un arco circular en el primer cuadrante. Aquí, el centro de la plumilla rectangular se mueve a sucesivas posiciones de la curva para producir la forma de curva mostrada. Las curvas obtenidas con una plumilla rectangular de este modo serán más gruesas donde la magnitud de la pendiente de la curva sea 1. Un grosor de curva uniforme se puede conseguir mediante la rotación de una plumilla circular para alinear ésta con la dirección de la pendiente, a medida que avanzamos sobre la curva o mediante la utilización de la forma de plumilla circular. Las curvas dibujadas con formas de plumillas o con formas de brochas se pueden mostrar con diferentes tamaños y con patrones superpuestos o brochazos simulados.



**FIGURA 4.12.** Un arco circular de trazo discontinuo obtenido con trazos discontinuos de 3 píxeles y un espacio entre trazos de 2 píxeles.



**FIGURA 4.13.** Un arco circular obtenido con una plumilla rectangular.



**FIGURA 4.14.** Líneas curvas dibujadas con un programa de pintura utilizando varias formas y patrones. De izquierda a derecha, las formas de la brocha son cuadrada, redonda, línea diagonal, patrón de puntos y aerógrafo con desvanecimiento.



**Figura 4.15.** Un muñeco daruma, un símbolo de buena fortuna en Japón, dibujado por el artista de las computadoras Koichi Kozaki empleando un sistema de pintura con brocha. Los muñecos daruma realmente carecen de ojos. Se pinta un ojo cuando se pide un deseo y el otro se pinta cuando el deseo se hace realidad. (Cortesía de Wacom Technology, Corp.)

Los programas de dibujo y pintura permiten la construcción de imágenes interactivamente mediante el uso de un dispositivo de apuntamiento, tal como un lapicero o una tableta gráfica, para esbozar varias formas de curva. En la Figura 4.14 se muestran algunos ejemplos de tales patrones de curva. La visualización de brochazos simulados es una opción de patrón adicional que puede proporcionarse en un paquete de pintura. La Figura 4.15 muestra algunos patrones que se pueden producir mediante modelado de diferentes tipos de brochas.

## 4.7 FUNCIONES OpenGL PARA LOS ATRIBUTOS DE LOS PUNTOS

---

El color mostrado en un punto se controla mediante los valores de color actuales de la lista de estado. Un color se especifica con la función `glColor` o `glIndex`.

Establecemos el tamaño de un punto en OpenGL con:

```
glPointSize (size);
```

y el punto se muestra entonces como un bloque de píxeles cuadrado. Al argumento `size` se le asigna un valor positivo en punto flotante, el cual se redondea a un entero (a menos que el punto se deba suavizar). El número de píxeles horizontales y verticales de la representación del punto se determina mediante el argumento `size`. Por tanto, un tamaño de punto de 1.0 muestra un único píxel y un tamaño de punto de 2.0 muestra una matriz de píxeles de tamaño 2 por 2. Si activamos las características de suavizado de OpenGL, el tamaño de un bloque de píxeles se modificará para suavizar los bordes. El valor predeterminado del tamaño de punto es 1.0.

Las funciones de atributo se pueden enumerar dentro o fuera de un par `glBegin`/ `glEnd`. Por ejemplo, el siguiente segmento de código dibuja tres puntos de distinto color y tamaño. El primero es un punto rojo de tamaño estándar, el segundo es un punto verde de doble tamaño y el tercero es un punto azul de tamaño triple.

```
glColor 3f  (1.0, 0.0, 0.0);
glBegin  (GL_POINTS);
    glVertex2i  (50, 100);
    glPointSize  (2.0);
    glColor3f  (0.0, 1.0, 0.0);
    glVertex2i  (75, 150);
    glPointSize  (3.0);
```

```

glColor3f (0.0, 0.0, 1.0);
glVertex2i (100, 200);
glEnd ();

```

## 4.8 FUNCIONES OpenGL PARA LOS ATRIBUTOS DE LAS LÍNEAS

---

Podemos controlar la apariencia de un segmento de línea recta en OpenGL con tres atributos: el color de la línea, el grosor de la línea y el estilo de la línea. Ya hemos visto cómo realizar una selección de color. OpenGL proporciona una función para establecer el grosor de una línea y otra para especificar un estilo de línea, tal como una línea discontinua o una línea de puntos.

### Función de grosor de línea de OpenGL

El grosor de línea se establece en OpenGL con la función:

```
glLineWidth (width);
```

Asignamos un valor en punto flotante al argumento `width`, y este valor se redondea al entero no negativo más cercano. Si el valor de entrada se redondea a 0.0, la línea se mostrará con un grosor estándar de 1.0, que es el grosor predeterminado. Sin embargo, cuando se aplica suavizado a una línea, los bordes se suavizan para reducir la apariencia de escalera debida a la digitalización. Los grosores fraccionarios son posibles. Algunas implementaciones de la función de grosor de línea admiten únicamente un número de grosores limitado y otras no admiten grosores distintos de 1.0.

La función de grosor de línea de OpenGL se implementa utilizando los métodos descritos en el Sección 4.5. Es decir, la magnitud de las separaciones horizontal y vertical de los puntos extremos de la línea,  $\Delta x$  y  $\Delta y$ , se comparan para determinar si se genera una línea gruesa utilizando extensiones de píxeles verticales o extensiones de píxeles horizontales.

### La función de estilo de línea de OpenGL

De manera predeterminada, un segmento de línea recta se visualiza como línea continua. Pero también podemos representar líneas discontinuas, líneas de puntos y una línea con una combinación de trazos y puntos. Podemos variar la longitud de los trazos y el espacio entre trazos o puntos. Establecemos el estilo de visualización actual de las líneas con la función de OpenGL:

```
glLineStipple (repeatFactor, pattern);
```

El argumento `pattern` se utiliza para referenciar un entero de 16 bits que describe cómo se visualizará la línea. Un bit de valor 1 en el patrón denota una posición de píxel «activado» y un bit de valor 0 indica una posición de píxel «desactivado». El patrón se aplica a los píxeles a lo largo de la trayectoria de la línea comenzando con los bit menos significativos del patrón. El patrón predeterminado es 0xFFFF (cada posición de píxel tiene un valor de 1), el cual produce una línea continua. El argumento entero `repeatFactor` especifica cuántas veces cada bit del patrón se repetirá antes de que se aplique el siguiente bit del patrón. De manera predeterminada, el valor de repetición es 1.

En una polilínea, un patrón de línea especificado no se comienza de nuevo al principio de cada segmento. Éste se aplica continuamente a lo largo de todos los segmentos, comenzando por el primer extremo de la polilínea y terminando en el extremo final del último segmento de la serie.

Como ejemplo de especificación del estilo de la línea, supóngase que se le asigna al argumento `pattern` la representación en hexadecimal 0x00FF y que el factor de repetición es 1. Esto mostrará una línea discontinua con ocho píxeles en cada trazo y ocho píxeles que están «activados» (un espacio de ocho píxeles) entre dos trazos. También, ya que los bits menos significativos se aplican en primer lugar, una línea comienza con

un trazo de ocho píxeles en su primer extremo. A este trazo le sigue un espacio de ocho píxeles, entonces otro trazo de ocho píxeles, y así sucesivamente hasta que se alcanza la posición del segundo extremo.

Antes de que se pueda visualizar una línea con el patrón de línea actual, debemos activar la característica de estilo de línea de OpenGL. Realizamos esto con la siguiente función.

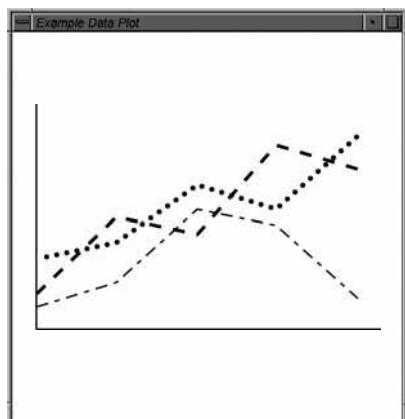
```
glEnable (GL_LINE_STIPPLE);
```

Si olvidamos incluir esta función de habilitación, las líneas se mostrarán como líneas continuas; es decir, se utiliza el patrón predeterminado 0xFFFF para mostrar los segmentos de línea. En cualquier momento, podemos desactivar la característica de patrón de línea con:

```
glDisable (GL_LINE_STIPPLE);
```

Esto reemplaza el patrón de estilo de línea con el patrón predeterminado (líneas continuas).

En el siguiente programa, mostramos el uso de las funciones OpenGL para los atributos de línea mediante el dibujo de tres gráficas de línea con diferentes estilos y grosores. La Figura 4.16 muestra los diagramas de datos que se podrían generar con este programa.



**FIGURA 4.16.** Dibujo de tres conjuntos de datos con tres diferentes estilos y grosores de línea en OpenGL: patrón trazo-punto con grosor sencillo, patrón discontinuo con doble grosor y patrón de puntos con grosor triple.

```
/* Define un tipo de datos para coordenadas universales bidimensionales. */
typedef struct { float x, y; } wcPt2D;

wcPt2D dataPts [5];

void linePlot (wcPt2D dataPts [5])
{
    int k;

    glBegin (GL_LINE_STRIP);
    for (k = 0; k < 5; k++)
        glVertex2f (dataPts [k].x, dataPts [k].y);

    glFlush ( );
    glEnd ( );
}
```

```

/* Invocar aquí un procedimiento para dibujar los ejes de coordenadas. */

glEnable (GL_LINE_STIPPLE);

/* Introduce el primer conjunto de valores (x, y). */
glLineStipple (1, 0x1C47);      // Dibuja una polilínea trazo-punto de grosor
                                // estándar.
linePlot (dataPts);

/* Introduce el segundo conjunto de valores (x, y). */
glLineStipple (1, 0x00FF);      // Dibuja una polilínea a trazos de grosor doble.
glLineWidth (2.0);
linePlot (dataPts);

/* Introduce el tercer conjunto de valores (x, y). */
glLineStipple (1, 0x0101);      // Dibuja una polilínea punteada de grosor triple.
glLineWidth (3.0);
linePlot (dataPts);

glDisable (GL_LINE_STIPPLE);

```

## Otros efectos de línea de OpenGL

Además de especificar el grosor, el estilo y un color sólido, podemos mostrar líneas con gradaciones de color. Por ejemplo, podemos variar el color a lo largo de la trayectoria de una línea continua mediante la asignación de un color diferente en cada extremo de la línea cuando definimos ésta. En el siguiente segmento de código mostramos esto asignando un color azul a un extremo de la línea y un color rojo al otro extremo. La línea continua se visualiza entonces como una interpolación lineal de los colores de los puntos extremos.

```

glShadeModel (GL_SMOOTH);

glBegin (GL_LINES);
    glColor3f (0.0, 0.0, 1.0);
    glVertex2i (50, 50);
    glColor3f (1.0, 0.0, 0.0);
    glVertex2i (250, 250);
glEnd ();

```

A la función `glShadeModel` también se le puede pasar el argumento `GL_FLAT`. En ese caso, el segmento de línea se muestra en un único color: el color del segundo extremo, (250, 250). Es decir, habríamos generado una línea roja. Realmente, `GL_SMOOTH` es el argumento predeterminado, por lo que generaríamos un segmento de línea de color suavemente interpolado incluso si no incluyéramos esta función en el código.

Podemos producir otros efectos mediante la representación de líneas adyacentes con diferentes patrones y colores. También podemos utilizar las características de OpenGL de fundido de color mediante la superposición de líneas u otros objetos con valores alfa que varían. Un trazo de brocha y otros efectos de pintura se pueden simular con un mapa de píxeles y fundido de color. El mapa de píxeles se puede entonces mover de forma interactiva para generar segmentos de línea. A los píxeles individuales del mapa de píxeles se les pueden asignar diferentes valores alfa para representar líneas como trazos de brocha o plumilla.

## 4.9 ATRIBUTOS DE RELLENO DE ÁREAS

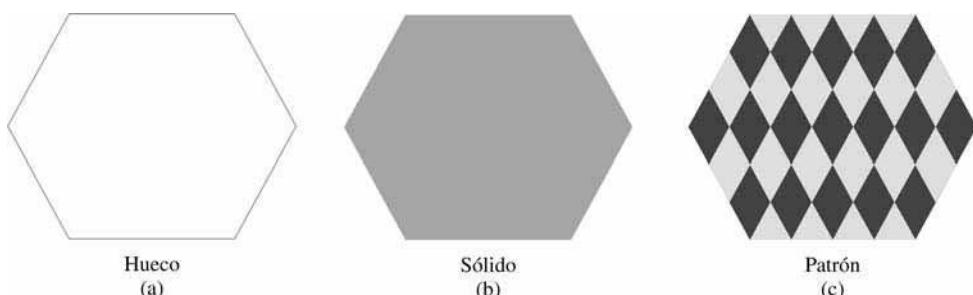
La mayoría de los paquetes gráficos limitan el relleno de áreas a polígonos, porque éstos se describen con ecuaciones lineales. Una mayor restricción requiere que las áreas a llenar sean polígonos convexos, para que las líneas de barrido no intersecten con más de dos lados de los polígonos. Sin embargo, por lo general, podemos llenar regiones cualesquiera, entre las que se incluyen los círculos, elipses y otros objetos con límites curvos. Los sistemas de aplicaciones, tales como los programas de pintura, proporcionan opciones de relleno de regiones de forma arbitraria.

Existen dos procedimientos básicos para llenar un área en sistemas digitalizados, una vez que la definición de la región a llenar se ha mapeado a coordenadas de píxel. Un procedimiento determina en primer lugar los intervalos de superposición de las líneas de barrido que cruzan el área. Entonces, las posiciones de los píxeles a lo largo de estos intervalos de superposición se cambian al color de relleno. Otro método para llenar regiones consiste en comenzar por una posición interior y «pintar» hacia afuera, píxel a píxel, desde este punto hasta encontrar las condiciones límite especificadas. El procedimiento de la línea de barrido se aplica habitualmente a formas simples tales como círculos o a las regiones con límites definidos a base de polilíneas. Los paquetes gráficos generales utilizan este método de relleno. Los algoritmos de relleno que utilizan un punto interior de comienzo son útiles para llenar áreas con límites más complejos y se utilizan en aplicaciones de pintura interactivas.

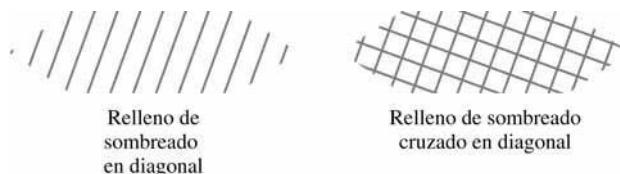
### Estilos de relleno

Un atributo básico de relleno proporcionado por una biblioteca gráfica general es el estilo de visualización del interior. Podemos visualizar la región con un único color, con un patrón de relleno específico, o con un estilo «hueco» mostrando únicamente el contorno de la región. Estos tres estilos de relleno se muestran en la Figura 4.17. También podemos llenar regiones seleccionadas de una escena empleando varios estilos de brochas, combinaciones de fundido de color, o texturas. Entre otras opciones se pueden incluir las especificaciones de visualización de los límites de un área de relleno. En el caso de los polígonos, podríamos mostrar los bordes con diferentes colores, grosor y estilos. Podemos seleccionar diferentes atributos de visualización para las caras anterior y posterior de una región.

Los patrones de relleno se pueden definir con matrices de color rectangular que enumeran diferentes colores para diferentes posiciones en la matriz. O, se podría especificar un patrón de relleno como una matriz de bits que indique qué posiciones relativas se han de mostrar con el color seleccionado. Una matriz que especifica un patrón de relleno es una *máscara* que se debe aplicar al área de visualización. Algunos sistemas gráficos proporcionan una posición inicial arbitraria para la superposición de la máscara. Desde esta posición de comienzo, la máscara se repite en dirección horizontal y vertical hasta que el área de visualización se ha llenado con copias no superpuestas del patrón. Donde el patrón se superpone con las áreas especificadas de relleno, la matriz con el patrón indica qué píxeles se deberían mostrar con un color concreto. Este proceso de llenado de un área con un patrón rectangular se denomina **disposición en mosaico** y un patrón de relleno



**FIGURA 4.17.** Estilos básicos de relleno de polígonos.



**FIGURA 4.18.** Áreas rellenas con patrones de entramado.

rectangular se denomina a veces **patrón en mosaico**. A veces, se dispone en el sistema de patrones de relleno predefinidos, tales como los patrones de relleno *con una trama* mostrados en la Figura 4.18.

Podemos implementar un patrón de relleno determinando donde el patrón se superpone con aquellas líneas de barrido que cruzan un área de relleno. Comenzando por una posición de partida específica del patrón de relleno, mapeamos los patrones rectangulares verticalmente a través de las líneas de barrido y horizontalmente a través de las posiciones de los píxeles de las líneas de barrido. Cada repetición de la matriz de patrón se realiza a intervalos determinados por la anchura y la altura de la máscara. Donde el patrón se superpone con el área de relleno, los colores de los píxeles se cambian de acuerdo a los valores almacenados en la máscara.

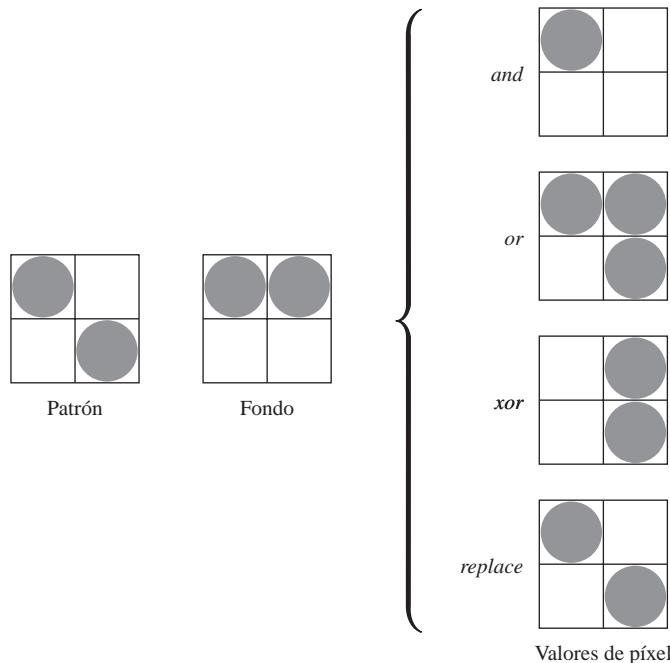
El relleno con tramas se podría aplicar a regiones mediante conjuntos de dibujo de segmentos de línea para mostrar un entramado único o entramado cruzado. El espacio y la pendiente de las líneas de la trama se podrían utilizar como parámetros en una tabla de trama. Alternativamente, el patrón del relleno con entramados se podría especificar como una matriz de patrón que produce conjuntos de líneas diagonales.

Un punto de referencia ( $x_p, y_p$ ) para la posición de comienzo de un patrón de relleno se puede establecer en cualquier posición conveniente, dentro o fuera de la región a llenar. Por ejemplo, el punto de referencia se podría establecer en un vértice del polígono. O el punto de referencia se podría elegir en la esquina inferior izquierda del rectángulo limitador (o caja delimitadora) determinado por las coordenadas límites de la región. Para simplificar la selección de las coordenadas de referencia, algunos paquetes siempre utilizan las coordenadas del origen de la ventana de visualización como posición de partida del patrón. Establecer siempre ( $x_p, y_p$ ) en el origen de coordenadas también simplifica las operaciones de disposición en mosaico cuando cada elemento de un patrón se debe mapear a un único píxel. Por ejemplo, si las filas de la matriz de patrón se referencian desde abajo hacia arriba, y comenzando con el valor 1, un valor de color se asigna entonces a la posición de píxel ( $x, y$ ) en coordenadas de pantalla desde la posición del patrón ( $y \bmod ny + 1, x \bmod nx + 1$ ). Donde,  $ny$  y  $nx$  especifican el número de filas y de columnas de la matriz del patrón. Sin embargo, estableciendo la posición de comienzo del patrón en el origen de coordenadas se asocia el patrón de relleno al fondo de la pantalla, en lugar de a la región de relleno. Las áreas de superposición o adyacentes llenadas con el mismo patrón podrían no mostrar frontera aparente. También, reposicionando y llenando un objeto con el mismo patrón se podría producir un cambio en los valores de los píxeles asignados sobre el interior del objeto. Un objeto en movimiento parecería que es transparente frente a un patrón de fondo estático, en lugar de moverse con un patrón interior fijo.

### Relleno de regiones con fundido de color

También se puede combinar un patrón de relleno con el color de fondo de varios modos. Un patrón se podría combinar con los colores de fondo utilizando un *factor de transparencia* que determine qué cantidad del fondo se debería mezclar con el color del objeto. O podríamos utilizar operaciones simples lógicas o de reemplazamiento. La Figura 4.19 muestra cómo las operaciones lógicas y de reemplazamiento combinarián un patrón de relleno 2 por 2 con un patrón de fondo en un sistema binario (blanco y negro).

Algunos métodos de relleno que utilizan fundido de color se han denominado algoritmos de **relleno suave** o **relleno tintado**. Un uso de estos métodos de relleno es suavizar los colores de relleno de los bordes del objeto que se han difuminado para suavizar los bordes. Otra aplicación de un algoritmo de relleno suave es permitir repintar un área de color que se llenó inicialmente con una brocha semitransparente, donde el color actual es entonces una mezcla del color de la brocha y los colores de fondo que hay «detrás del» área. En otro caso, queremos que el nuevo color de relleno tenga las mismas variaciones sobre el área que el color de relleno actual.



**FIGURA 4.19.** Combinación de un patrón de relleno con un patrón de fondo utilizando operaciones lógicas *and*, *or* y *xor* (*or exclusivo*), y usando reemplazamiento simple.

Como ejemplo de esta clase de relleno, el *algoritmo de relleno suave lineal* vuelve a pintar un área que originalmente se pintó mezclando un color  $\mathbf{F}$  de primer plano con un color simple de fondo  $\mathbf{B}$ , donde  $\mathbf{F} = \mathbf{B}$ . Si suponemos que conocemos los valores de  $\mathbf{F}$  y de  $\mathbf{B}$  podemos comprobar el contenido actual del búfer de imagen para determinar cómo estos colores se combinaron. El color actual RGB de cada píxel dentro del área se rellena con alguna combinación lineal de  $\mathbf{F}$  y  $\mathbf{B}$ :

$$\mathbf{P} = t\mathbf{F} + (1 - t)\mathbf{B} \quad (4.2)$$

donde el factor de transparencia  $t$  toma un valor entre 0 y 1 en cada píxel. Para valores de  $t$  menores que 0,5, el color de fondo contribuye más al color interior de la región que el color de relleno. La Ecuación vectorial 4.2 contiene las componentes RGB de los colores, con:

$$\mathbf{P} = (P_R, P_G, P_B), \quad \mathbf{F} = (F_R, F_G, F_B), \quad \mathbf{B} = (B_R, B_G, B_B) \quad (4.3)$$

Podemos por tanto calcular el valor del parámetro  $t$  utilizando una de las componentes de color RGB:

$$t = \frac{P_k - B_k}{F_k - B_k} \quad (4.4)$$

donde  $k = R, G$ , o  $B$ ; y  $F_k \neq B_k$ . Teóricamente, el argumento  $t$  tiene el mismo valor en cada componente RGB, pero los cálculos de redondeo para obtener códigos enteros pueden producir valores diferentes de  $t$  para componentes diferentes. Podemos minimizar este error de redondeo mediante la selección de la mayor diferencia entre  $\mathbf{F}$  y  $\mathbf{B}$ . Este valor de  $t$  se utiliza entonces para mezclar el nuevo color de relleno  $\mathbf{NF}$  con el color de fondo. Podemos realizar esta mezcla usando un procedimiento modificado de relleno por inundación o un procedimiento de relleno por límites, como se describe en el Sección 4.13.

Se pueden aplicar procedimientos similares de fundido de color a un área cuyo color de primer plano haya que fusionar con múltiples áreas de color de fondo, tales como un patrón de tablero de ajedrez. Cuando los colores de fondo  $B_1$  y  $B_2$  se mezclan con el color de primer plano  $\mathbf{F}$ , el color del pixel resultante  $\mathbf{P}$  es:

$$\mathbf{P} = t_0\mathbf{F} + t_1\mathbf{B}_1 + (1 - t_0 - t_1)\mathbf{B}_2 \quad (4.5)$$

donde la suma de los coeficientes de los términos de color  $t_0$ ,  $t_1$  y  $(1 - t_0 - t_1)$  debe ser igual a 1. Podemos establecer un sistema de dos ecuaciones utilizando dos de las tres componentes de color RGB para resolver los dos parámetros de proporcionalidad  $t_0$  y  $t_1$ . Estos parámetros se utilizan entonces para mezclar el nuevo color de relleno con los dos colores de fondo y obtener el nuevo color de píxel. Con tres colores de fondo y un color de primer plano, o con dos colores de fondo y dos colores de primer plano, necesitamos las tres ecuaciones RGB para obtener las cantidades relativas de los cuatro colores. Para algunas combinaciones de colores de primer plano y colores de fondo, sin embargo, el sistema de dos o tres ecuaciones RGB no se puede resolver. Esto ocurre cuando los valores de color son todos muy similares o cuando son todos proporcionales entre sí.

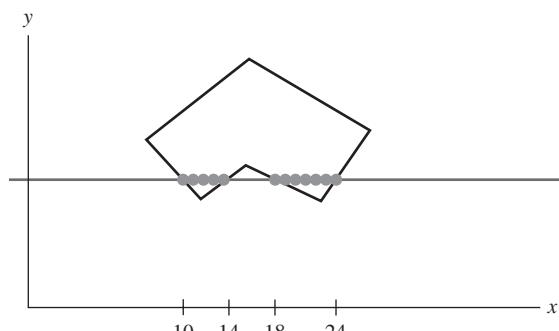
## 4.10 ALGORITMO GENERAL DE RELLENO DE POLÍGONOS MEDIANTE LÍNEAS DE BARRIDO

---

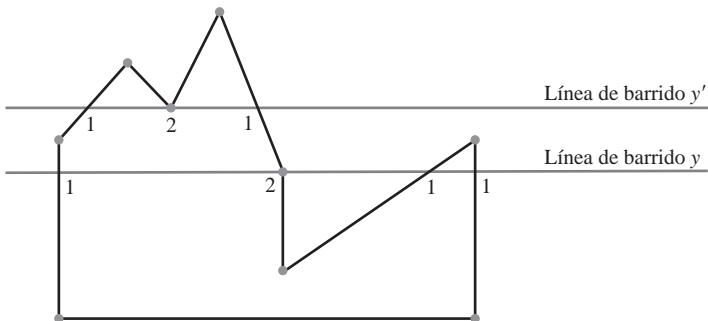
El relleno mediante líneas de barrido de una región se realiza determinando en primer lugar los puntos de intersección de los límites de la región que se va a llenar con las líneas de barrido de la pantalla. Entonces se aplican los colores de relleno a cada parte de la línea de barrido que se encuentra en el interior de la región a llenar. El algoritmo de relleno mediante líneas de barrido identifica las mismas regiones interiores que la regla par-impar (Sección 3.15). El área más simple para llenar es un polígono, ya que cada punto de intersección de la línea de barrido con el límite del polígono se obtiene mediante la resolución de un sistema de dos ecuaciones lineales, donde la ecuación de la línea de barrido es simplemente  $y = \text{constante}$ .

La Figura 4.20 muestra el procedimiento básico de líneas de barrido para el relleno con un color liso de un polígono. Para cada línea de barrido que atraviesa el polígono, las intersecciones con los bordes se ordenan de izquierda a derecha, y entonces las posiciones de los píxeles entre cada par de intersección se cambian al color de relleno especificado. En el ejemplo de la Figura 4.20, los cuatro píxeles de intersección con los límites del polígono definen dos tramos de píxeles interiores. Por tanto, el color de relleno se aplica a los cinco píxeles desde  $x = 10$  a  $x = 14$  y a los siete píxeles desde  $x = 18$  a  $x = 24$ . Si hay que aplicar un patrón de relleno al polígono, entonces el color de cada píxel a lo largo de la línea de barrido se determina a partir de su posición de solapamiento con el patrón de relleno.

Sin embargo, el algoritmo de relleno mediante líneas de barrido para un polígono no es tan simple como la Figura 4.20 podría sugerir. Cada vez que una línea de barrido pasa a través de un vértice, intersecta con dos aristas del polígono en dicho punto. En algunos casos, esto puede producir un número impar de intersecciones con la frontera para una línea de barrido. La Figura 4.21 muestra dos líneas de barrido que cruzan un área de relleno de un polígono y que intersectan con un vértice. La línea de barrido  $y'$  intersecta con un número par de aristas, y los dos pares de puntos de intersección a lo largo de esta línea de barrido identifican correctamente las extensiones interiores de píxeles. Pero la línea de barrido  $y$  intersecta con cinco aristas del polí-



**FIGURA 4.20.** Píxeles interiores a lo largo de una línea de barrido que pasa a través de un área de relleno de un polígono.



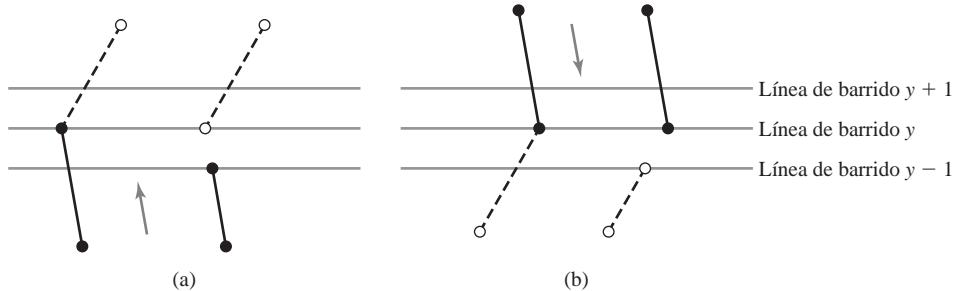
**FIGURA 4.21.** Puntos de intersección a lo largo de líneas de barrido que intersectan con los vértices de un polígono. La línea de barrido  $y$  genera un número impar de intersecciones, pero la línea de barrido  $y'$  genera un número par de intersecciones que se pueden emparejar para identificar correctamente las extensiones de píxeles interiores.

gono. Para identificar los píxeles interiores de la línea de barrido  $y$ , debemos contar la intersección con el vértice como un único punto. Por tanto, a medida que procesamos las líneas de barrido, necesitamos distinguir entre estos casos.

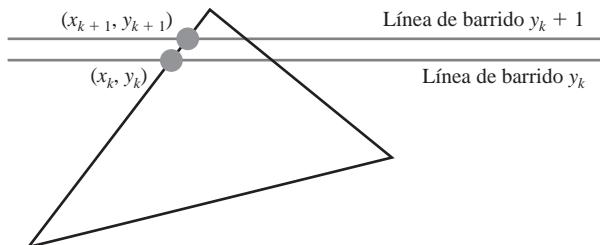
Podemos detectar la diferencia topológica entre la línea de barrido  $y$  y la línea de barrido  $y'$  mediante la anotación de la posición relativa de las aristas de intersección con la línea de barrido. En el caso de la línea de barrido  $y$ , las dos aristas que comparten un vértice de intersección están en lados opuestos a la línea de barrido. Pero en el caso de la línea  $y'$  las dos aristas de intersección están ambas sobre la línea de barrido. Por tanto, un vértice que tiene aristas contiguas en lados opuestos de una línea de barrido de intersección se debería contar como un único punto de intersección con la frontera. Podemos identificar estos vértices mediante el trazado de la frontera del polígono según el movimiento de las agujas del reloj o en sentido contrario y observando los cambios relativos en las coordenadas  $y$  del vértice, a medida que nos movemos de una arista a la siguiente. Si los tres valores de la coordenada  $y$  y de los tres puntos extremos de dos aristas consecutivas crecen o decrecen monótonamente, necesitamos contar el vértice compartido (medio) como un punto de intersección único para la línea de barrido que pasa a través del vértice. En cualquier otro caso, el vértice compartido representa un extremo local (mínimo o máximo) del límite del polígono, y las dos intersecciones con las aristas de la línea de barrido que pasa a través de aquel vértice se pueden añadir a la lista de intersecciones.

Un método para implementar el ajuste de la cuenta de intersecciones con los vértices consiste en acortar algunas aristas del polígono para dividir aquellos vértices que se deberían contar como una intersección. Podemos procesar las aristas no horizontales que hay a lo largo de los límites del polígono en el orden especificado, según las agujas del reloj o en sentido contrario. A medida que procesamos cada arista, podemos comprobar si dicha arista y la siguiente no horizontal tienen los valores de las coordenadas  $y$  y de sus puntos extremos monótonamente crecientes o decrecientes. Si esto es así, la arista inferior se puede acortar para asegurar que sólo se genera un punto de intersección para la línea de barrido que atraviesa el vértice común que une las dos aristas. La Figura 4.22 muestra el acortamiento de una arista. Cuando las coordenadas  $y$  y del punto extremo de las dos aristas crecen, el valor  $y$  del punto extremo superior del arista actual se decrementa en una unidad, como en la Figura 4.22(a). Cuando los valores  $y$  y del punto extremo decrecen monótonamente, como en la Figura 4.22(b), decrementamos la coordenada  $y$  y del punto extremo superior de la arista siguiente a la actual.

Habitualmente, ciertas propiedades de una parte de una escena están relacionadas de algún modo con las propiedades de otras partes de la escena. Estas **propiedades de coherencia** se pueden utilizar en los algoritmos de los gráficos por computadora para reducir el procesamiento. Los métodos de coherencia implican a menudo cálculos incrementales aplicados a lo largo de una única línea de barrido o entre dos líneas de barrido sucesivas. Por ejemplo, en la determinación de las intersecciones de una arista con el área a llenar, podemos establecer cálculos incrementales de coordenadas a lo largo de cualquier arista mediante la explotación del hecho de que la pendiente de una arista se mantiene constante de una línea de barrido a la siguiente. La



**FIGURA 4.22.** Ajuste de los valores de  $y$  del punto extremo para un polígono, a medida que procesamos las aristas en orden alrededor del perímetro del polígono. La arista que se está procesando actualmente se indica con una línea continua. En (a), la coordenada  $y$  del punto extremo superior de la arista actual se decrementa en una unidad. En (b), la coordenada  $y$  del punto extremo superior de la arista siguiente se decrementa en una unidad.



**FIGURA 4.23.** Dos líneas de barrido sucesivas que intersectan con los límites de un polígono.

Figura 4.23 muestra dos líneas de barrido sucesivas que cruzan la arista izquierda de un triángulo. La pendiente de esta arista se puede expresar en términos de las coordenadas de la intersección con la línea de barrido:

$$m = \frac{y_{k+1} - y_k}{x_{k+1} - x_k} \quad (4.6)$$

Ya que el cambio en las coordenadas  $y$  entre las dos líneas de barrido es simplemente,

$$y_{k+1} - y_k = 1 \quad (4.7)$$

el valor  $x_{k+1}$  de intersección según el eje  $x$  de la línea de barrido superior se puede determinar a partir del valor  $x_k$  de intersección según el eje  $x$  de la línea de barrido precedente del siguiente modo,

$$x_{k+1} = x_k + \frac{1}{m} \quad (4.8)$$

Cada sucesiva  $x$  de la intersección se puede por tanto calcular añadiendo el inverso de la pendiente y redondeando al entero más próximo.

Una implementación paralela obvia del algoritmo de relleno consiste en asignar cada línea de barrido que cruza el polígono a un procesador independiente. Los cálculos de las intersecciones con las aristas se realizan entonces de forma independiente. A lo largo de una arista con pendiente  $m$ , el valor de intersección  $x_k$  de la línea de barrido  $k$  sobre la línea inicial de barrido se puede calcular del siguiente modo:

$$x_k = x_0 + \frac{k}{m} \quad (4.9)$$

En un algoritmo secuencial de relleno, el incremento de los valores de  $x$  en la cantidad  $\frac{1}{m}$  a lo largo de una arista se puede realizar con operaciones enteras renombrando la pendiente  $m$  como el cociente de dos enteros:

$$m = \frac{\Delta y}{\Delta x}$$

donde  $\Delta x$  y  $\Delta y$  son las diferencias entre las coordenadas  $x$  e  $y$  de los puntos extremos de la arista. Por tanto, los cálculos incrementales de la  $x$  de intersección a lo largo de una arista con sucesivas líneas de barrido se pueden expresar del siguiente modo:

$$x_{k+1} = x_k + \frac{\Delta x}{\Delta y} \quad (4.10)$$

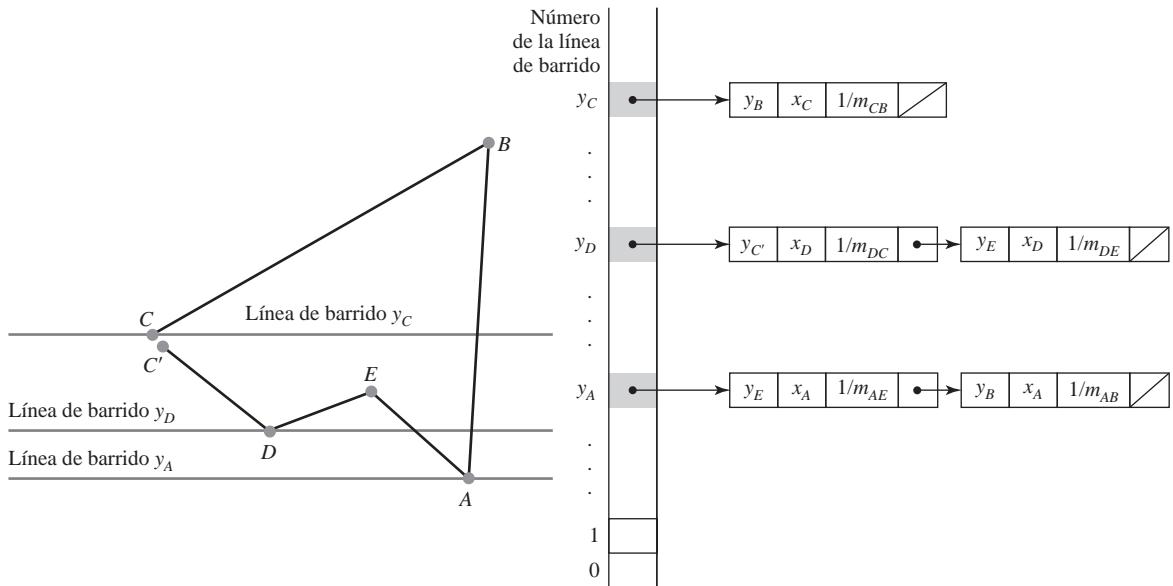
Empleando esta ecuación, podemos realizar la evaluación entera de la  $x$  de intersección inicializando un contador en 0 y después incrementando el contador con el valor  $\Delta x$  cada vez que nos desplazamos a una nueva línea de barrido. Cada vez que el valor del contador llega a ser mayor o igual que  $\Delta y$ , incrementamos el valor actual de la  $x$  de intersección en una unidad y decrementamos el contador con el valor  $\Delta y$ . Este procedimiento es equivalente a mantener la parte entera y la parte fraccionaria de la  $x$  de intersección e incrementar la parte fraccionaria hasta que alcanzamos el siguiente valor entero.

A modo de ejemplo de este esquema de incremento entero, supóngase que disponemos de una arista con una pendiente  $m = \frac{7}{3}$ . En la línea inicial de barrido, establecemos el contador en 0 y el incremento del contador en 3. A medida que nos desplazamos hacia arriba hacia las tres líneas de barrido siguientes a lo largo de esta arista, al contador se le asignan sucesivamente los valores 3, 6 y 9. En la tercera línea de barrido sobre la línea inicial de barrido, el contador toma un valor mayor que 7. Por lo que incrementamos la coordenada  $x$  de intersección en 1, y reiniciamos el contador al valor  $9 - 7 = 2$ . Continuamos determinando las intersecciones de las líneas de barrido de este modo hasta alcanzar el punto extremo superior de la arista. Se realizan cálculos similares para las intersecciones con las aristas de pendientes negativas.

Podemos redondear al entero más próximo el valor de la  $x$  de intersección del píxel, en lugar de truncar para obtener valores enteros, modificando el algoritmo de intersección con las aristas para que el incremento sea comparado con  $\Delta y/2$ . Esto se puede hacer con aritmética entera incrementando el contador con el valor  $2\Delta y$  en cada paso y comparando el incremento con  $\Delta y$ . Cuando el incremento es mayor o igual que  $\Delta y$ , incrementamos el valor de  $x$  en 1 y decrementamos el contador con el valor  $2\Delta y$ . En el ejemplo previo de  $m = \frac{7}{3}$ , los valores del contador para las primeras líneas de barrido sobre la línea inicial de barrido para esta arista serían ahora 6, 12 (reducido a -2), 4, 10 (reducido a -4), 2, 8 (reducido a -6), 0, 6, y 12 (reducido a -2). Ahora  $x$  se incrementaría en las líneas de barrido 2, 4, 6, 9, y así sucesivamente, sobre la línea inicial de barrido en esta arista. Los cálculos adicionales que se requieren para cada arista son  $2\Delta x = \Delta x + \Delta x$  y  $2\Delta y = \Delta y + \Delta y$ , que se realizan como pasos de preprocesamiento.

Para realizar de forma eficiente el relleno de un polígono, podemos en primer lugar almacenar el contorno del polígono en una *tabla de aristas ordenadas* que contenga toda la información necesaria para procesar las líneas de barrido eficientemente. Procediendo con las aristas en orden según el movimiento de las agujas del reloj o en orden contrario, podemos utilizar un búfer de ordenación para almacenar las aristas, ordenadas según el menor valor de  $y$  de cada arista, en las posiciones corregidas de las líneas de barrido. Sólo las aristas no horizontales se introducen en la tabla de aristas ordenadas. Cuando se procesan las aristas, también podemos acortar ciertas aristas para resolver la cuestión de la intersección con los vértices. Cada entrada en la tabla para una línea de barrido particular contiene el valor máximo de  $y$  para dicha arista, el valor de la  $x$  de intersección (en el vértice inferior) para la arista, y el inverso de la pendiente de la arista. Para cada línea de barrido, se ordenan las aristas de izquierda a derecha. La Figura 4.24 muestra un polígono y su tabla asociada de aristas ordenadas.

A continuación, procesamos las líneas de barrido desde la parte inferior del polígono hacia su parte superior, produciendo una *lista de aristas activas* para cada línea de barrido que cruza el contorno del polígono. La lista de aristas activas para una línea de barrido contiene todas las aristas con las que se cruza dicha línea



**FIGURA 4.24.** Un polígono y su tabla de aristas ordenadas, con la arista **DC** acortada una unidad en la dirección del eje *y*.

de barrido, con los cálculos de coherencia iterativos empleados para obtener las intersecciones con las aristas.

La implementación de los cálculos de las intersecciones con las aristas se puede facilitar almacenando los valores de  $\Delta x$  y  $\Delta y$  en la lista de aristas ordenadas. También, para asegurar que rellenamos correctamente el interior de los polígonos especificados, podemos aplicar las consideraciones estudiadas en la Sección 3.13. Para cada línea de barrido, rellenamos las extensiones de píxeles para cada par de  $x$  de intersección, comenzando por el valor de la  $x$  de intersección situada más a la izquierda y terminando en el valor anterior al valor de la  $x$  de intersección situada más a la derecha. Y cada arista del polígono se puede acortar en una unidad en la dirección del eje *y* en el punto extremo superior. Estas medidas también garantizan que los píxeles de polígonos adyacentes no se superpondrán.

## 4.11 RELLENO DE POLÍGONOS CONVEXOS MEDIANTE LÍNEAS DE BARRIDO

Cuando aplicamos un procedimiento de relleno mediante líneas de barrido a un polígono convexo, puede que no haya más que una única extensión interior para cada línea de barrido de la pantalla. Por lo que necesitaremos procesar las aristas del polígono sólo hasta que hayamos encontrado dos intersecciones con los límites para cada línea de barrido que cruza el interior del polígono.

El algoritmo general de línea de barrido de polígonos estudiado en la sección anterior se puede simplificar considerablemente en el caso del relleno de polígonos convexos. De nuevo utilizamos extensiones de coordenadas para determinar qué aristas cruzan una línea de barrido. Los cálculos de las intersecciones con estas aristas después determinan la extensión interior de píxeles para aquella línea de barrido, donde cualquier vértice cruzado se cuenta como un único punto de intersección con el límite. Cuando una línea de barrido intersecta en un único vértice (en una cúspide, por ejemplo), dibujamos sólo dicho punto. Algunos paquetes gráficos restringen más aún el relleno de áreas a únicamente triángulos. Esto hace que el relleno sea aún más sencillo, ya que cada triángulo sólo tiene tres aristas que procesar.

## 4.12 RELLENO DE REGIONES CON LÍMITES CURVOS MEDIANTE LÍNEAS DE BARRIDO

---

Puesto que un área con límites curvos se describe con ecuaciones no lineales, su relleno mediante líneas de barrido generalmente requiere más tiempo que el relleno de un polígono mediante líneas de barrido. Podemos utilizar la misma técnica general detallada en el Sección 4.10, pero los cálculos de las intersecciones con los límites se realizan con las ecuaciones de las curvas. La pendiente del límite cambia de forma continua, por lo que no podemos utilizar directamente los cálculos incrementales, que se emplean con las aristas rectas.

En el caso de curvas simples como círculos o elipses, podemos aplicar métodos de relleno similares a los empleados con los polígonos convexos. Cada línea de barrido que cruza el interior de un círculo o de una elipse tiene sólo dos intersecciones con su contorno. Podemos determinar estos dos puntos de intersección con el límite de un círculo o de una elipse utilizando cálculos incrementales del método del punto medio. Después, simplemente rellenamos la extensión de píxeles horizontal entre ambos puntos de intersección. Las simetrías entre cuadrantes (y entre octantes en los círculos) se utilizan para reducir los cálculos de los límites.

Métodos similares se pueden utilizar para generar el área de relleno de una sección curva. Por ejemplo, un área delimitada por un arco elíptico y una línea recta (Figura 4.25) se puede llenar utilizando una combinación de procedimientos de curvas y de rectas. Las simetrías y los cálculos incrementales se utilizan siempre que sea posible para reducir los cálculos.

El relleno de otras áreas curvadas puede implicar un procesamiento considerablemente mayor. Podríamos utilizar métodos incrementales similares en combinación con técnicas numéricas para determinar las intersecciones de las líneas de barrido, pero habitualmente los límites curvos se aproximan mediante segmentos de línea recta.

## 4.13 MÉTODOS DE RELLENO DE ÁREAS CON LÍMITES IRREGULARES

---

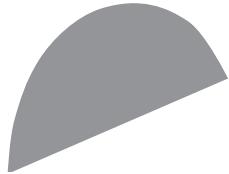
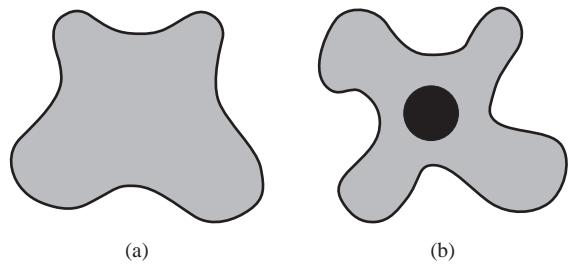
Otra técnica para llenar un área especificada consiste en comenzar por un punto interior y «pintar» el interior, punto a punto, hasta los límites. Esta técnica es particularmente útil para llenar áreas con límites irregulares, tales como un diseño creado con un programa de pintura. Generalmente, estos métodos requieren la introducción de un punto de comienzo dentro del área a llenar e información del color de los límites o del interior.

Podemos llenar regiones irregulares con un único color o con un patrón de color. En el caso del relleno con un patrón, superponemos una máscara de color, como se estudió en el Sección 4.9. Cuando se procesa cada píxel dentro de la región, se determina su color con los valores correspondientes del patrón de superpuesto.

### Algoritmo de relleno por contorno

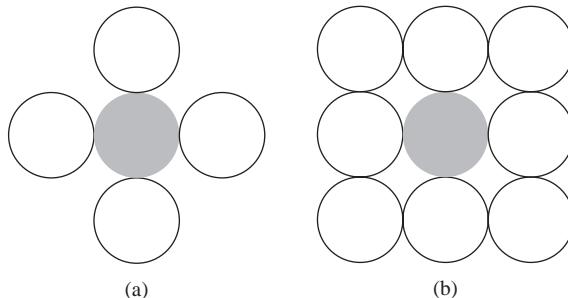
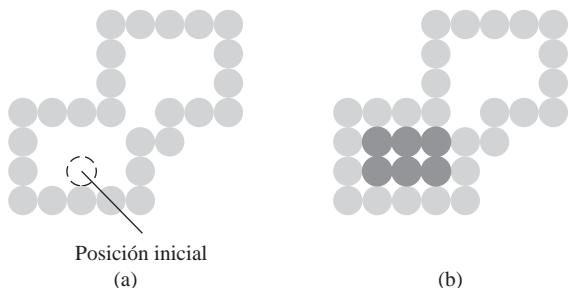
Si el límite de una región se especifica en un único color, podemos llenar el interior de esta región, píxel a píxel, hasta encontrar el color de dicho límite. Este método, denominado **algoritmo de relleno por contorno**, se emplea en paquetes de pintura interactivos, donde los puntos interiores se seleccionan fácilmente. Empleando una tableta gráfica u otro dispositivo interactivo, un artista o diseñador puede esbozar el contorno de una figura, seleccionar un color de relleno de un menú de color, especificar el color de los límites del área, y seleccionar un punto interior. El interior de la figura se pinta entonces con el color de relleno. Se pueden establecer tanto límites interiores como exteriores para definir un área para llenar por frontera. La Figura 4.26 muestra ejemplos de especificación de color de regiones.

Básicamente, un algoritmo de relleno por contorno comienza a partir de un punto interior ( $x, y$ ) y comprueba el color de los puntos vecinos. Si un punto comprobado no presenta el color de la frontera, se cambia su color al color de relleno y se comprueban sus vecinos. Este procedimiento continúa hasta que todos los píxeles se han procesado hasta comprobar el color del límite designado para el área.

**FIGURA 4.25.** Relleno del interior de un arco elíptico.**FIGURA 4.26.** Ejemplo de límites de color para el procedimiento de relleno por contorno (en negro en la figura).

La Figura 4.27 muestra dos métodos de procesamiento de los píxeles vecinos a partir de la posición actual que se está comprobando. En la Figura 4.27(a), se comprueban cuatro puntos vecinos. Estos son los píxeles que están a la derecha, a la izquierda, encima y debajo del píxel actual. Las áreas llenadas con este método se denominan **4-conectadas**. El segundo método, mostrado en la Figura 4.27(b), se utiliza para llenar figuras más complejas. Aquí el conjunto de posiciones vecinas que hay que comprobar incluye los píxeles de las cuatro diagonales, así como las de las direcciones cardinales. Los métodos de relleno que utilizan esta técnica se denominan **8-conectados**. Un algoritmo de relleno por contorno 8-conectado llenaría correctamente el interior del área definida en la Figura 4.28, pero un algoritmo de relleno por contorno 4-conectado llenaría sólo parte de dicha región.

El siguiente procedimiento muestra un método recursivo para pintar un área 4-conectada con un color liso, especificado en el argumento `fillColor`, hasta un color de contorno especificado con el argumento `borderColor`. Podemos ampliar este procedimiento para llenar una región 8-conectada mediante la inclusión de cuatro líneas adicionales para comprobar las posiciones diagonales ( $x \pm 1, y \pm 1$ ).

**FIGURA 4.27.** Métodos de relleno aplicados a un área 4-conectada (a) y a un área 8-conectada (b). Los círculos huecos representan píxeles que hay que comprobar a partir de la posición de comprobación actual, mostrada con un color sólido.**FIGURA 4.28.** El área definida dentro del contorno de color (a) se rellena sólo parcialmente en (b) utilizando un algoritmo de relleno por contorno 4-conectado.

```

void boundaryFill4 (int x, int y, int fillColor, int borderColor)
{
    int interiorColor;

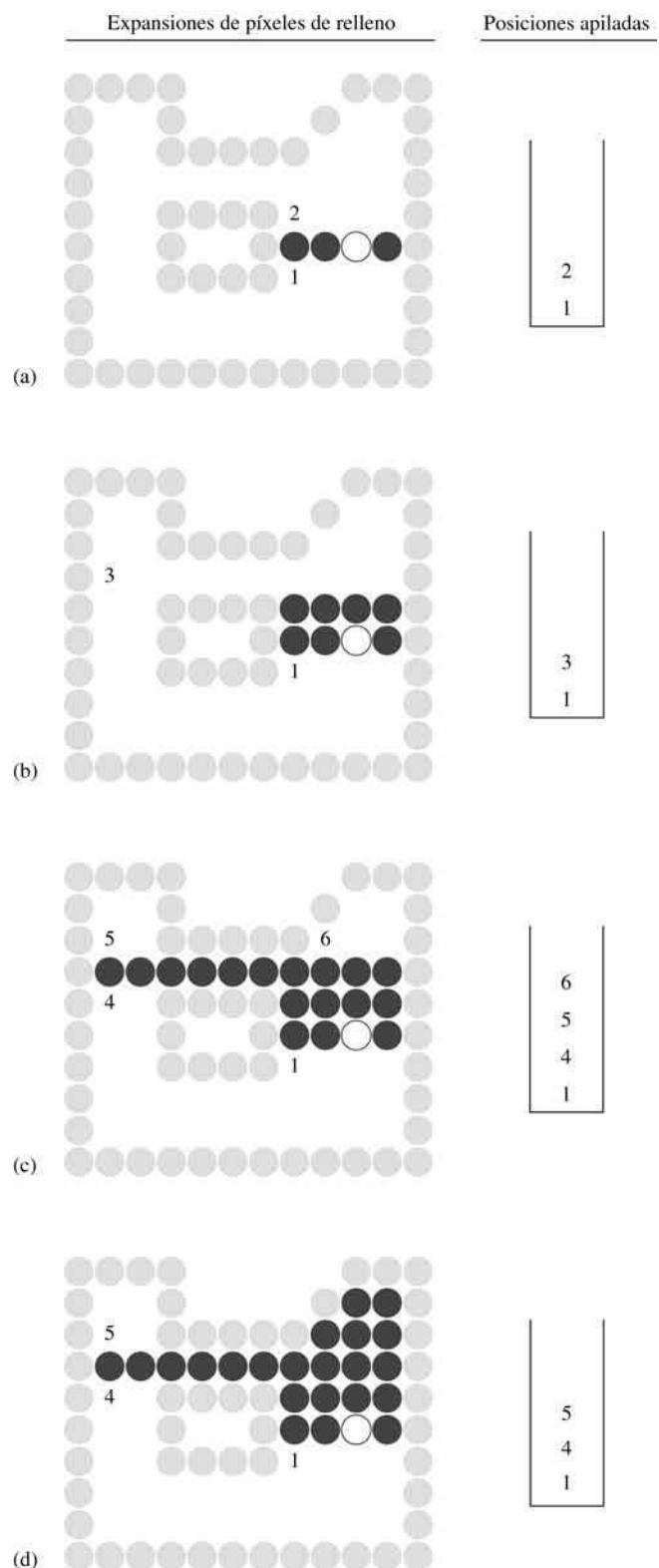
    /* Establece el color actual en fillColor, después realiza las siguientes
       operaciones. */
    getPixel (x, y, interiorColor);
    if ((interiorColor != borderColor) && (interiorColor != fillColor)) {
        setPixel (x, y);      // Set color of pixel to fillColor.
        boundaryFill4 (x + 1, y , fillColor, borderColor);
        boundaryFill4 (x - 1, y , fillColor, borderColor);
        boundaryFill4 (x , y + 1, fillColor, borderColor);
        boundaryFill4 (x , y - 1, fillColor, borderColor)
    }
}

```

Los algoritmos de relleno por contorno recursivos pueden no llenar regiones correctamente si algunos píxeles interiores ya se muestran con el color de relleno. Esto ocurre porque el algoritmo comprueba los píxeles próximos tanto para el color del contorno como para el color de relleno. El encontrar un píxel con el color de relleno puede provocar que una rama recursiva termine, dejando otros píxeles interiores sin llenar. Para evitar esto, podemos cambiar en primer lugar el color de cualesquiera píxeles interiores que estén inicialmente con el color de relleno antes de aplicar el procedimiento de relleno por contorno.

También, ya que este procedimiento requiere un considerable apilamiento de los puntos vecinos, generalmente se emplean métodos más eficientes. Estos métodos llenan extensiones horizontales de píxeles a través de líneas de barrido, en lugar de proceder con los puntos vecinos 4-conectados u 8-conectados. Entonces sólo necesitamos apilar una posición de comienzo para cada extensión horizontal de píxeles, en lugar de apilar todas las posiciones no procesadas alrededor de la posición actual. Comenzando por el punto interior inicial con este método, en primer lugar llenamos las extensiones contiguas de píxeles de esta línea de barrido de partida. Después localizamos y apilamos las posiciones de comienzo para las extensiones de las líneas de barrido adyacentes, donde las extensiones se definen como la cadena horizontal y contigua de posiciones limitada por píxeles que se muestran con el color de contorno. En cada paso subsiguiente, recuperamos la siguiente posición de comienzo de la parte superior de la pila y repetimos el proceso.

En la Figura 4.29 se muestra un ejemplo de cómo las extensiones de píxeles se podrían llenar utilizando esta técnica para llenar regiones 4-conectadas. En este ejemplo, procesamos en primer lugar las líneas de barrido sucesivamente desde la línea de comienzo hasta el límite superior. Después de que todas las líneas de barrido superiores se han procesado, llenamos las extensiones de píxeles de las restantes líneas de barrido en orden hasta el límite inferior. El píxel situado más a la izquierda en cada extensión horizontal se localiza y se apila, en orden, hacia la izquierda o hacia la derecha a través de las sucesivas líneas de barrido, como se muestra en la Figura 4.29. En la parte (a) de esta figura, se ha llenado la extensión inicial y se han apilado las posiciones de comienzo 1 y 2 de las extensiones de las siguientes líneas de barrido (por debajo y por encima). En la Figura 4.29(b), la posición 2 se ha obtenido de la pila y se ha procesado para producir la extensión llenada que se muestra, y el píxel de comienzo (posición 3) de la única extensión de la línea siguiente de barrido se ha apilado. Después de procesar la posición 3, en la Figura 4.29 se muestran las extensiones llenadas y las posiciones apiladas. La Figura 4.29(d) muestra los píxeles llenos después de procesar todas las extensiones de la parte superior derecha del área especificada. La posición 5 se procesa a continuación y se llenan las extensiones de la parte superior izquierda de la región; después se toma la posición 4 para continuar el procesamiento de las líneas de barrido inferiores.



**FIGURA 4.29.** Relleno por contorno a través de las extensiones de píxeles de un área 4-conectada: (a) línea inicial de barrido con una extensión de píxeles rellenada, que muestra la posición del punto inicial (hueco) y las posiciones apiladas de las extensiones de píxeles de las líneas de barrido adyacentes. (b) Extensión de píxeles rellenada en la primera línea de barrido sobre la línea inicial de barrido y el contenido actual de la pila. (c) Extensiones de píxeles rellenadas en las dos primeras líneas de barrido sobre línea inicial de barrido y el contenido actual de la pila. (d) Completadas las extensiones de píxeles de la parte superior derecha de la región definida y las restantes posiciones apiladas que se han de procesar.



FIGURA 4.30. Un área definida por límites de múltiples colores.

### Algoritmo de relleno por inundación

A veces se desea llenar (o cambiar de color) un área que no está definida por un único color de contorno. La Figura 4.30 muestra un área limitada por varias regiones de colores diferentes. Podemos pintar tales áreas reemplazando un color interior especificado en lugar de buscar un color de contorno concreto. Este procedimiento de relleno se denomina **algoritmo de relleno por inundación**. Comenzamos por un punto interior especificado ( $x, y$ ) y se cambian los valores de todos los píxeles que actualmente tienen un color interior dado al color de relleno deseado. Si el área que queremos pintar tiene más de un color interior, podemos en primer lugar cambiar el valor de los píxeles para que todos los puntos interiores tengan el mismo color. Utilizando una técnica 4-conectada u 8-conectada, después procedemos con las posiciones de los píxeles hasta que se han repintado todos los puntos interiores. El siguiente procedimiento por inundación rellena una región 4-conectada recursivamente, comenzando por la posición de entrada.

```
void floodFill4 (int x, int y, int fillColor, int interiorColor)
{
    int color;

    /* Establece el color actual en fillColor, después realiza las siguientes
       operaciones. */
    getColor (x, y, color);
    if (color = interiorColor) {
        setColor (x, y);      // Set color of pixel to fillColor.
        floodFill4 (x + 1, y, fillColor, interiorColor);
        floodFill4 (x - 1, y, fillColor, interiorColor);
        floodFill4 (x, y + 1, fillColor, interiorColor);
        floodFill4 (x, y - 1, fillColor, interiorColor)
    }
}
```

Podemos modificar el procedimiento anterior para reducir los requisitos de almacenamiento de la pila rellenando extensiones horizontales de píxeles, como se estudió en el algoritmo de relleno por contorno. En esta técnica, apilamos únicamente las posiciones de comienzo de aquellas extensiones de píxeles que tienen el valor `interiorColor`. Los pasos de este algoritmo modificado de relleno por inundación son similares a los mostrados en la Figura 4.29 para el relleno por contorno. Comenzando por la primera posición de cada extensión, los valores de los píxeles se reemplazan hasta que se encuentra un valor distinto de `interiorColor`.

## 4.14 FUNCIONES OpenGL PARA ATRIBUTOS DE RELLENO DE ÁREAS

En el paquete gráfico OpenGL, se dispone de subrutinas de relleno de áreas sólo para polígonos convexos. Generamos visualizaciones de polígonos convexos llenos siguiendo estos cuatro pasos:

- (1) Definir un patrón de relleno
- (2) Invocar la subrutina de relleno de polígonos
- (3) Activar la característica de OpenGL de relleno de polígonos
- (4) Describir los polígonos que se van a llenar

Un patrón de relleno de polígonos se visualiza hasta e incluyendo las aristas del polígono. Por tanto, no hay líneas límite alrededor de la región de relleno a menos que añadamos éstas específicamente a la representación.

Además de especificar un patrón de relleno para el interior de un polígono, existen otras muchas opciones. Una opción consiste en representar un polígono hueco, en el que no se aplica un color o patrón interior y sólo se generan las aristas. Un polígono hueco es equivalente a mostrar una primitiva de polilínea cerrada. Otra opción es mostrar los vértices del polígono, sin relleno interior ni aristas. También, se pueden designar atributos diferentes para las caras anterior y posterior de un área de relleno poligonal.

### Función de relleno con patrón de OpenGL

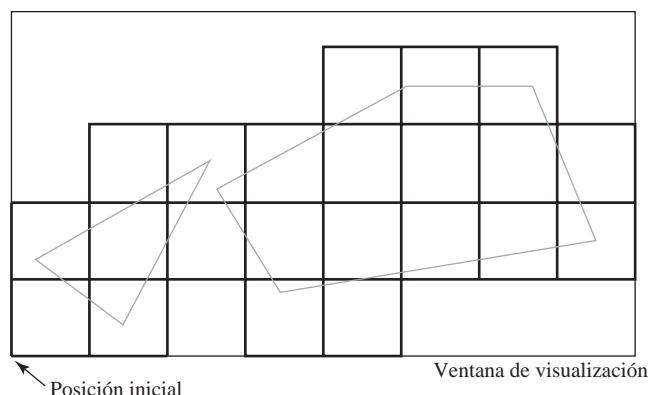
De forma predeterminada, un polígono convexo se muestra como una región de color liso, utilizando la configuración de color actual. Para llenar un polígono con un patrón en OpenGL, utilizamos una máscara de 32 por 32 bits. Un valor de 1 en la máscara indica que el píxel correspondiente se ha de cambiar al color actual, y un 0 deja el valor de dicha posición del búfer de imagen sin cambios. El patrón de relleno se especifica con bytes sin signo utilizando el tipo de dato de OpenGL `GLubyte`, como hicimos con la función `glBitmap`. Definimos un patrón de bits con valores hexadecimales como, por ejemplo,

```
Glubyte fillPattern [ ] = {
    0xff, 0x00, 0xff, 0x00, ... };
```

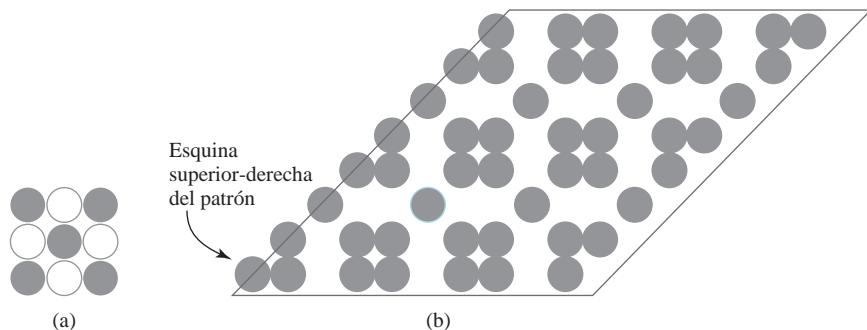
Los bits se deben especificar comenzando por la fila inferior del patrón, y continuando hasta la fila superior (32) del patrón, como hicimos con `bitShape` en el Sección 3.19. Este patrón se replica a través de toda el área de la ventana de visualización, comenzando por la esquina inferior izquierda de la ventana. Los polígonos especificados se llenan donde el patrón se superpone con dichos polígonos (Figura 4.31).

Una vez que hemos definido una máscara, podemos establecer ésta como el patrón de relleno actual con la función:

```
glPolygonStipple (fillPattern);
```



**FIGURA 4.31.** Disposición en mosaico de un patrón de relleno rectangular en una ventana de visualización para llenar dos polígonos convexos.



**FIGURA 4.32.** Un patrón de 3 por 3 bits (a) superpuesto a un paralelogramo para producir el área de relleno de (b), donde la esquina superior derecha del patrón coincide con la esquina inferior izquierda del paralelogramo.

A continuación, necesitamos habilitar las subrutinas de relleno antes de especificar los vértices de los polígonos que han de rellenarse con el patrón actual. Esto lo hacemos con la línea:

```
glEnable (GL_POLYGON_STIPPLE);
```

De forma similar, desactivamos el relleno con patrones con la instrucción:

```
glDisable (GL_POLYGON_STIPPLE);
```

La Figura 4.32 muestra cómo un patrón de 3 por 3 bits, repetida sobre una máscara de 32 por 32 bits, se podría aplicar para llenar un paralelogramo.

## Patrones de textura e interpolación de OpenGL

Otro método para llenar polígonos consiste en utilizar patrones de textura, como se estudia en el Capítulo 10. Este método puede producir patrones de relleno que simulan la apariencia superficial de la madera, ladrillos, acero cepillado o algún otro material. También, podemos obtener una coloración por interpolación del interior de un polígono como hicimos con la primitiva de líneas. Para ello, asignamos colores diferentes a los vértices del polígono. El relleno por interpolación del interior de un polígono se utiliza para producir visualizaciones realistas de superficies sombreadas con varias condiciones de iluminación.

Como ejemplo de relleno por interpolación, el siguiente fragmento de código asigna el color azul, rojo o verde a cada uno de los tres vértices de un triángulo. El relleno del polígono es una interpolación lineal de los colores de sus vértices.

```
glShadeModel (GL_SMOOTH);
glBegin (GL_TRIANGLES);
    glColor3f (0.0, 0.0, 1.0);
    glVertex2i (50, 50);
    glColor3f (1.0, 0.0, 0.0);
    glVertex2i (150, 50);
    glColor3f (0.0, 1.0, 0.0);
    glVertex2i (75, 150);
glEnd ( );
```

Por supuesto, si se establece un único color para todo el triángulo, el polígono se rellena con dicho color. Si cambiamos el argumento de la función `glShadeModel` a `GL_FLAT` en este ejemplo, el polígono se rellena con el último color especificado (verde). El valor `GL_SMOOTH` es el sombreado predeterminado, pero podemos incluir esta especificación para recordar que el polígono se ha de rellenar por interpolación de los colores de sus vértices.

## Métodos OpenGL para modelos alámbricos

También podemos elegir mostrar sólo las aristas de los polígonos. Esto produce una representación alámbrica o hueca del polígono. O podríamos mostrar un polígono dibujando únicamente un conjunto de puntos en las posiciones de sus vértices. Estas opciones se seleccionan con la función:

```
glPolygonMode (face, displayMode);
```

Utilizamos el argumento `face` para designar qué cara del polígono queremos mostrar sólo como aristas o sólo como vértices. A este argumento se le asigna `GL_FRONT`, `GL_BACK` o `GL_FRONT_AND_BACK`, para indicar la cara frontal, la cara trasera, o ambas, respectivamente. Después, si sólo queremos que se muestren las aristas del polígono, asignaremos la constante `GL_LINE` al argumento `displayMode`. Para dibujar sólo los puntos de los vértices del polígono, asignaremos la constante `GL_POINT` al argumento `displayMode`. Una tercera opción es `GL_FILL`. Pero este es el modo de visualización predeterminado, por lo que habitualmente sólo invitamos `glPolygonMode` cuando queremos establecer como atributos para los polígonos las aristas o los vértices.

Otra opción consiste en visualizar un polígono tanto con un relleno interior como un color o patrón para sus aristas (o para sus vértices) diferentes. Esto se realiza especificando el polígono dos veces: una con el argumento `displayMode` establecido en `GL_FILL` y luego de nuevo con `displayMode` definido como `GL_LINE` (o `GL_POINT`). Por ejemplo, el siguiente fragmento de código rellena el interior de un polígono con un color verde, y después se asigna a las aristas un color rojo.

```
glColor3f (0.0, 1.0, 0.0);
/* Invocar la subrutina de generación del polígono */

	glColor3f (1.0, 0.0, 0.0);
	glPolygonMode (GL_FRONT, GL_LINE);
/* Invocar la subrutina de generación del polígono de nuevo*/
```

Para un polígono tridimensional (aquel que no tiene todos los vértices en el plano *xy*), este método para mostrar las aristas de un polígono relleno puede producir huecos a lo largo de las aristas. Este efecto, a veces se denomina **costura** (*stitching*), es debido a las diferencias entre los cálculos en el algoritmo de relleno por línea de barrido y los cálculos en el algoritmo de dibujo de líneas en las aristas. Cuando se rellena el interior de un polígono tridimensional, el valor de la profundidad (distancia desde el plano *xy*) se calcula para cada posición (*x*, *y*). Pero este valor de profundidad en una arista del polígono no es a menudo exactamente el mismo que el valor de profundidad calculado por el algoritmo de dibujo de líneas para la misma posición (*x*, *y*). Por tanto, al hacer pruebas de visibilidad, el color de relleno interior se podría utilizar en lugar de un color de arista para mostrar algunos puntos a lo largo de los límites de un polígono.

Una manera de eliminar los huecos a lo largo de las aristas visualizadas de un polígono tridimensional, consiste en cambiar los valores de la profundidad calculados con la subrutina de relleno, para que no se superpongan con los valores de profundidad de las aristas para dicho polígono. Hacemos esto con las dos funciones siguientes de OpenGL.

```
 glEnable (GL_POLYGON_OFFSET_FILL);
 glPolygonOffset (factor1, factor2);
```

La primera función activa la subrutina de compensación para el relleno por línea de barrido y la segunda función se utiliza para establecer una pareja de valores en punto flotante, `factor1` y `factor2`, que se utilizan para calcular la cantidad de compensación de la profundidad. El cálculo de esta compensación de la profundidad es:

$$\text{depthOffset} = \text{factor1} \cdot \text{maxSlope} + \text{factor2} \cdot \text{const} \quad (4.11)$$

donde «`maxSlope`» es la máxima pendiente del polígono y «`const`» es una constante de implementación. Para un polígono en el plano *xy*, la pendiente es 0. En cualquier otro caso, la máxima pendiente se calcula como el

cambio en la profundidad del polígono dividido entre el cambio en  $x$  o en  $y$ . Un valor típico para los dos factores es 0.75 o 1.0, aunque se necesita alguna experimentación con los valores de los factores para conseguir buenos resultados. Como ejemplo de asignación de valores a los valores de compensación, podemos modificar el fragmento anterior de código como sigue:

```
glColor3f (0.0, 1.0, 0.0);
 glEnable (GL_POLYGON_OFFSET_FILL);
 glPolygonOffset (1.0, 1.0);
 /* Invocar la subrutina de generación del polígono */
 glDisable (GL_POLYGON_OFFSET_FILL);

 glColor3f (1.0, 0.0, 0.0);
 glPolygonMode (GL_FRONT, GL_LINE);
 /* Invocar la subrutina de generación del polígono de nuevo*/
```

Ahora el relleno interior del polígono queda un poco más alejado en profundidad, de modo que no interfiere con los valores de profundidad de sus aristas. También es posible implementar este método aplicando la compensación al algoritmo de dibujo de líneas, cambiando el argumento de la función  `glEnable` a `GL_POLYGON_OFFSET_LINE`. En este caso, queremos utilizar factores negativos para acercar los valores de profundidad de las aristas. Si sólo quisiéramos mostrar puntos de diferentes colores en las posiciones de los vértices, en lugar de realzar las aristas, el argumento de la función  `glEnable` debería ser `GL_POLYGON_OFFSET_POINT`.

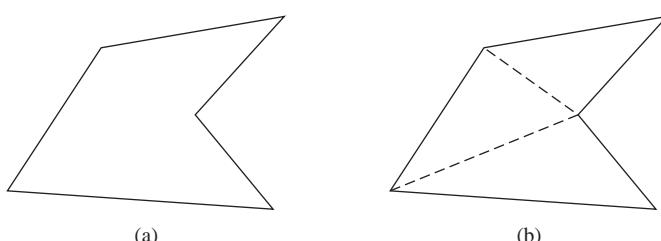
Otro método para eliminar el efecto de costura a lo largo de las aristas de un polígono consiste en utilizar el búfer de patrones de OpenGL, con el fin de limitar el relleno interior del polígono de modo que éste no se superponga con las aristas. Pero esta técnica es más complicada y generalmente más lenta, por lo que se prefiere el método de compensación de profundidad de polígonos.

Para visualizar un polígono cóncavo utilizando las subrutinas de OpenGL, debemos en primer lugar dividirlo en un conjunto de polígonos convexos. Habitualmente, dividimos un polígono en un conjunto de triángulos, utilizando los métodos descritos en el Sección 3.15. Después podríamos visualizar el polígono cóncavo como una región rellena, llenando los triángulos. De modo similar, si queremos mostrar únicamente los vértices del polígono, dibujamos los vértices de los triángulos. Pero, para visualizar el polígono cóncavo original en su forma alámbrica, no podemos cambiar sólo el modo de visualización a `GL_LINE`, porque esto mostraría todas las aristas de los triángulos que son interiores al polígono cóncavo original (Figura 4.33).

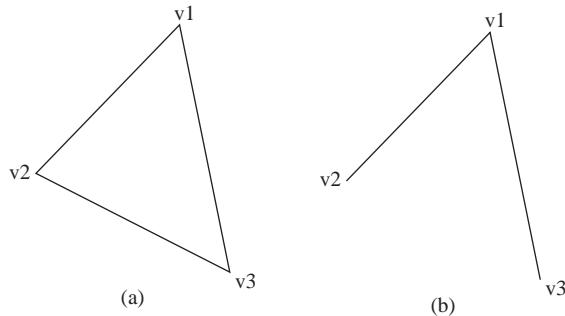
Afortunadamente, OpenGL proporciona un mecanismo que permite eliminar de una visualización en forma de alámbrica una serie de aristas seleccionadas. Cada vértice del polígono se almacena con una bandera (*flag*) de un bit que indica si este vértice está o no conectado al siguiente vértice mediante una arista del contorno. Por tanto, todo lo que necesitamos hacer es cambiar esta bandera de un bit a «desactivada» y la arista que sigue a dicho vértice no se visualizará. Establecemos esta bandera para una arista con la siguiente función.

```
glEdgeFlag (flag);
```

Para indicar que dicho vértice no precede a una arista del contorno, asignamos la constante de OpenGL `GL_FALSE` al argumento `flag`. Esto se aplica a todos los vértices que se especifiquen posteriormente hasta



**FIGURA 4.33.** La división de un polígono cóncavo (a) en un conjunto de triángulos (b) produce aristas de triángulos (con líneas discontinuas) que son interiores al polígono original.



**FIGURA 4.34.** El triángulo (a) se puede visualizar como aparece en (b) cambiando la bandera de la arista del vértice v2 al valor `GL_FALSE`, asumiendo que los vértices se especifican en sentido contrario a las agujas del reloj.

que se haga otra llamada a `glEdgeFlag`. La constante de OpenGL `GL_TRUE` activa de nuevo la bandera de la arista, el cual es su estado predeterminado. La función `glEdgeFlag` se puede situar entre pares `glBegin/glEnd`. Para mostrar el uso de una bandera de arista, el código siguiente muestra sólo dos aristas del triángulo definido (Figura 4.34).

```
glPolygonMode ( GL_FRONT_AND_BACK, GL_LINE );

glBegin ( GL_POLYGON );
    glVertex3fv ( v1 );
    glEdgeFlag ( GL_FALSE );
    glVertex3fv ( v2 );
    glEdgeFlag ( GL_TRUE );
    glVertex3fv ( v3 );
glEnd ( );
```

Las banderas de las aristas de los polígonos también se pueden especificar en un vector que se podría combinar o asociar a un vector de vértices (Secciones 3.17 y 4.3). Las líneas de código para crear un vector de banderas de aristas son:

```
 glEnableClientState ( GL_EDGE_FLAG_ARRAY );
 glEnableFlagPointer ( offset, edgeFlagArray );
```

El argumento `offset` indica el número de bytes entre los valores de las banderas de las aristas en el vector `edgeFlagArray`. El valor predeterminado del argumento `offset` es 0.

## La función de cara frontal de OpenGL

Aunque, de manera predeterminada, la ordenación de los vértices de un polígono controla la identificación de las caras frontal y trasera, podemos etiquetar de forma independiente las superficies seleccionadas de una escena como frontales o traseras con la función:

```
 glFrontFace ( vertexOrder );
```

Si cambiamos el argumento `vertexOrder` a la constante de OpenGL `GL_CW`, un polígono definido a continuación con una ordenación de sus vértices en el sentido de las agujas del reloj se considera que es de cara frontal. Esta característica de OpenGL se puede utilizar para intercambiar las caras de un polígono, en el que hayamos especificado sus vértices en el orden correspondiente al sentido horario. La constante `GL_CCW` etiqueta una ordenación de los vértices en sentido contrario a las agujas del reloj como de cara frontal, que es la ordenación predeterminada.

## 4.15 ATRIBUTOS DE LOS CARACTERES

---

La apariencia de los caracteres mostrados se controla con atributos tales como la fuente, el tamaño, el color y la orientación. En muchos paquetes, los atributos se pueden cambiar en cadenas de caracteres completas (texto) o en caracteres individuales, que se pueden utilizar para propósitos especiales tales como dibujar un gráfico de datos.

Hay disponible una gran cantidad de opciones posibles para la visualización de texto. En primer lugar, se puede elegir la fuente, que es un conjunto de caracteres con un diseño particular tal como New York, Courier, Helvetica, London, Times Roman y diversos grupos de símbolos especiales. Los caracteres de una fuente seleccionada se pueden mostrar también con diversos estilos de subrayado (continuo, punteado, doble), en **negrita**, en *cursiva* y letra hueca o sombreada.

La configuración del color para visualizar texto se puede almacenar en la lista de atributos del sistema y usarse con los procedimientos que generan las definiciones de los caracteres en el búfer de imagen. Cuando se ha de visualizar una cadena de caracteres, se utiliza el color actual para cambiar los valores de los píxeles del búfer de imagen, que se corresponden con las formas de los caracteres y sus posiciones.

Podríamos ajustar el tamaño del texto cambiando de escala las dimensiones de conjunto (altura y anchura) de los caracteres o sólo la altura o la anchura. El tamaño de los caracteres (altura) se especifica en las impresoras y por los compositores en *puntos*, donde 1 punto es, aproximadamente, 0,035146 cm (o 0,013837 pulgadas, lo cual es aproximadamente  $\frac{1}{72}$  pulgadas). Por ejemplo, los caracteres de este libro utilizan una fuente de 10 puntos. Las medidas en puntos especifican el tamaño del *cuerpo* de un carácter (Figura 4.35), pero fuentes diferentes con las mismas especificaciones en puntos pueden tener tamaños de caracteres diferentes, dependiendo del diseño de la fuente. La distancia entre la *línea inferior* y la *línea superior* del cuerpo de un carácter es la misma para todos los caracteres de un tamaño y una fuente concretas, pero el ancho del cuerpo puede variar. Las *fuentes proporcionalmente espaciadas* asignan un ancho de cuerpo más pequeño a los caracteres estrechos tales como *i, j, l* y *f* en comparación con caracteres anchos como *W* o *M*. La *altura del carácter* se define como la distancia entre la *línea base* y la *línea de tapa (cap)* de los caracteres. Los caracteres alargados (*kerned*), tales como *f* y *j* en la Figura 4.35, habitualmente se extienden más allá de los límites del cuerpo de los caracteres y las letras con extremos descendentes (*g, j, p, q, y*) se extienden por debajo de la línea de base. Cada carácter se posiciona dentro del cuerpo del carácter mediante un diseño de fuente, de tal modo que se logra un espacio adecuado a lo largo y entre las líneas impresas, cuando el texto se visualiza con cuerpos de caracteres que se tocan.

A veces, el tamaño del texto se ajusta sin cambiar la relación ancho-altura de los caracteres. La Figura 4.36 muestra una cadena de caracteres con tres alturas diferentes de caracteres, manteniendo la relación ancho-altura. En la Figura 4.37 se muestran ejemplos de texto con una altura constante y un ancho variable.

El espacio entre caracteres es otro atributo que a menudo se puede asignar a una cadena de caracteres. La Figura 4.38 muestra una cadena de caracteres con tres configuraciones diferentes para el espacio entre caracteres.

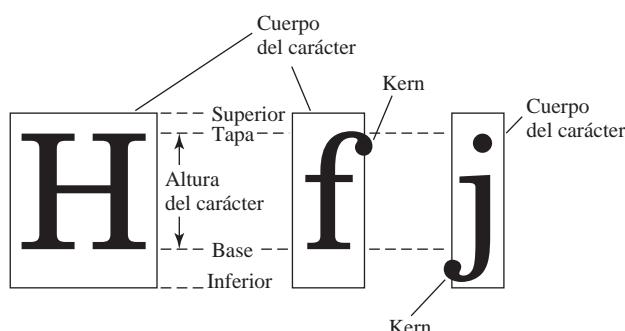


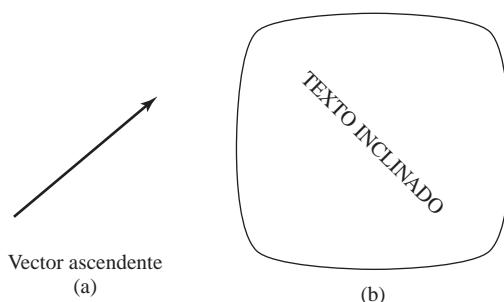
FIGURA 4.35. Ejemplos de cuerpos de caracteres.

Altura 1	anchura 0.5	Espaciado 0.0
Altura 2	anchura 1.0	Espaciado 0.5
Altura 3	anchura 2.0	Espaciado 1.0

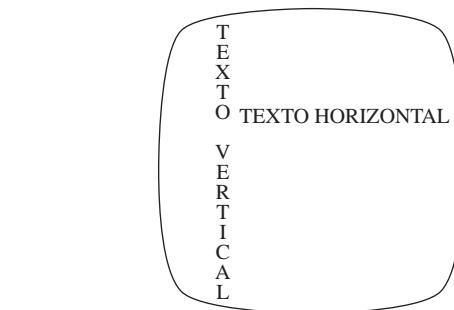
**FIGURA 4.36.** Cadenas de caracteres visualizadas con diferentes configuraciones de altura de cuerpo y con una relación ancho-altura constante.

**FIGURA 4.37.** Cadenas de caracteres visualizadas con tamaños que varían según el ancho del carácter pero no según su altura.

**FIGURA 4.38.** Cadenas de caracteres con diferentes valores de espacio entre caracteres.



**FIGURA 4.39.** Dirección del vector que apunta hacia arriba (a) que controla la orientación del texto mostrado (b).



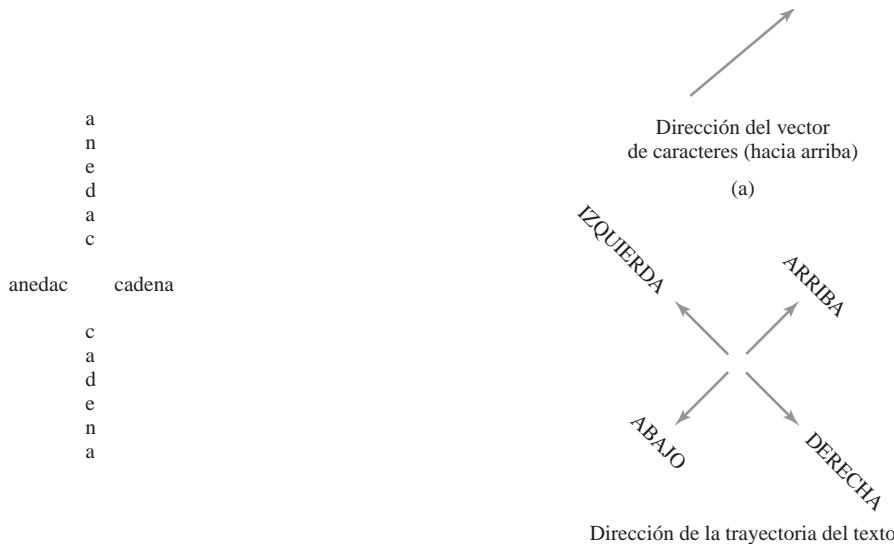
**FIGURA 4.40.** Los atributos de la trayectoria del texto se pueden cambiar para producir disposiciones horizontales o verticales de cadenas de caracteres.

La orientación de una cadena de caracteres se puede establecer según la dirección de un **vector de orientación de caracteres**. El texto se visualiza de modo que la orientación de los caracteres desde la línea base hasta la línea de tapa esté en la dirección del vector de orientación de caracteres. Por ejemplo, con la dirección del vector de orientación de caracteres a  $45^\circ$ , el texto se visualizaría como se muestra en la Figura 4.39. Un procedimiento para orientar texto podría rotar caracteres para que los lados de los cuerpos de los caracteres, desde la línea base hasta la línea de tapa, estén alineados con el vector de orientación de caracteres. Las formas de los caracteres rotados se convierten por barrido en el búfer de imagen.

Es útil en muchas aplicaciones poder disponer cadenas de caracteres vertical u horizontalmente. En la Figura 4.40 se muestran ejemplos de esto. También podríamos disponer los caracteres de una cadena para que se leyese hacia la izquierda o la derecha, o hacia arriba o hacia abajo. En la Figura 4.41 se muestran ejemplos de texto con estas opciones. Un procedimiento para implementar la orientación de la trayectoria del texto, ajusta la posición de los caracteres individuales en el búfer de imagen según la opción seleccionada.

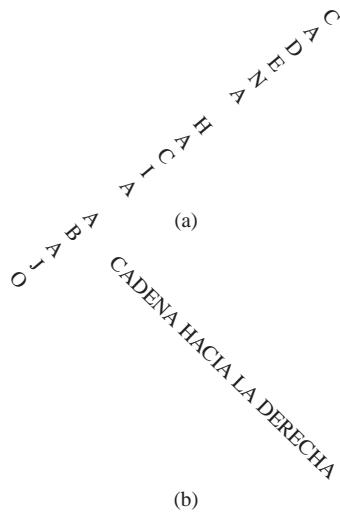
Las cadenas de caracteres también se pueden orientar utilizando una combinación de las especificaciones del vector de orientación de caracteres y de la trayectoria del texto, para producir texto inclinado. La Figura 4.42 muestra las direcciones de cadenas de caracteres generadas con varias configuraciones de trayectoria de texto y un vector de orientación de caracteres de  $45^\circ$ . En la Figura 4.43 se muestran ejemplos de cadenas de caracteres generadas con valores *abajo* y *derecha* de la trayectoria del texto y con este vector de orientación de caracteres.

Otro posible atributo de las cadenas de caracteres es la alineación. Este atributo especifica cómo se ha de visualizar el texto respecto a una posición de referencia. Por ejemplo, los caracteres individuales se podrían alinear según las líneas de base o el centro de los caracteres. La Figura 4.44 muestra las posiciones típicas de

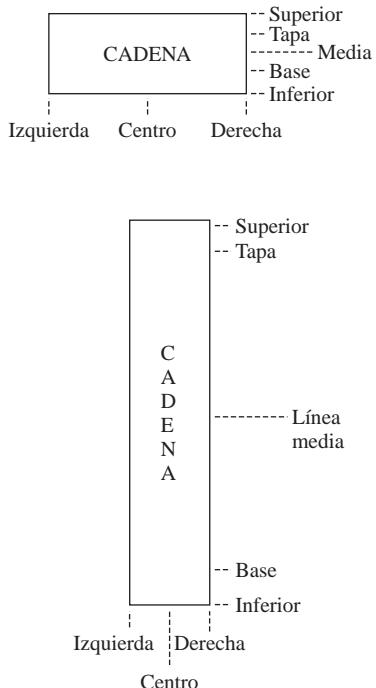


**FIGURA 4.41.** Una cadena de caracteres visualizada con las cuatro opciones de trayectoria de texto: izquierda, derecha, arriba y abajo.

**FIGURA 4.42.** Una especificación de vector de orientación de caracteres (a) y sus direcciones asociadas de la trayectoria del texto (b).



**FIGURA 4.43.** El vector de orientación de caracteres de 45° de la Figura 4.42 produce la representación (a) para una trayectoria hacia abajo y la representación (b) para una trayectoria hacia la derecha.



**FIGURA 4.44.** Alineaciones de los caracteres para cadenas horizontales y verticales.

A	S	ALINEACIÓN
L	U	DERECHA
I	P	
N	E	
E	R	A ALIENACIÓN
A	I	L CENTRADA
C	O	I I
I	R	N N
Ó		E F
N		A E
		C R
		I I
Ó	O	ALINEACIÓN
N	R	IZQUIERDA

**FIGURA 4.45.** Alineaciones de cadenas de caracteres.

los caracteres para alineaciones horizontales y verticales. También son posibles las alineaciones de las cadenas de caracteres. La Figura 4.45 muestra posiciones de alineación comunes para etiquetas de texto horizontales y verticales.

En algunos paquetes gráficos, también se encuentra disponible el atributo de precisión de texto. Este parámetro especifica la cantidad de detalle y las opciones de proceso particulares que se han de utilizar con una cadena de texto. Para una cadena de texto de baja precisión, muchos atributos, tales como la trayectoria del texto se ignoran, y se utilizan procedimientos más rápidos para procesar los caracteres a través de la *pipeline* de visualización.

Finalmente, una biblioteca de subrutinas de procesamiento de texto suministra a menudo un conjunto de caracteres especiales, tales como un pequeño círculo o una cruz, que son útiles en diversas aplicaciones. Estos caracteres se utilizan más a menudo como símbolos de marcación en esquemas de redes o en conjuntos de graficación de datos. Los atributos de estos símbolos de marcación son habitualmente el *color* y el *tamaño*.

## 4.16 FUNCIONES OpenGL PARA LOS ATRIBUTOS DE CARACTERES

---

Disponemos de dos métodos para visualizar caracteres con el paquete OpenGL. Podemos diseñar un conjunto de fuentes empleando las funciones de mapas de bits de la biblioteca del núcleo, o podemos invocar las subrutinas de generación de caracteres de GLUT. La biblioteca GLUT contiene funciones para la visualización de mapas de bits predefinidos y conjuntos de caracteres de impacto. Por tanto, podemos establecer que los atributos de los caracteres son aquellos que se aplican a los mapas de bits o los segmentos de líneas.

Para las fuentes de mapa de bits o de contorno, el estado de color actual determina el color de visualización. Por lo general, la designación de la fuente determina el espaciado y el tamaño de los caracteres, tal como GLUT\_BITMAT\_9\_BY\_15 y GLUT\_STROKE\_MONO\_ROMAN. Pero también podemos establecer el ancho de la línea y el tipo de línea de las fuentes de contorno. Especificamos el ancho de una línea con la función `glLineWidth`, y seleccionamos un tipo de línea con la función `glLineStipple`. Las fuentes de impacto de GLUT se visualizarán entonces utilizando los valores actuales que hayamos especificado para los atributos de ancho de línea y tipo de línea.

Podemos lograr algunas otras características de la visualización de texto empleando las funciones de transformación descritas en el Capítulo 5. Las subrutinas de transformación permiten cambiar de escala, posicionar y rotar los caracteres de impacto de GLUT en un espacio bidimensional o en un espacio tridimensional. Además, se pueden utilizar las transformaciones de visualización tridimensional (Capítulo 7) para generar otros efectos de visualización.

## 4.17 SUAVIZADO

---

Los segmentos de línea y otras primitivas gráficas generadas por algoritmos de barrido estudiadas en el Capítulo 3 tienen una apariencia dentada o de peldaño de escalera, porque el proceso de muestreo digitaliza

los puntos de coordenadas de un objeto en posiciones de píxel enteras y discretas. Esta distorsión de la información debida al muestreo de baja frecuencia (submuestreo) se denomina **aliasing**. Podemos mejorar la apariencia de las líneas digitalizadas mostradas aplicando métodos de suavizado (**antialiasing**), que compensen el proceso de submuestreo.

En la Figura 4.46 se muestra un ejemplo de los efectos del submuestreo. Para evitar perder información en tales objetos periódicos, necesitamos cambiar la frecuencia de muestreo a al menos dos veces la mayor frecuencia del objeto, que se denomina **frecuencia de muestreo de Nyquist** (o velocidad de muestreo de Nyquist) $f_s$ :

$$f_s = 2f_{\max} \quad (4.12)$$

Otro modo de manifestar esto consiste en que el intervalo de muestreo debería ser no mayor que la mitad del intervalo de ciclo (llamado **intervalo de muestreo de Nyquist**). Para un intervalo de muestreo según el eje  $x$ , el intervalo de muestreo de Nyquist  $\Delta x_s$  es:

$$\Delta x_s = \frac{\Delta x_{\text{ciclo}}}{2} \quad (4.13)$$

donde  $\Delta x_{\text{ciclo}} = 1/f_{\max}$ . En la Figura 4.46, el intervalo de muestreo es una vez y media el intervalo de ciclo, por lo que el intervalo de muestreo es al menos tres veces demasiado grande. Si queremos recuperar toda la información del objeto de este ejemplo, necesitamos reducir el intervalo de muestreo a un tercio del tamaño mostrado en la figura.

Un modo de incrementar la velocidad de muestreo en sistemas digitalizados consiste, simplemente, en mostrar los objetos con una resolución más alta. Pero incluso a la resolución más alta posible con la tecnología actual, aparecerá algún grado de dentado. Hay otro límite que viene dado por el tamaño máximo del búfer de imagen con el que se puede mantener una velocidad de refresco de 60 cuadros o más por segundo. Además, para representar objetos de forma precisa con parámetros continuos, necesitaríamos intervalos de muestreo arbitrariamente pequeños. Por tanto, a menos que se desarrolle tecnología hardware para manejar búferes de imagen arbitrariamente grandes, incrementar la resolución de la pantalla no es una solución completa para el problema del escalonamiento (*aliasing*).

En sistemas digitalizados que son capaces de visualizar más de dos niveles de intensidad por color, podemos aplicar métodos de suavizado para modificar las intensidades de los píxeles. Variando adecuadamente las intensidades de los píxeles a lo largo de los límites de las primitivas, podemos suavizar las aristas para reducir su apariencia dentada.

Un método de suavizado directo consiste en incrementar el período de muestreo, tratando la pantalla como si estuviese cubierta con una cuadrícula más fina que la disponible realmente. Podemos utilizar entonces múltiples puntos de muestreo a través de esta cuadrícula más fina para determinar un nivel de intensidad adecuado para cada píxel de la pantalla. Esta técnica de muestreo de las características de un objeto con una resolución alta y visualización de los resultados con una resolución más baja se denomina **supermuestreo** (o **posfiltrado**, ya que el método implica el cálculo de las intensidades en las posiciones de la cuadrícula de subpíxeles, para después combinar los resultados y obtener las intensidades de los píxeles). Las posiciones de los píxeles visualizados son manchas de luz que cubren un área finita de la pantalla, y no puntos matemáticos infinitesimales. Pero en los algoritmos de líneas y de relleno de áreas que hemos estudiado, la intensidad de cada píxel se determina mediante la localización de un único punto en los límites del objeto. Mediante supermuestreo, obtenemos información de intensidad de múltiples puntos que contribuyen a la intensidad global de un píxel.

Una alternativa al supermuestreo es determinar la intensidad del píxel calculando las áreas de superposición de cada píxel con los objetos que se van a mostrar. El suavizado mediante el cálculo de las áreas de superposición se denomina **muestreo de áreas** (o **prefiltrado**, ya que la intensidad del píxel como un todo se determina sin calcular intensidades de subpíxel). Las áreas de superposición de los píxeles se obtienen determinando dónde los límites de los objetos intersectan con las fronteras de los píxeles individuales.



**FIGURA 4.46.** El muestreo de la forma periódica (a) en las posiciones indicadas produce la representación (b) de baja frecuencia y escalonada (con *aliasing*).

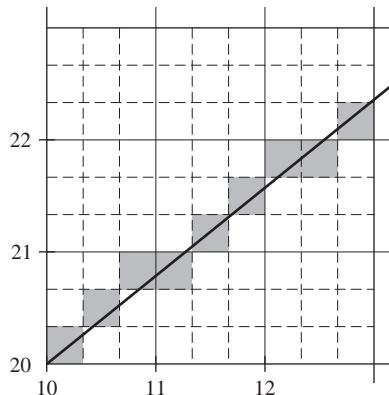
Los objetos digitalizados también se pueden suavizar cambiando la ubicación de visualización de las áreas de píxeles. Esta técnica, denominada **puesta en fase de los píxeles** (*pixel phasing*), se aplica «microposicionando» el haz de electrones en relación con la geometría del objeto. Por ejemplo, las posiciones de los píxeles a lo largo de un segmento de línea recta se pueden acercar a la trayectoria de la línea definida para suavizar el efecto de peldaño de escalera de la digitalización.

### Supermuestreo de segmentos de línea recta

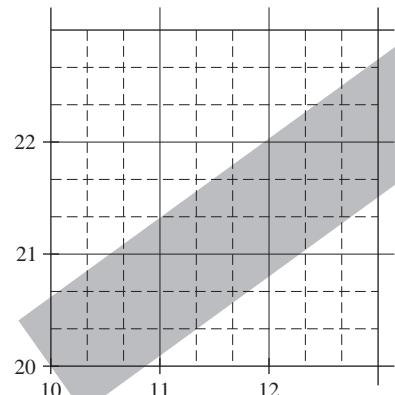
El supermuestreo se puede realizar de varias formas. En un segmento de línea recta, podemos dividir cada píxel en un número de subpíxeles y contar el número de subpíxeles que se superponen con la trayectoria de la línea. El nivel de intensidad de cada píxel se cambia entonces a un valor que es proporcional a este número de subpíxeles. En la Figura 4.47 se muestra un ejemplo de este método. Cada área cuadrada del píxel se divide en nueve subpíxeles cuadrados de igual tamaño. Las regiones sombreadas muestran los subpíxeles que se deberían seleccionar con el algoritmo de Bresenham. Esta técnica proporciona tres configuraciones de intensidad por encima de cero, ya que el número máximo de subpíxeles que se pueden seleccionar dentro de cada píxel es tres. En este ejemplo, el píxel de posición (10, 20) se cambia a la intensidad máxima (nivel 3); los píxeles de posiciones (11, 21) y (12, 21) se cambian ambos al siguiente nivel de intensidad más alto (nivel 2); y los píxeles de posiciones (11, 20) y (12, 22) se cambian ambos a la intensidad más baja por encima de cero (nivel 1). Por tanto, la intensidad de la línea se extiende sobre un número más grande de píxeles para suavizar el efecto de dentado original. Este procedimiento muestra una línea difuminada en la vecindad de los peldaños de escalera (entre recorridos horizontales). Si queremos utilizar más niveles de intensidad para suavizar la línea con este método, incrementaremos el número de posiciones de muestreo en cada píxel. Dieciséis subpíxeles proporcionan cuatro niveles de intensidad por encima de cero; veinticinco subpíxeles proporcionan cinco niveles; y así sucesivamente.

En el ejemplo de supermuestreo de la Figura 4.47, consideramos áreas de píxeles de tamaño finito, pero tratamos la línea como una entidad matemática de ancho cero. Realmente, las líneas visualizadas poseen un ancho aproximadamente igual al del píxel. Si tenemos en cuenta el ancho finito de la línea, podemos realizar supermuestreo cambiando la intensidad de los píxeles de forma proporcional al número de subpíxeles dentro del polígono que representa el área de la línea. Se puede considerar que un subpíxel se encuentra dentro de la línea si su esquina inferior izquierda está dentro de las fronteras del polígono. Una ventaja de este procedimiento de supermuestro es que el número de niveles de intensidad posibles para cada píxel, es igual al número total de subpíxeles dentro del área de píxeles. En el ejemplo de la Figura 4.47, podemos representar esta línea de ancho finito posicionando las fronteras del polígono, paralelas a la trayectoria de la línea como se muestra en la Figura 4.48. Ahora se puede cambiar cada píxel a uno de los nueve posibles niveles de brillo por encima de cero.

Otra ventaja del supermuestro con una línea de ancho finito es que la intensidad total de la línea se distribuye sobre más píxeles. En la Figura 4.48, ahora el píxel de posición (10, 21) en la cuadrícula está encendido (con intensidad de nivel 2), y también tenemos en cuenta las contribuciones de los píxeles inmediatamente inferiores e inmediatamente a su izquierda. También, si disponemos de una pantalla en color, podemos ampliar este método para tener en cuenta los colores de fondo. Una línea concreta podría cruzar varias áreas de colores diferentes. Podemos calcular el valor medio de las intensidades de los subpíxeles para obtener el



**FIGURA 4.47.** Supermuestreo de posiciones de subpíxeles a lo largo de un segmento de línea recta cuyo extremo izquierdo está en las coordenadas de pantalla (10, 20).



**FIGURA 4.48.** Supermuestreo de posiciones de subpíxeles en relación con el interior de una línea de ancho finito.

color de píxel. Por ejemplo, si cinco subpíxeles dentro de un área de píxel concreto se encuentran dentro de los límites de una línea de color rojo y los cuatro restantes píxeles se encuentran dentro de un área de fondo azul, podemos calcular el color de este píxel del siguiente modo:

$$\text{píxel}_{\text{color}} = \frac{(5 \cdot \text{rojo} + 4 \cdot \text{azul})}{9}$$

La desventaja del supermuestreo de una línea de ancho finito es que la identificación de los subpíxeles interiores requiere más cálculos que la simple determinación de qué subpíxeles se encuentran a lo largo de la trayectoria de la línea. También, necesitamos tener en cuenta el posicionamiento de los límites de la línea en relación con la trayectoria de la misma. Este posicionamiento depende de la pendiente de la línea. Para una línea con pendiente de  $45^\circ$ , la trayectoria de la línea está centrada en el área del polígono; pero para una línea horizontal o vertical, es deseable que la trayectoria de la línea sea uno de los límites del polígono. A modo de ejemplo, una línea horizontal que pase a través de las coordenadas de la cuadrícula (10, 20) se podría representar como un polígono limitado por las líneas horizontales de la cuadrícula  $y = 20$  e  $y = 21$ . De forma similar, el polígono que representa una línea vertical a través de (10, 20) puede tener sus límites verticales en las líneas de la cuadrícula  $x = 10$  y  $x = 11$ . En el caso de que la pendiente de la línea sea  $|m| < 1$ , la trayectoria de la línea matemática se posiciona proporcionalmente más cerca del límite inferior del polígono; y en el caso de que la pendiente sea  $|m| > 1$ , la trayectoria de la línea se sitúa más cerca del límite superior del polígono.

### Máscaras de ponderación de subpíxeles

Los algoritmos de supermuestreo se implementan a menudo dando más peso a los subpíxeles próximos al centro del área de un píxel, ya que se espera que estos subpíxeles sean más importantes en la determinación de la intensidad global de un píxel. En las subdivisiones 3 por 3 de los píxeles que hemos considerado hasta ahora, es posible utilizar una combinación de pesos como los de la Figura 4.49. Aquí el subpíxel central tiene un peso que es cuatro veces el peso de los subpíxeles de las esquinas y dos veces el peso del resto de los subpíxeles. Las intensidades calculadas para cada uno de los nueve subpíxeles se promediarían entonces para que el subpixel central se pondere con un factor de  $\frac{1}{4}$ ; los píxeles superior, inferior y laterales se pondren con un factor de  $\frac{1}{8}$ ; y los subpíxeles de las esquinas con un factor de  $\frac{1}{16}$ . Una matriz de valores que especifica la importancia relativa de los subpíxeles se denomina, habitualmente, *máscara de ponderación*. Se pueden establecer máscaras similares para cuadrículas de subpíxeles mayores. También, estas máscaras se amplían habitualmen-

te para incluir las contribuciones de los subpíxeles pertenecientes a los píxeles vecinos, con el fin de que las intensidades se puedan promediar con los píxeles adyacentes para proporcionar una variación de intensidad más suave entre píxeles.

### Muestreo por área de segmentos de línea recta

Realizamos un muestreo por área de una línea recta cambiando la intensidad de los píxeles proporcionalmente al área de superposición del píxel con la línea de ancho finito. La línea se puede tratar como un rectángulo. La sección del área de la línea entre dos líneas adyacentes verticales (o dos adyacentes horizontales) de la cuadrícula de pantalla es entonces un trapezoide. Las áreas de superposición de los píxeles se calculan determinando cuánto el trapezoide se superpone sobre cada píxel en aquella columna (o fila). En la Figura 4.48, el píxel de coordenadas de la cuadrícula (10, 20) se cubre, aproximadamente, al 90 por ciento por el área de la línea, por lo que su intensidad se cambiaría al 90 por ciento de la intensidad máxima. De forma similar, el píxel de posición (10, 21) se cambiaría a una intensidad de, aproximadamente, el 15% del máximo. En la Figura 4.48 se muestra un método de estimación de las áreas de superposición de los píxeles con un ejemplo de supermuestreo. El número total de subpíxeles dentro de los límites de la línea es aproximadamente igual al área de superposición. Esta estimación se puede mejorar usando cuadrículas de subpíxeles más finas.

### Técnicas de filtrado

Un método más preciso para suavizar líneas consiste en usar técnicas de **filtrado**. El método es similar a aplicar una máscara de pesos de píxeles, pero ahora se utiliza una *superficie de pesos* (o *función de filtrado*) con-

1	2	1
2	4	2
1	2	1

FIGURA 4.49. Pesos relativos para una cuadrícula de subpíxeles 3 por 3.

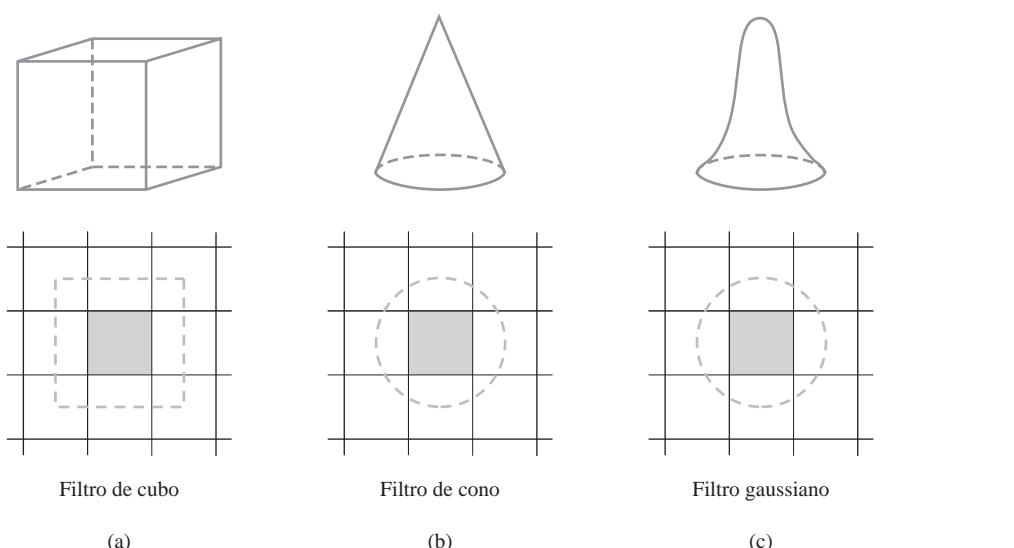


FIGURA 4.50. Funciones de filtrado que se usan habitualmente para suavizar las trayectorias de las líneas. El volumen de cada filtro está normalizado al valor 1. La altura proporciona el peso relativo de cada posición de subpíxel.

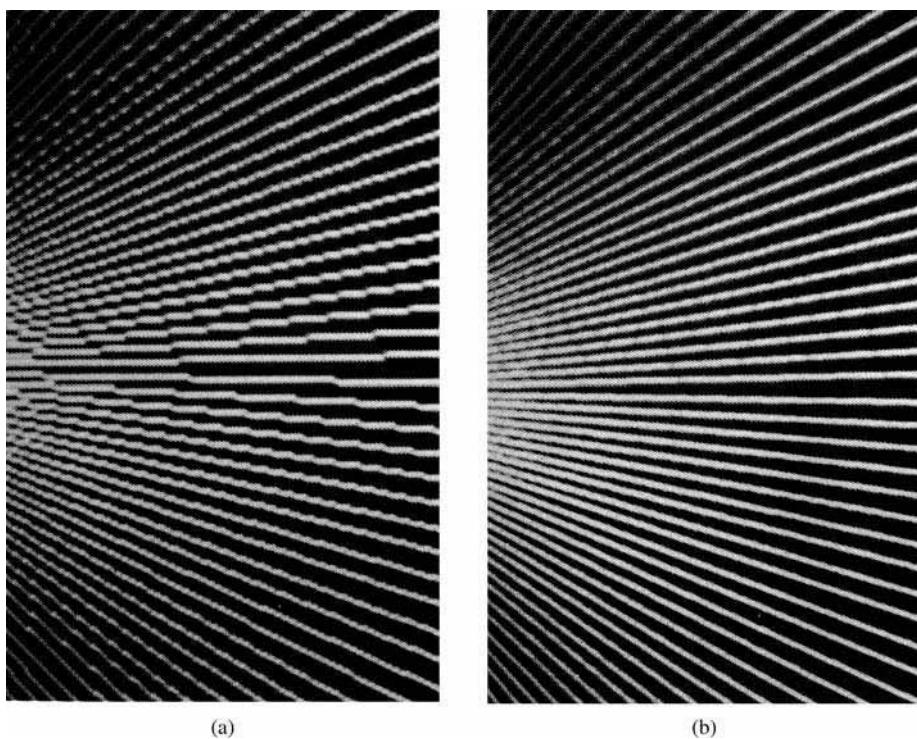
tinua que cubre el píxel. La Figura 4.50 muestra ejemplos de funciones de filtrado rectangulares, cónicas y gaussianas. Los métodos de aplicación de la función de filtrado son similares a los empleados en la máscara de pesos, pero ahora integramos sobre la superficie del píxel para obtener la intensidad promedio ponderada. Para reducir los cálculos, se utilizan habitualmente tablas de búsqueda para evaluar las integrales.

## Ajuste de fase de los píxeles

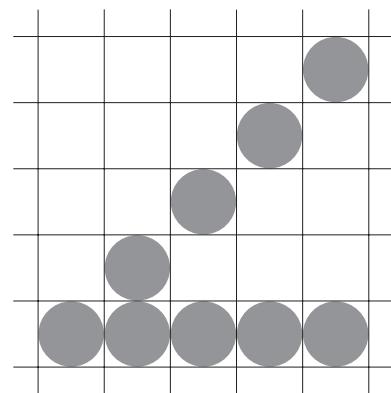
En los sistemas digitalizados que pueden tratar posiciones de subpíxeles dentro de la cuadrícula de la pantalla, se puede utilizar el ajuste de fase de los píxeles para suavizar los objetos. Una línea de la pantalla se suaviza con esta técnica moviendo (microposicionando) las posiciones de los píxeles más próximas a la trayectoria de la línea. Los sistemas que incorporan *ajuste de fase de los píxeles* se diseñan para que el haz de electrones se pueda modificar en una fracción del diámetro del píxel. El haz de electrones se modifica habitualmente en  $\frac{1}{4}$ ,  $\frac{1}{2}$  o  $\frac{3}{4}$  del diámetro del píxel para dibujar los puntos más cercanos a la verdadera trayectoria de la línea o del borde del objeto. Algunos sistemas también permiten ajustar el tamaño de píxeles individuales como medio adicional de distribución de intensidades. La Figura 4.51 muestra los efectos de suavizado del ajuste de fase de los píxeles para varias trayectorias de línea.

## Compensación de diferencias en la intensidad de las líneas

El suavizado de una línea para eliminar el efecto de peldaño de escalera también compensa otro efecto de la digitalización, como se muestra en la Figura 4.52. Ambas líneas se dibujan con el mismo número de píxeles, pero la línea diagonal es más larga que la horizontal en un factor de  $\sqrt{2}$ . Por ejemplo, si la línea horizontal



**FIGURA 4.51.** Las líneas dentadas (a), dibujadas en un sistema Merlin 9200, se suavizan (b) con una técnica de suavizado llamada ajuste de fase de los píxeles. Esta técnica incrementa el número de puntos que puede tratar el sistema de 768 por 576 a 3072 por 2304. (Cortesía de Peritek Corp.)



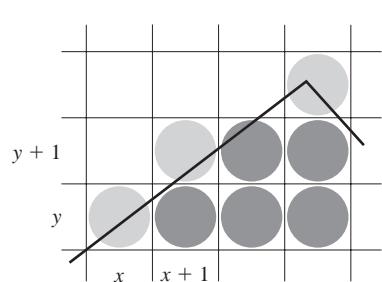
**FIGURA 4.52.** Líneas de diferente longitud dibujadas con el mismo número de píxeles en cada línea.

tenía una longitud de 10 centímetros, la línea diagonal tendría una longitud de más de 14 centímetros. El efecto visual de esto es que la línea diagonal aparece menos brillante que la línea horizontal, ya que la línea diagonal se muestra con una menor intensidad por unidad de longitud. El algoritmo de dibujo de líneas se podría adaptar para compensar este efecto ajustando la intensidad de cada línea según su pendiente. Las líneas horizontales y verticales se mostrarían con la menor intensidad, mientras que a las líneas a  $45^\circ$  se las dotaría de la intensidad más elevada. Pero si se aplican técnicas de suavizado a una visualización, las intensidades se compensan automáticamente. Cuando el ancho finito de una línea se tiene en consideración, las intensidades de los píxeles se ajustan para que la línea se muestre con una intensidad total proporcional a su longitud.

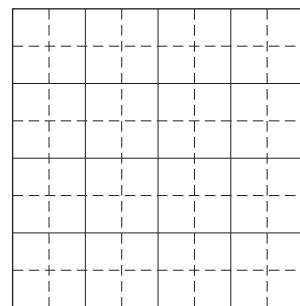
### Suavizado de los límites de las áreas

Los conceptos de suavizado que hemos estudiado para las líneas, también se pueden aplicar a los límites de las áreas para eliminar la apariencia dentada. Podríamos incorporar estos procedimientos al algoritmo de líneas de barrido para suavizar los límites al generar un área.

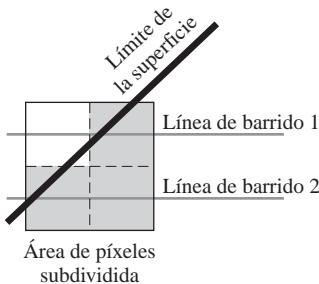
Si las capacidades del sistema permiten el reposicionamiento de los píxeles, se podrían suavizar los límites de las áreas modificando las posiciones de los píxeles más cercanos al contorno. Otros métodos ajustan la intensidad de los píxeles de los límites según el porcentaje del área del píxel que es interior al objeto. En la Figura 4.53, el píxel de la posición  $(x, y)$  tiene aproximadamente la mitad de su área dentro del contorno del polígono. Por tanto, la intensidad en dicha posición se debería ajustar a la mitad de su valor asignado. En la siguiente posición  $(x + 1, y + 1)$  a lo largo del límite, la intensidad se ajusta a aproximadamente un tercio del valor asignado para dicho punto. Ajustes similares, basados en el porcentaje del área del píxel cubierta, se aplican a otros valores de la intensidad alrededor del límite.



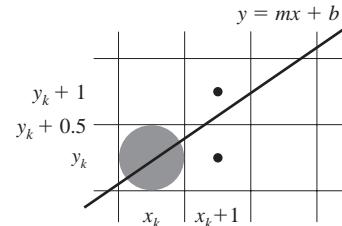
**FIGURA 4.53.** Ajuste de las intensidades de los píxeles a lo largo del límite de un área.



**FIGURA 4.54.** Sección de píxeles 4 por 4 de una pantalla digitalizada subdividida en una cuadrícula de 8 por 8.



**FIGURA 4.55.** Un área de píxeles subdividida con tres subdivisiones dentro de la línea de frontera de un objeto.



**FIGURA 4.56.** Borde de un área de relleno que pasa a través de una sección de una cuadricula de píxeles.

Los métodos de supermuestreo se pueden aplicar determinando el número de subpíxeles que se encuentran en el interior de un objeto. En la Figura 4.54 se muestra un esquema de particionamiento con cuatro subáreas por píxel. La cuadricula original de 4 por 4 píxeles se transforma en una cuadricula de 8 por 8. Ahora procesamos ocho líneas de barrido a través de esta cuadricula en lugar de cuatro. La Figura 4.55 muestra una de las áreas de los píxeles de esta cuadricula que se superpone con el límite de un objeto. Con estas dos líneas de barrido, determinaremos qué tres áreas de subpíxeles se encuentran dentro del límite de la superficie. Por tanto, establecemos la intensidad del píxel al 75% de su valor máximo.

Otro método para determinar el porcentaje del área de píxel dentro un área de relleno, desarrollado por Pitteway y Watkinson, se basa en el algoritmo de la línea del punto medio. Este algoritmo selecciona el píxel siguiente a lo largo de una línea comprobando la situación de la posición media entre dos píxeles. Como en el algoritmo de Bresenham, establecemos un parámetro de decisión  $p$  cuyo signo indica cuál de los dos píxeles candidatos siguientes está más cerca de la línea. Modificando ligeramente la forma de  $p$ , obtenemos una cantidad que también proporciona el porcentaje del área actual de píxel que está cubierta por un objeto.

En primer lugar, consideramos el método para una línea con una pendiente  $m$  dentro del rango que varía entre 0 y 1. En la Figura 4.56, se muestra una trayectoria de línea recta en una cuadricula de píxeles. Asumiendo que se ha dibujado el píxel de la posición  $(x_k, y_k)$ , el siguiente píxel más cercano a la línea en  $x = x_k + 1$  es el píxel en  $y_k$  o en  $y_k + 1$ . Podemos determinar qué píxel es el más cercano con el cálculo:

$$y - y_{\text{mid}} = [m(x_k + 1) + b] - (y_k + 0.5) \quad (4.14)$$

Este cálculo proporciona la distancia vertical desde la verdadera coordenada  $y$  de la línea al punto intermedio entre los píxeles de las posiciones  $y_k$  e  $y_k + 1$ . Si esta diferencia es negativa, el píxel  $y_k$  es más cercano a la línea. Si la diferencia es positiva, el píxel  $y_k + 1$  es más cercano. Podemos ajustar este cálculo para que produzca un número positivo dentro del rango que varía desde 0 a 1 añadiendo la cantidad  $1 - m$ :

$$p = [m(x_k + 1) + b] - (y_k + 0.5) + (1 - m)1 \quad (4.15)$$

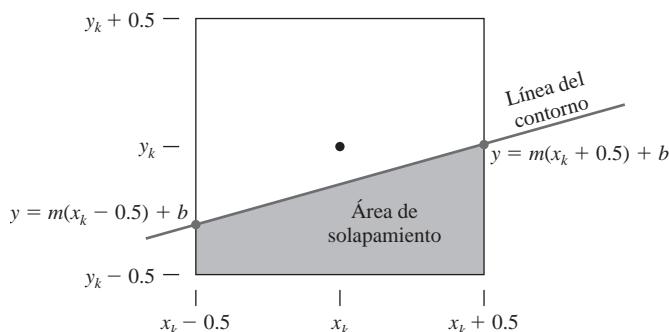
Ahora el píxel en  $y_k$  es más cercano si  $p < 1 - m$ , y el píxel en  $y_k + 1$  es más cercano si  $p > 1 - m$ .

El parámetro  $p$  también mide la cantidad del píxel actual que se superpone con el área. Para el píxel de posición  $(x_k, y_k)$  de la Figura 4.57, la parte interior del píxel tiene un área que se puede calcular del siguiente modo

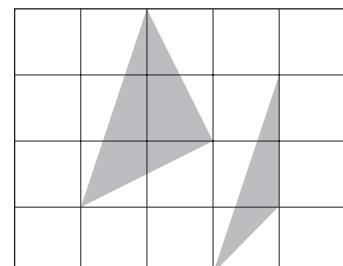
$$\text{Área} = m \cdot x_k + b - y_k + 0.5 \quad (4.16)$$

Esta expresión del área de superposición del píxel de la posición  $(x_k, y_k)$  es la misma del parámetro  $p$  de la Ecuación 4.15. Por tanto, evaluando  $p$  para determinar la posición del píxel siguiente a lo largo del contorno del polígono, también determinamos el porcentaje del área cubierta del píxel actual.

Podemos generalizar este algoritmo para tener en cuenta las líneas con pendientes negativas y las líneas con pendientes mayores que 1. Este cálculo del parámetro  $p$  se podría entonces incorporar al algoritmo de la



**FIGURA 4.57.** Área de superposición de un píxel rectangular, centrado en la posición  $(x_k, y_k)$ , con el interior del área de relleno de un polígono.



**FIGURA 4.58.** Polígonos con más de una línea de su contorno que pasa a través de regiones de píxeles individuales.

línea de punto medio, para localizar las posiciones de los píxeles a lo largo de la arista del polígono y, de forma concurrente, ajustar las intensidades de los píxeles a lo largo de las líneas del contorno. También, podemos ajustar los cálculos para referenciar las coordenadas de los píxeles de coordenadas más bajas de la parte izquierda y mantener las proporciones del área, como se estudió en el Sección 3.13.

En los vértices del polígono y para muchos polígonos estrechos, como los mostrados en la Figura 4.58, tenemos más de una arista del contorno que pasa a través del área de un píxel. En estos casos, necesitamos modificar el algoritmo de Pitteway-Watkinson procesando todas las aristas que pasan a través de un píxel y determinando el área interior correcta.

Las técnicas de filtrado estudiadas para el suavizado de líneas se pueden también aplicar a las aristas de un área. Los variados métodos de suavizado se pueden aplicar a áreas de polígonos o a regiones con límites curvos. Las ecuaciones que describen los límites se utilizan para estimar la cantidad de píxel que se superpone con el área a visualizar. Las técnicas de coherencia se utilizan a lo largo y entre líneas de barrido para simplificar los cálculos.

## 4.18 FUNCIONES OpenGL DE SUAVIZADO

Activamos las subrutinas de suavizado de OpenGL con la función:

```
glEnable (primitiveType);
```

donde al argumento `primitiveType` se le asignan las constantes simbólicas `GL_POINT_SMOOTH`, `GL_LINE_SMOOTH` o `GL_POLYGON_SMOOTH`.

Asumiendo que especificamos los valores de color utilizando el modo RGBA, también necesitamos activar las operaciones de fundido de color de OpenGL:

```
glEnable (GL_BLEND);
```

A continuación, aplicamos el método de fundido de color descrito en el Sección 4.3 utilizando la función:

```
glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

Las operaciones de suavizado son más efectivas si utilizamos valores grandes de la componente alfa en las especificaciones de color de los objetos.

El suavizado también se puede aplicar cuando utilizamos tablas de color. Sin embargo, en este modo de color, debemos crear una *rampa de color*, que es una tabla de gradación del color desde el color de fondo hasta el color del objeto. Esta rampa de color se utiliza después para suavizar los límites del objeto.

## 4.19 FUNCIONES DE CONSULTA DE OpenGL

---

Podemos obtener los valores actuales de los parámetros de estado, incluyendo las configuraciones de los atributos, empleando las **funciones de consulta** de OpenGL. Estas funciones copian los valores de estado específicos en una matriz, que podemos guardar para reutilizarla posteriormente o para comprobar el estado actual del sistema si se produce un error.

Para los valores de los atributos actuales utilizamos una función apropiada «`glGet`», tal como:

```
glGetBooleanv ( )      glGetFloatv ( )
glGetIntegerv ( )     glGetDoublev ( )
```

En cada una de las funciones precedentes, especificamos dos argumentos. El primer argumento es una constante simbólica de OpenGL que identifica un atributo u otro parámetro de estado. El segundo argumento es un puntero a una matriz del tipo de datos indicado por el nombre de la función. Por ejemplo, podemos obtener la configuración del color RGBA en punto flotante actual con:

```
glGetFloatv (GL_CURRENT_COLOR, colorValues);
```

Las componentes de color actuales se pasan entonces a la matriz `colorValues`. Para obtener los valores enteros de las componentes de color actuales, invocamos la función `glGetIntegerv`. En algunos casos, se puede necesitar una conversión de tipos para devolver el tipo de datos especificado.

Otras constantes de OpenGL, tales como `GL_POINT_SIZE`, `GL_LINE_WIDTH` y `GL_CURRENT_RASTER_POSITION`, se pueden utilizar en estas funciones para devolver los valores de estado actuales. Podríamos comprobar el rango de los tamaños de los puntos o los anchos de las líneas permitidos utilizando las constantes `GL_POINT_SIZE_RANGE` y `GL_LINE_WIDTH_RANGE`.

Aunque podemos obtener y reutilizar la configuración de un único atributo con las funciones `glGet`, OpenGL proporciona otras funciones para guardar grupos de atributos y reutilizar sus valores. Veremos el uso de estas funciones para guardar las configuraciones actuales de los atributos en la siguiente sección.

Hay muchas otros parámetros de estado y de sistema que son útiles a menudo para consulta. Por ejemplo, para determinar cuantos bits por píxel se proporcionan en el búfer de imagen en un sistema concreto, podemos preguntar al sistema cuántos bits hay disponibles para cada componente individual de color, del siguiente modo:

```
glGetIntegerv (GL_RED_BITS, redBitSize);
```

En esta instrucción, a la matriz `redBitSize` se le asigna el número de bits de la componente roja disponibles en cada uno de los búferes (búfer de imagen, búfer de profundidad, búfer de acumulación y búfer de patrones). De forma similar, podemos preguntar por los otros bits de color utilizando `GL_GREEN_BITS`, `GL_BLUE_BITS`, `GL_ALPHA_BITS` o `GL_INDEX_BITS`.

También podemos conocer si se han establecido las banderas de las aristas, si una cara de un polígono se ha etiquetado como cara frontal o cara posterior, y si el sistema soporta doble búfer. Podemos preguntar si ciertas subrutinas, tales como las de fundido de color, punteado de líneas o suavizado, se han habilitado o deshabilitado.

## 4.20 GRUPOS DE ATRIBUTOS DE OpenGL

---

Los atributos y otros parámetros de estado de OpenGL están organizados en **grupos de atributos**. Cada grupo contiene un conjunto de parámetros de estado relacionados. Por ejemplo, el **grupo de atributos de los puntos** contiene los parámetros del tamaño y de suavizado (*antialiasing*) de los puntos y el **grupo de atributos de las líneas** contiene el grosor, el estado de los trazos, el patrón de los trazos, el contador de repetición de los trazos y el estado de suavizado de las líneas. De forma similar, el **grupo de atributos de los polígonos** contiene once parámetros de los polígonos, tales como el patrón de relleno, la bandera de cara frontal y el estan-

do de suavizado de los polígonos. Ya que el color es un atributo común a todas las primitivas, éste tiene su propio grupo de atributos. Algunos parámetros se incluyen en más de un grupo.

Aproximadamente hay disponibles veinte grupos de atributos diferentes en OpenGL. Todos los parámetros de uno o más grupos se pueden guardar o reestablecer (reset) con una única función. Con el siguiente comando se guardan todos los parámetros de un grupo especificado.

```
glPushAttrib (attrGroup);
```

Al parámetro `attrGroup` se le asigna una constante simbólica de OpenGL que identifica el grupo de atributos, tal como `GL_POINT_BIT`, `GL_LINE_BIT` o `GL_POLYGON_BIT`. Para guardar los parámetros del color, utilizamos la constante simbólica `GL_CURRENT_BIT`. Podemos guardar todos los parámetros de estado de todos los grupos de atributos con la constante `GL_ALL_ATTRIB_BITS`. La función `glPushAttrib` sitúa todos los parámetros del grupo especificado en la **pila de atributos**.

También podemos guardar los parámetros de dos o más grupos combinando sus constantes simbólicas con la operación lógica OR. La siguiente línea sitúa todos los parámetros de los puntos, las líneas y los polígonos en la pila de atributos.

```
glPushAttrib (GL_POINT_BIT | GL_LINE_BIT | GL_POLYGON_BIT);
```

Una vez que hemos guardado un grupo de parámetros de estado, podemos rehabilitar todos los valores de la pila de atributos con la función:

```
glPopAttrib ( );
```

No se utilizan argumentos en la función `glPopAttrib` porque ésta reestablece el estado actual de OpenGL utilizando todos los valores de la pila.

Estos comandos para guardar y reestablecer los parámetros de estado utilizan un *servidor de pila de atributos*. En OpenGL, también hay disponible un *cliente de pila de atributos* para guardar y reestablecer los parámetros de estado del cliente. Las funciones de acceso a esta pila son `glPushClientAttrib` y `glPopClientAttrib`. Sólo hay disponibles dos grupos de atributos de cliente: uno para los modos de almacenamiento de píxel y el otro para las matrices de vértices. Entre los parámetros de almacenamiento de píxel se incluye información tal como la alineación de bytes y el tipo de matrices utilizadas para almacenar subimágenes de una pantalla. Los parámetros para las matrices de vértices proporcionan información acerca del estado actual de las matrices de vértices, tal como el estado habilitado/deshabilitado de diversas matrices.

## 4.21 RESUMEN

---

Los atributos controlan las características de visualización de las primitivas gráficas. En muchos sistemas gráficos, los valores de los atributos se almacenan como variables de estado y las primitivas se generan utilizando los valores actuales de los atributos. Cuando cambiamos el valor de una variable de estado, este cambio sólo afecta a las primitivas definidas después de éste.

Un atributo común en todas primitivas es el color, que se especifica muy a menudo en términos de componentes RGB (o RGBA). Los valores de color rojo, verde y azul se almacenan en el búfer de imagen, y se utilizan para controlar la intensidad de los tres cañones de un monitor RGB. Las selecciones de color se pueden hacer también utilizando tablas de búsqueda de colores. En este caso, un color del búfer de imagen se indica como un índice de una tabla, y la posición de la tabla de dicho índice almacena un conjunto concreto de valores de color RGB. Las tablas de color son útiles en la visualización de datos y en aplicaciones de procesamiento de imágenes, y también se pueden utilizar para proporcionar un gran rango de colores sin que se requiera un gran búfer de imagen. A menudo, los paquetes de gráficos por computadora proporcionan opciones para utilizar tablas o almacenar los valores de color directamente en el búfer de imagen.

Los atributos básicos de los puntos son el color y el tamaño. En sistemas digitalizados, los diversos tamaños de los puntos se visualizan como matrices cuadradas de píxeles. Los atributos de las líneas son el color,

el grosor y el estilo. Las especificaciones del grosor de una línea se dan en términos de múltiplos de una línea estándar de un píxel de grosor. Entre los atributos del estilo de las líneas se incluyen las líneas de trazo continuo, de trazo discontinuo y de puntos, así como varios estilos de brochas y plumillas. Estos atributos se pueden aplicar tanto a líneas rectas como a líneas curvas.

Entre los atributos de relleno de líneas se incluyen el relleno de color liso, el relleno con patrón o la visualización hueca que sólo muestra los límites del área. Se pueden especificar varios patrones de relleno con matrices de color, que se mapean al interior de la región. Los métodos de línea de barrido se utilizan habitualmente para llenar polígonos, círculos o elipses. A través de cada línea de barrido se aplica el relleno interior a las posiciones de los píxeles entre cada par de intersecciones con los límites, de izquierda a derecha. En el caso de los polígonos, las intersecciones de las líneas de barrido con los vértices pueden producir un número impar de intersecciones. Esto se puede resolver acortando algunas aristas del polígono. Los algoritmos de relleno por líneas de barrido se pueden simplificar si las áreas que se van a llenar se restringen a polígonos convexos. Se puede lograr una mayor simplificación si todas las áreas que hay que llenar de una escena son triángulos. A los píxeles interiores a lo largo de cada línea de barrido se les asigna el valor de color adecuado, dependiendo de las especificaciones de los atributos de relleno. Generalmente, los programas de dibujo muestran las regiones llenadas empleando un método de relleno por contorno o un método de relleno por inundación. Cada uno de estos dos métodos de relleno requiere un punto interior de partida. El interior se pinta a continuación píxel a píxel desde el punto inicial hasta los límites de la región.

Las áreas también se pueden llenar utilizando fundido de color. Este tipo de relleno se aplica para el suavizado y en paquetes de dibujo. Los procedimientos de relleno suave proporcionan un nuevo color de relleno de una región que tiene las mismas variaciones que el color de relleno previo. Un ejemplo de esta técnica es el algoritmo de relleno suave lineal, que supone que el relleno previo era una combinación lineal de los colores de primer plano y de fondo. Entonces se determina esta misma relación lineal a partir de la configuración del búfer de imagen y se utiliza para volver a pintar el área con el nuevo color.

Los caracteres se pueden visualizar con diferentes estilos (fuentes), colores, tamaños, espaciados y orientaciones. Para cambiar la orientación de una cadena de caracteres, podemos especificar una dirección para el vector de orientación de caracteres y una dirección para la trayectoria del texto. Además, podemos establecer la alineación de una cadena de caracteres en relación con una posición de comienzo en coordenadas. Los caracteres individuales, llamados símbolos de marcación, se pueden utilizar en aplicaciones tales como el dibujo de gráficos de datos. Los símbolos de marcación se pueden visualizar con varios tamaños y colores utilizando caracteres estándar o símbolos especiales.

Ya que la conversión por líneas es un proceso de digitalización en sistemas digitales, las primitivas visualizadas poseen una apariencia dentada. Esto se debe al submuestreo de la información, que redondea los valores de las coordenadas a las posiciones de los píxeles. Podemos mejorar la apariencia de las primitivas de barrido aplicando procedimientos de suavizado (*antialiasing*), que ajusten las intensidades de los píxeles. Un método para hacer esto es el supermuestreo. Es decir, consideramos que cada píxel está compuesto por subpíxeles, calculamos la intensidad de los subpíxeles y promediamos los valores de todos los subpíxeles. También podemos ponderar las contribuciones de los subpíxeles según su posición, asignando pesos más altos a los subpíxeles centrales. De forma alternativa, podemos realizar un muestreo por área y determinar el porcentaje del área cubierta de un píxel de pantalla, para después establecer la intensidad de píxel de forma proporcional a este porcentaje. Otro método de suavizado consiste en construir configuraciones especiales de hardware que puedan modificar las posiciones de sus píxeles.

En OpenGL, los valores de los atributos de las primitivas se almacenan en forma de variables de estado. Una configuración de un atributo se utiliza en todas las primitivas definidas posteriormente hasta que se cambie el valor del atributo. El cambio del valor de un atributo no afecta a las primitivas representadas anteriormente. Podemos especificar colores en OpenGL utilizando el modo de color RGB (RGBA) o el modo de color indexado, que utiliza los índices de una tabla de colores para seleccionar los colores. También, podemos fundir los colores utilizando la componente de color alfa. Y podemos especificar valores en matrices de color que hay que usarlas junto con matrices de vectores. Además del color, OpenGL proporciona funciones para seleccionar el tamaño del punto, el grosor de las líneas, el estilo de las líneas y el estilo de relleno de polígonos.

convexos, así como funciones para visualizar las áreas de relleno de polígonos como un conjunto de aristas o un conjunto de puntos en sus vértices. También podemos eliminar aristas de polígonos seleccionadas y podemos invertir la especificación de las caras frontal y posterior. Podemos generar cadenas de texto en OpenGL utilizando mapas de bits o subrutinas que están disponibles en GLUT. Entre los atributos que se pueden establecer para visualizar caracteres de GLUT se incluye el color, la fuente, el tamaño, el espaciado, el grosor de las líneas y el tipo de línea. La biblioteca OpenGL también proporciona funciones para suavizar la visualización de las primitivas de salida. Podemos utilizar funciones de consulta para obtener los valores actuales de las variables de estado y podemos también obtener todos los valores de un grupo de atributos de OpenGL utilizando una única función.

La Tabla 4.2 resume las funciones de atributos de OpenGL estudiadas en este capítulo. Además, la tabla enumera algunas funciones relacionadas con los atributos.

**TABLA 4.2. RESUMEN DE LAS FUNCIONES DE LOS ATRIBUTOS DE OPENGL.**

<i>Función</i>	<i>Descripción</i>
<code>glutInitDisplayMode</code>	Selecciona el modo de color, que puede ser <code>GLUT_RGB</code> o <code>GLUT_INDEX</code> .
<code> glColor*</code>	Especifica un color RGB o RGBA.
<code>glIndex*</code>	Especifica un color utilizando un índice de tabla de color.
<code>glutSetColor (index, r, g, b);</code>	Carga un color en una posición de la tabla de color.
<code> glEnable (GL_BLEND);</code>	Activa el fundido de color.
<code> glBlendFunc (sFact, dFact);</code>	Especifica los factores del fundido de color.
<code> glEnableClientState ; (GL_COLOR_ARRAY)</code>	Activa las características de matriz de color de OpenGL.
<code> glColorPointer (size, type, stride, array);</code>	Especifica una matriz de color RGB.
<code> glIndexPointer (type, stride, array);</code>	Especifica una matriz de color usando el modo de color indexado.
<code> glPointSize (size);</code>	Especifica un tamaño de punto.
<code> glLineWidth (width);</code>	Especifica un grosor de línea.
<code> glEnable (GL_LINE_STIPPLE);</code>	Activa el estilo de las líneas.
<code> glEnable (GL_POLYGON_STIPPLE);</code>	Activa el estilo de relleno.
<code> glLineStipple (repeat, pattern);</code>	Especifica el patrón de estilo de línea.
<code> glPolygonStipple (pattern);</code>	Especifica el patrón de relleno.
<code> glPolygonMode</code>	Muestra la cara frontal o la cara posterior como un conjunto de aristas o como un conjunto de vértices.
<code> glEdgeFlag</code>	Establece la bandera de la arista del relleno de polígonos en el valor <code>GL_TRUE</code> o <code>GL_FALSE</code> para determinar el estado de visualización de una arista.
<code> glFrontFace</code>	Especifica el orden de los vértices de la cara frontal como <code>GL_CCW</code> o <code>GL_CW</code> .

(Continúa)

**TABLA 4.2.** RESUMEN DE LAS FUNCIONES DE LOS ATRIBUTOS DE OPENGL. (*Cont.*)

<i>Función</i>	<i>Descripción</i>
<code>glEnable</code>	Activa el suavizado con <code>GL_POINT_SMOOTH</code> , <code>GL_LINE_SMOOTH</code> , o <code>GL_POLYGON_SMOOTH</code> . (También se necesita activar el fundido de color.)
<code>glGet**</code>	Varias funciones de consulta, que requieren la especificación del tipo de datos, el nombre simbólico de un parámetro de estado y un puntero a una matriz.
<code>glPushAttrib</code>	Guarda todos los parámetros de estado de un grupo de atributos especificado.
<code>glPopAttrib ( ) ;</code>	Rehabilita todos los valores de los parámetros de estado que se guardaron la última vez.

## REFERENCIAS

Las técnicas de relleno se muestran en Fishkin y Barsky (1984). Las técnicas de suavizado se estudian en Pitteway y Watkinson (1980), Crow (1981), Turkowski (1982), Fujimoto e Iwata (1983), Korein y Badler (1983), Kirk y Arvo (1991), y Wu (1991). Las aplicaciones de la escala de grises se muestran en Crow (1978). Otros estudios sobre los atributos y parámetros de estado se encuentran disponibles en Glassner (1990), Arvo (1991), Kirk (1992), Heckbert (1994) y Paeth (1995).

Se pueden encontrar ejemplos de programación utilizando las funciones de atributos de OpenGL en Woo, Neider, Davis y Shreiner (1999). Una lista completa de las funciones de atributos de OpenGL se encuentra disponible en Shreiner (2000). Los atributos de los caracteres de GLUT se estudian en Kilgard (1996).

## EJERCICIOS

- 4.1 Utilice la función `glutSetColor` para establecer una tabla de color para un conjunto de entrada de valores de color.
- 4.2 Utilizando matrices de vértices y de color, establezca la descripción de una escena que contenga al menos seis objetos bidimensionales.
- 4.3 Escriba un programa para visualizar la descripción de la escena bidimensional del ejercicio anterior.
- 4.4 Utilizando matrices de vértices y de color, establezca la descripción de una escena que contenga al menos cuatro objetos tridimensionales.
- 4.5 Escriba un programa para visualizar una escena en escala de grises con «nubes», en la que las formas de las nubes se deban describir como patrones de puntos sobre un fondo de cielo azul. Las regiones claras y oscuras de las nubes se deben modelar utilizando puntos de diversos tamaños y espaciado entre puntos. (Por ejemplo, una región muy clara se puede modelar con puntos pequeños, ampliamente espaciados y de color gris claro. De forma similar, una región oscura se puede modelar con puntos grandes, más próximos y de color gris oscuro.)
- 4.6 Modifique el programa del ejercicio anterior para visualizar las nubes con patrones de color rojo y amarillo como podrían verse al amanecer o al atardecer. Para lograr un efecto realista, utilice diferentes tonos de rojo y amarillo (y tal vez verde) en los puntos.
- 4.7 Implemente una función general de estilo de línea modificando el algoritmo de dibujo de líneas de Bresenham para representar líneas continuas, a trazos o de puntos.

- 4.8 Implemente una función de estilo de línea utilizando un algoritmo de línea de punto medio para representar líneas continuas, a trazos o de puntos.
- 4.9 Idee un método paralelo para implementar una función de estilo de línea.
- 4.10 Idee un método paralelo para implementar una función de grosor de línea.
- 4.11 Una línea especificada por dos puntos extremos y un grosor se puede convertir en un polígono rectangular con cuatro vértices y, a continuación, se puede visualizar utilizando un método de línea de barrido. Desarrolle un algoritmo eficiente para calcular con una computadora los cuatro vértices que se necesitan para definir tal rectángulo, con los puntos extremos de la línea y el grosor de la línea como argumentos de entrada.
- 4.12 Implemente una función de grosor de línea de un programa de dibujo de líneas para que se pueda visualizar con uno de tres grosores de línea.
- 4.13 Escriba un programa para generar un gráfico de líneas de tres conjuntos de datos definidos sobre el mismo rango de coordenadas del eje  $x$ . La entrada del programa la constituyen los tres conjuntos de datos y las etiquetas del gráfico. Los conjuntos de datos se deben redimensionar para que queden ajustados dentro de un rango de coordenadas definido en la ventana de visualización. Cada conjunto de datos se debe dibujar con un estilo diferente de línea.
- 4.14 Modifique el programa del ejercicio anterior para dibujar los tres conjuntos de datos con colores diferentes, así como con diferentes estilos de línea.
- 4.15 Establezca un algoritmo de visualización de líneas gruesas con extremos abruptos, redondeados o cuadrados. Estas opciones se pueden proporcionar con un menú de opciones.
- 4.16 Idee un algoritmo de visualización de polilíneas gruesas con una unión en punta, una unión redondeada o una unión biselada. Estas opciones se pueden proporcionar con un menú de opciones.
- 4.17 Modifique los fragmentos de código de la Sección 4.8 para mostrar gráficos de datos de líneas, para que el argumento de grosor de línea se pase al procedimiento `linePlot`.
- 4.18 Modifique los fragmentos de código de la Sección 4.8 para mostrar gráficos de datos de líneas, para que el argumento de estilo de línea se pase al procedimiento `linePlot`.
- 4.19 Complete el programa de la Sección 4.8 para mostrar gráficos de líneas utilizando como datos de entrada los procedentes de un archivo de datos.
- 4.20 Complete el programa de la Sección 4.8 para mostrar gráficos de líneas utilizando como datos de entrada los procedentes de un archivo de datos. Además, el programa debe proporcionar etiquetado de los ejes y de las coordenadas del área de visualización de la pantalla. Los conjuntos de datos se deben redimensionar para que se ajusten al rango de coordenadas de la ventana de visualización y cada línea se debe mostrar con un estilo de línea, grosor y color diferentes.
- 4.21 Implemente un menú de opciones de plumilla y brocha para un procedimiento de dibujo de líneas, que incluya al menos dos opciones: forma redonda y forma cuadrada.
- 4.22 Modifique un algoritmo de dibujo de líneas para que la intensidad de la línea de salida se modifique según su pendiente. Es decir, ajustando las intensidades de los píxeles según el valor de la pendiente; todas las líneas se visualizan con la misma intensidad por unidad de longitud.
- 4.23 Defina e implemente una función para controlar el estilo de línea (continua, a trazos, de puntos) de las elipses que se representen.
- 4.24 Defina e implemente una función para cambiar el grosor de las elipses que se representen.
- 4.25 Escriba una subrutina para mostrar un gráfico de barras en un área especificada de la pantalla. La entrada debe incluir el conjunto de datos, el etiquetado de los ejes de coordenadas y las coordenadas del área de pantalla. El conjunto de datos se debe redimensionar para que se ajuste al área designada de pantalla, y las barras se deben mostrar con los colores o patrones designados.
- 4.26 Escriba un programa para mostrar dos conjuntos de datos definidos sobre el mismo rango de coordenadas del eje  $x$ , con los valores de los datos cambiados de escala para que se ajusten a una región especificada de la pantalla de visualización. Las barras de un conjunto de datos se deben desplazar horizontalmente para producir un patrón de superposición de barra que facilite la comparación de los dos conjuntos de datos. Utilice un color o patrón de relleno diferente para los dos conjuntos de barras.

- 4.27 Idee un algoritmo para implementar una tabla de búsqueda de color.
- 4.28 Suponga que dispone de un sistema de pantalla de video de 8 pulgadas por 10 pulgadas que puede mostrar 100 píxeles por pulgada. Si se utiliza en este sistema una tabla de búsqueda de color con 64 posiciones, ¿cuál es el menor tamaño posible (en bytes) del búfer de imagen?
- 4.29 Considere un sistema digitalizado RGB que tiene un búfer de imagen de 512 por 512 píxeles con 20 bits por píxel y una tabla de búsqueda de color con 24 bits por píxel. (a) ¿Cuántos niveles de gris distintos se pueden visualizar en este sistema? (b) ¿Cuántos colores distintos (incluidos los niveles de gris) se pueden visualizar? (c) ¿Cuántos colores se pueden visualizar en cualquier momento? (d) ¿Cuál es el tamaño total de la memoria? (e) Explique dos métodos para reducir el tamaño de la memoria manteniendo las mismas capacidades de color.
- 4.30 Modifique el algoritmo de líneas de barrido para aplicar cualquier patrón de relleno rectangular especificado al interior de un polígono, comenzando por una posición de patrón designada.
- 4.31 Escriba un programa para convertir el interior de una elipse especificada en un color sólido.
- 4.32 Escriba un procedimiento para llenar el interior de una elipse con un patrón especificado.
- 4.33 Escriba un procedimiento para llenar el interior de cualquier conjunto especificado de vértices de un área de relleno, incluido uno que tenga aristas que se cruzan, utilizando la regla del número de vueltas distinto de cero para identificar las regiones interiores.
- 4.34 Modifique el algoritmo de relleno por contorno para una región 4-conectada para evitar un apilado excesivo incorporando métodos de línea de barrido.
- 4.35 Escriba un procedimiento de relleno por contorno para llenar una región 8-conectada.
- 4.36 Explique cómo una elipse visualizada con el método del punto medio se podría llenar adecuadamente con un algoritmo de relleno por contorno.
- 4.37 Desarrolle e implemente un algoritmo de relleno por inundación para llenar el interior de un área especificada.
- 4.38 Defina e implemente un procedimiento para cambiar el tamaño del patrón existente de relleno rectangular.
- 4.39 Escriba un procedimiento para implementar un algoritmo de relleno suave. Defina con cuidado qué debe realizar el algoritmo y cómo se deben combinar los colores.
- 4.40 Idee un algoritmo para ajustar la altura y la anchura de los caracteres definidos como patrones de cuadricula rectangular.
- 4.41 Implemente subrutinas para establecer el vector de orientación de caracteres y la trayectoria del texto para controlar la visualización de cadenas de caracteres.
- 4.42 Escriba un programa para alinear texto como se indique de acuerdo con los valores de entrada de los parámetros de alineación.
- 4.43 Desarrolle procedimientos para implementar atributos de marcadores (tamaño y color).
- 4.44 Implemente un procedimiento de suavizado ampliando el algoritmo de línea de Bresenham, para ajustar las intensidades de los píxeles en la vecindad de la trayectoria de una línea.
- 4.45 Implemente un procedimiento de suavizado para el algoritmo de la línea del punto medio.
- 4.46 Desarrolle un algoritmo de suavizado para límites elípticos.
- 4.47 Modifique el algoritmo de línea de barrido de relleno de áreas para incorporar suavizado. Utilice técnicas de coherencia para reducir los cálculos en las líneas sucesivas de barrido.
- 4.48 Escriba un programa para implementar el algoritmo de suavizado de Pitteway-Watkinson como un procedimiento de línea de barrido para llenar el interior de un polígono, utilizando la función de dibujo de puntos de OpenGL.

## CAPÍTULO 5

# Transformaciones geométricas



Una escena de gráficos por computadora que contiene una parte de un paisaje del Movimiento Browniano y el reflejo de la luna en el agua . (Cortesía de Ken Musgrave y Benoit B. Mandelbrot, Mathematics and Computer Sciencie, Universidad de Yale).

<b>5.1</b>	Transformaciones geométricas bidimensionales básicas	<b>5.9</b>	Transformaciones geométricas en un espacio tridimensional
<b>5.2</b>	Representación matricial y coordenadas homogéneas	<b>5.10</b>	Translaciones tridimensionales
<b>5.3</b>	Transformaciones inversas	<b>5.11</b>	Rotaciones tridimensionales
<b>5.4</b>	Transformaciones bidimensionales compuestas	<b>5.12</b>	Escalado tridimensional
<b>5.5</b>	Otras transformaciones bidimensionales	<b>5.13</b>	Transformaciones tridimensionales compuestas
<b>5.6</b>	Métodos de rasterización para transformaciones geométricas	<b>5.14</b>	Otras transformaciones tridimensionales
<b>5.7</b>	Transformaciones de rasterización en OpenGL	<b>5.15</b>	Transformaciones entre sistemas de coordenadas tridimensionales
<b>5.8</b>	Transformaciones entre sistemas de coordenadas bidimensionales	<b>5.16</b>	Transformaciones afines
		<b>5.17</b>	Funciones de transformaciones geométricas en OpenGL
		<b>5.18</b>	Resumen

Hasta ahora, hemos visto cómo podemos describir una escena en términos de primitivas gráficas, tales como una línea de segmentos y áreas completas, y los atributos asociados a dichas primitivas. Y hemos explorado los algoritmos de rastreo de líneas para mostrar primitivas de salida en un dispositivo de rastreo. Ahora, echaremos un vistazo a las operaciones de transformación que se pueden aplicar a objetos para recolocarlos o darlos un tamaño diferente. Estas operaciones también son usadas en la visualización de rutinas que convierten una descripción de una escena de coordenadas universales en un despliegue para un dispositivo de salida. Además, son usados en variedad de otras aplicaciones, tales como diseño de ayuda y animación por computador. Un arquitecto, por ejemplo, crea un esquema/plano ordenando la orientación y el tamaño de las partes que componen el diseño, y un animador por computadora desarrolla una secuencia de vídeo moviendo la posición de la «cámara» o los objetos en la escena a lo largo de caminos específicos. Las operaciones que se aplican a descripciones geométricas de un objeto para cambiar su posición, orientación o tamaño se llaman **transformaciones geométricas**.

A veces las operaciones sobre transformaciones geométricas también se llaman *transformaciones de modelado*, pero algunos paquetes gráficos hacen distinción entre los dos términos. En general, las transformaciones de modelado se usan para construir una escena o para dar una descripción jerárquica de un objeto complejo que está compuesto por distintas partes, las cuales a su vez pueden estar compuestas por partes más simples y así sucesivamente. Como ejemplo, un avión se compone de alas, cola, fuselaje, motor y otros componentes, cada uno de los cuales puede ser especificado en términos de componentes de segundo nivel, y así sucesivamente, bajando en la jerarquía de partes de los componentes. De este modo, el avión puede ser descrito en términos de dichos componentes y una transformación de «modelo» asociado para cada uno que describe cómo ese componente va a encajar dentro del diseño total del avión.

Las transformaciones geométricas, por otro lado, pueden usarse para describir cómo los objetos deben moverse a lo largo de una escena durante una secuencia de animación o simplemente, para verlos desde otro ángulo. Por tanto, algunos paquetes gráficos ofrecen dos juegos de rutinas de transformación, mientras otros paquetes tienen un único juego de funciones que pueden ser usadas tanto por transformaciones geométricas como por transformaciones de modelado.

## 5.1 TRANSFORMACIONES GEOMÉTRICAS BIDIMENSIONALES BÁSICAS

Las funciones de transformaciones geométricas que se pueden encontrar en todos los paquetes gráficos, son aquellas que se usan para la traslación, la rotación y el cambio de escala. Otras rutinas de transformaciones útiles, que a veces se incluyen en los paquetes, son las operaciones de reflexión e inclinación. Para introducir los conceptos generales asociados a las transformaciones geométricas, se van a considerar en primer lugar, las operaciones en dos dimensiones, y después se discutirá cómo las ideas básicas pueden extenderse a escenas tridimensionales. Una vez que se hayan comprendido los conceptos básicos, se podrán escribir fácilmente rutinas para representar transformaciones geométricas de objetos en escenas bidimensionales.

### Traslaciones bidimensionales

Se realiza una **traslación** de un punto sencillo de coordenadas, mediante la inclusión de compensaciones en sus propias coordenadas, para generar una nueva posición de coordenadas. En efecto, se está moviendo la posición del punto original a lo largo de una trayectoria en línea recta hacia su nueva localización. De modo similar, una traslación es aplicable a un objeto que se define con múltiples posiciones de coordenadas, tales como cuadriláteros, mediante la recolocación de todas las posiciones de sus coordenadas, usando el mismo desplazamiento a lo largo de trayectorias paralelas. Así, el objeto completo se muestra en la nueva localización.

Para trasladar una posición bidimensional, añadimos **distancias de traslación**  $t_x$  y  $t_y$  a las coordenadas originales  $(x, y)$  para obtener la nueva posición de coordenadas  $(x', y')$  como se muestra en la Figura 5.1.

$$x' = x + t_x \quad y' = y + t_y \quad (5.1)$$

El par de distancia de traslación  $(t_x, t_y)$  se llama **vector de traslación** o **vector de cambio**.

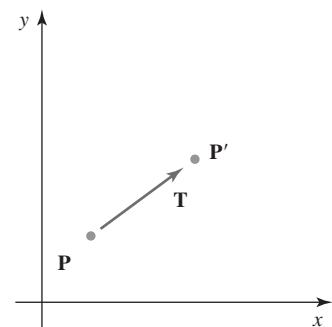
Podemos expresar las Ecuaciones de traslación 5.1 como una única ecuación de una matriz, usando los siguientes vectores columna para representar posiciones de coordenadas y el vector de traslación.

$$\mathbf{P} = \begin{bmatrix} x \\ y \end{bmatrix}, \quad \mathbf{P}' = \begin{bmatrix} x' \\ y' \end{bmatrix}, \quad \mathbf{T} = \begin{bmatrix} t_x \\ t_y \end{bmatrix} \quad (5.2)$$

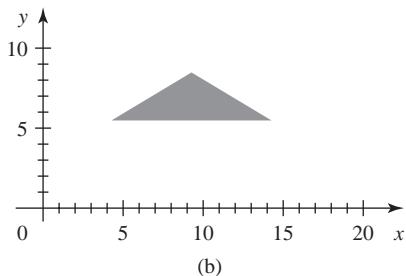
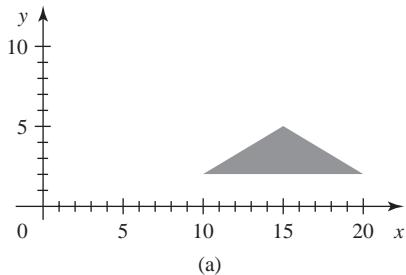
Esto nos permite escribir las ecuaciones de traslación bidimensionales en forma de matriz.

$$\mathbf{P}' = \mathbf{P} + \mathbf{T} \quad (5.3)$$

La traslación es un tipo de *transformación de sólido-rígido* que mueve objetos sin deformarlos. Esto es, cada punto de un objeto es trasladado en la misma medida. Un segmento en línea recta es trasladado mediante la aplicación de una ecuación de transformación a cada uno de los puntos finales de la línea y redibujando la línea entre los dos nuevos puntos finales. Un polígono se traslada de forma similar. Se añade un vector de



**FIGURA 5.1.** Traslación de un punto desde la posición  $\mathbf{P}$  a la posición  $\mathbf{P}'$  usando un vector de traslación  $\mathbf{T}$ .



**FIGURA 5.2.** Movimiento de un polígono desde la posición (a) a la posición (b) con el vector de traslación  $(-5.50, 3.75)$ .

traslación a la posición de las coordenadas para cada vértice y después se regenera el polígono usando un nuevo conjunto de coordenadas de vértices. La Figura 5.2 ilustra la aplicación del vector de traslación especificado para mover un objeto de una posición a otra.

La siguiente rutina ilustra las operaciones de traslación. Un vector de traslación de entrada se usa para mover los vértices de un polígono desde una posición de un universo de coordenadas a otro, y las rutinas de OpenGL se usan para regenerar el polígono trasladado.

```

class wcPt2D {
public:
    GLfloat x, y;
};

void translatePolygon (wcPt2D * verts, GLint nVerts, GLfloat tx, GLfloat ty)
{
    GLint k;

    for (k = 0; k < nVerts; k++) {
        verts [k].x = verts [k].x + tx;
        verts [k].y = verts [k].y + ty;
    }
    glBegin (GL_POLYGON);
    for (k = 0; k < nVerts; k++)
        glVertex2f (verts [k].x, verts [k].y);
    glEnd ( );
}

```

Si se desea borrar el polígono original, se puede mostrar con un color de fondo antes de trasladarlo. En algunos paquetes gráficos hay disponibles otros métodos para borrar componentes de dibujo. También, si se desea guardar la posición del polígono original, se pueden almacenar las posiciones trasladadas en un registro diferente.

Para trasladar otros objetos se usan métodos similares. Para cambiar la posición de un círculo o una elipse, se puede trasladar el centro de coordenadas y redibujar la figura en la nueva localización. Para una curva *spline*, se trasladan los puntos que definen la trayectoria de la curva y después se reconstruyen las secciones de la curva entre las nuevas posiciones de coordenadas.

## Rotaciones bidimensionales

Se genera una transformación de **rotación** de un objeto mediante la especificación de un **eje de rotación** y un **ángulo de rotación**. Todos los puntos del objeto son entonces transformados a la nueva posición, mediante la rotación de puntos con el ángulo especificado sobre el eje de rotación.

Una rotación bidimensional de un objeto se obtiene mediante la recolocación del objeto a lo largo de una trayectoria circular sobre el plano *xy*. En este caso, se está rotando el objeto sobre un eje de rotación que es perpendicular al plano (paralelo al eje de coordenadas *z*). Los parámetros para la rotación bidimensional son el ángulo de rotación  $\theta$ , y una posición  $(x_r, y_r)$  llamada **punto de rotación** (o **punto de pivote**) sobre los cuales el objeto va a ser rotado (Figura 5.3). El punto de pivote es la posición de intersección entre el eje de coordenadas y el plano *xy*. Un valor positivo para el ángulo  $\theta$  define una rotación en sentido contrario a las agujas del reloj sobre el punto de pivote, como en la Figura 5.3, y un valor negativo rota objetos en el sentido de las agujas del reloj.

Para simplificar la explicación del método básico, primero hay que determinar las ecuaciones de transformación para la rotación de un punto de posición **P**, cuando el punto de pivote está en el origen de coordenadas. La relación entre el angular y las coordenadas de las posiciones originales y transformadas se muestra en la Figura 5.4. En esta figura,  $r$  es la distancia constante del punto respecto del origen, el ángulo  $\phi$  es la posición angular original del punto desde la horizontal, y  $\theta$  es el ángulo de rotación. Usando identidades trigonométricas estándar, podemos expresar las coordenadas transformadas en función de los ángulos  $\theta$  y  $\phi$  como:

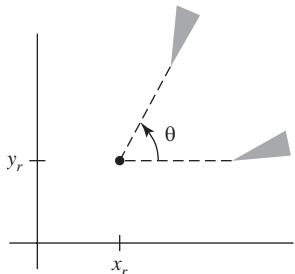
$$\begin{aligned} x' &= r \cos(\phi + \theta) = r \cos\phi \cos\theta - r \sin\phi \sin\theta \\ y' &= r \sin(\phi + \theta) = r \cos\phi \sin\theta - r \sin\phi \cos\theta \end{aligned} \quad (5.4)$$

Las coordenadas originales del punto en coordenadas polares son:

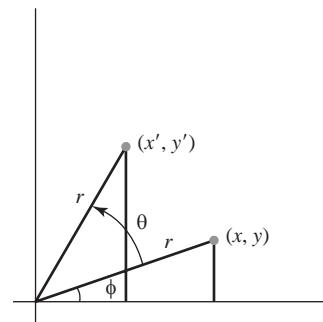
$$x = r \cos\phi, \quad y = r \sin\phi \quad (5.5)$$

Sustituyendo las expresiones de 5.5 en la Ecuación 5.4, obtenemos las ecuaciones de transformación para rotar la posición de un punto  $(x, y)$  aplicando un ángulo  $\theta$  sobre el origen:

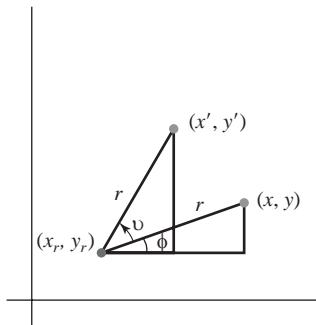
$$\begin{aligned} x' &= x \cos\theta - y \sin\theta \\ y' &= x \sin\theta - y \cos\theta \end{aligned} \quad (5.6)$$



**FIGURA 5.3.** Rotación de un objeto un ángulo  $\theta$  alrededor del punto de pivote  $(x_r, y_r)$ .



**FIGURA 5.4.** Rotación de un punto desde la posición  $(x, y)$  a la posición  $(x', y')$  un ángulo  $\theta$  respecto del origen de coordenadas. El desplazamiento angular original del punto respecto del eje *x* es  $\phi$ .



**FIGURA 5.5.** Rotación de un punto desde la posición  $(x, y)$  a la posición  $(x', y')$  un ángulo  $\theta$  respecto al punto de rotación  $(x_r, y_r)$ .

Con las representaciones del vector columna 5.2, para posiciones de coordenadas, podemos escribir las ecuaciones de rotación en forma de matriz

$$\mathbf{P}' = \mathbf{R} \cdot \mathbf{P} \quad (5.7)$$

donde la matriz de rotación es:

$$\mathbf{R} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \quad (5.8)$$

Una representación del vector columna para una posición de coordenadas  $\mathbf{P}$  como en las Ecuaciones 5.2, es una notación matemática estándar. En cualquier caso, los primeros sistemas gráficos a veces usaban una representación de vector-fila para posiciones de puntos. Esto cambia el orden en el que la matriz de multiplicación para una rotación sería representada. Pero ahora, todos los paquetes gráficos como OpenGL, Java, PHIGS y GKS siguen los convenios del estándar vector-columna.

La rotación de un punto sobre una posición de pivote arbitraria se ilustra en la Figura 5.5. Usando las relaciones trigonométricas indicadas por los dos triángulos rectángulos de esta figura, se pueden generalizar las Ecuaciones 5.6 para obtener las ecuaciones de transformación para la rotación de un punto sobre cualquier posición de rotación específica  $(x_r, y_r)$ :

$$\begin{aligned} x' &= x_r + (x - x_r) \cos \theta - (y - y_r) \sin \theta \\ y' &= y_r + (x - x_r) \sin \theta + (y - y_r) \cos \theta \end{aligned} \quad (5.9)$$

Estas ecuaciones de rotación generales difieren de las Ecuaciones 5.6, por la inclusión de términos aditivos, así como factores multiplicativos en los valores de coordenadas. La expresión de la matriz 5.7 puede modificarse para incluir las coordenadas pivote añadiendo la matriz de vector columna, cuyos elementos contienen los términos aditivos (traslacionales) de las Ecuaciones 5.9. De todos modos, hay mejores maneras de formular dichas ecuaciones matriciales, por lo que en la Sección 5.2 se expone un esquema más consistente para representar ecuaciones de transformación.

Al igual que con las traslaciones, las rotaciones son transformaciones de sólido-rígido que mueven objetos sin deformarlos. Cada punto de un objeto se rota un mismo ángulo. Un segmento en línea recta se rota mediante la aplicación de las ecuaciones de rotación 5.9 a cada uno de sus puntos finales o extremos y redibujando luego la línea entre los nuevos extremos. Un polígono se rota desplazando cada uno de sus vértices usando el ángulo de rotación especificado y después regenerando el polígono usando los nuevos vértices. Rotamos una curva reposicionando los puntos de definición para la curva y redibujándola después. Un círculo o una elipse, por ejemplo, pueden rotarse sobre un punto de pivote no centrado, moviendo la posición del centro a través del arco que sustenta el ángulo de rotación especificado. Y podemos rotar una elipse sobre su propio centro de coordenadas, sencillamente rotando el eje mayor y el eje menor.

En el siguiente código de ejemplo, se rota un polígono sobre un punto de pivote de un universo de coordenadas especificado. Los parámetros de entrada para el procedimiento de rotación son los vértices origina-

les del polígono, las coordenadas del punto de pivote y el ángulo de rotación `theta` especificado en radianes. Siguiendo la transformación de la posición de los vértices, el polígono se regenera usando rutinas OpenGL.

```

class wcPt2D {
public:
    GLfloat x, y;
};

void rotatePolygon (wcPt2D * verts, GLint nVerts, wcPt2D pivPt, GLdouble theta)
{
    wcPt2D * vertsRot;
    GLint k;

    for (k = 0; k < nVerts; k++) {
        vertsRot [k].x = pivPt.x + (verts [k].x - pivPt.x) * cos (theta)
                        - (verts [k].y - pivPt.y) * sin (theta);
        vertsRot [k].y = pivPt.y + (verts [k].x - pivPt.x) * sin (theta)
                        + (verts [k].y - pivPt.y) * cos (theta);
    }
    glBegin {GL_POLYGON};
    for (k = 0; k < nVerts; k++)
        glVertex2f (vertsRot [k].x, vertsRot [k].y);
    glEnd ( );
}

```

## Cambio de escala bidimensional

Para alterar el tamaño de un objeto, aplicamos transformaciones **de escala**. Una simple operación de cambio de escala bidimensional se lleva a cabo multiplicando las posiciones de los objetos ( $x, y$ ) por los **fatores de escala**  $s_x$  y  $s_y$  para producir las coordenadas transformadas ( $x', y'$ ):

$$x' = x \cdot s_x, \quad y' = y \cdot s_y \quad (5.10)$$

El factor de escala  $s_x$  cambia la escala de un objeto en la dirección  $x$ , mientras que  $s_y$  hace el cambio de escala en la dirección  $y$ . Las ecuaciones básicas del cambio de escala en dos dimensiones 5.10 pueden también escribirse en la forma de la matriz siguiente.

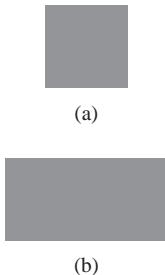
$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} \quad (5.11)$$

o,

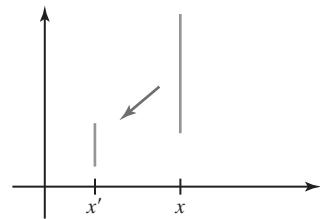
$$\mathbf{P}' = \mathbf{S} \cdot \mathbf{P} \quad (5.12)$$

donde  $\mathbf{S}$  es la matriz 2 por 2 de cambio de escala en la Ecuación 5.11.

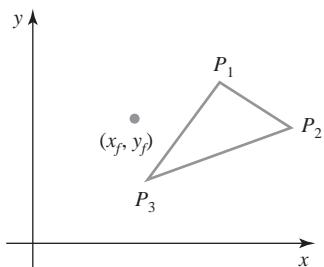
Cualquier valor positivo puede ser asignado a los valores de escala  $s_x$  y  $s_y$ . Valores inferiores a 1 reducen el tamaño de los objetos; valores superiores a 1 producen alargamientos. Especificando un valor de 1 tanto para  $s_x$  como para  $s_y$  se deja el tamaño del objeto inalterado. Cuando a  $s_x$  y  $s_y$  se les asigna el mismo valor, se produce un **cambio de escala uniforme** que mantiene las proporciones relativas del objeto. Valores desiguales de  $s_x$  y  $s_y$  resultan en un **cambio de escala diferente** que es a menudo usado en aplicaciones de diseño, donde los dibujos son construidos desde unas pocas formas básicas que pueden ajustarse mediante escalas y



**FIGURA 5.6.** Conversión de un cuadrado (a) en un rectángulo (b) mediante los factores de escala  $s_x = 2$  y  $s_y = 1$ .



**FIGURA 5.7.** Cambio de escala de una línea aplicando la Ecuación 5.12 con  $s_x = s_y = 0.5$ , reduciéndose su tamaño y aproximándose al origen de coordenadas.



**FIGURA 5.8.** Cambio de escala respecto a un punto fijo seleccionado  $(x_f, y_f)$ . La distancia desde cada vértice del polígono al punto fijo se escala mediante las Ecuaciones de transformación 5.13.

transformaciones posicionales (Figura 5.6). En algunos sistemas, los valores negativos también pueden especificarse mediante parámetros de escala. Ello no sólo le da un nuevo tamaño al objeto, además lo refleja sobre uno o más ejes de coordenadas.

Los objetos transformados con la Ecuación 5.11 son tanto escalables como reubicables. Los factores de escala con valores absolutos inferiores a 1 mueven los objetos aproximándolos al origen, mientras que valores absolutos mayores que 1 mueven la posición de las coordenadas alejándolas del origen. La Figura 5.7 ilustra el cambio de escala de una línea asignando el valor 0.5 a  $s_x$  y  $s_y$  en la Ecuación 5.11. Tanto la línea de longitud como la distancia desde el origen se reducen en un factor de  $\frac{1}{2}$ .

Podemos controlar la localización de un objeto cambiado de escala eligiendo una posición, llamada **punto fijo**, que debe permanecer sin cambios después de la transformación de escala. Las coordenadas para el punto fijo,  $(x_f, y_f)$  son a menudo elegidas de la posición de algún objeto, tal como su centroide (Apéndice A), aunque puede elegirse cualquier otra posición espacial. A los objetos se les da ahora otro tamaño mediante el cambio de escala de las distancias entre los puntos de los objetos y el punto fijo (Figura 5.8). Para la posición de coordenadas  $(x, y)$  las coordenadas de escala  $(x', y')$  se calculan a partir de las siguientes relaciones.

$$x' - x_f = (x - x_f) s_x, \quad y' - y_f = (y - y_f) s_y \quad (5.13)$$

Podemos escribir las Ecuaciones 5.13 para separar los términos multiplicativo y aditivo como:

$$x' = x \cdot s_x + x_f(1 - s_x) \quad (5.14)$$

$$y' = y \cdot s_y + y_f(1 - s_y)$$

donde los términos aditivos  $x_f(1 - s_x)$  e  $y_f(1 - s_y)$  son constantes para todos los puntos del objeto.

Incluir las coordenadas para un punto fijo en las ecuaciones de escala es similar a incluir coordenadas para un punto de pivote en ecuaciones de rotación. Podemos configurar un vector columna cuyos elementos sean términos constantes en las Ecuaciones 5.14 y después sumar este vector columna al producto  $\mathbf{S} \cdot \mathbf{P}$  en la Ecuación 5.12. En la próxima sección veremos la formulación de una matriz para las ecuaciones de transformación que implican sólo matrices de multiplicación.

Los polígonos cambian de escala mediante la aplicación de las Ecuaciones de transformación 5.14 a cada vértice, regenerando después el polígono usando los vértices transformados. Para otros objetos, aplicamos las ecuaciones de transformación de escala a los parámetros que definen el objeto. Para cambiar el tamaño de un círculo, podemos reducir su radio y calcular las nuevas posiciones de las coordenadas del contorno de la circunferencia. Y para cambiar el tamaño de una elipse, aplicamos el escalado de los parámetros sobre sus ejes para luego trazar la nueva posición de la elipse sobre su centro de coordenadas.

El siguiente procedimiento ilustra una aplicación de los cálculos de cambio de escala para un polígono. Las coordenadas para los vértices del polígono y para el punto fijo son parámetros de entrada, junto con los factores de escala. Después de realizar las transformaciones de coordenadas, se usan las rutinas OpenGL para generar el polígono cambiado de escala.

---

```

class wcPt2D {
public:
    GLfloat x, y;
};

void scalePolygon (wcPt2D * verts, GLint nVerts, wcPt2D fixedPt,
                   GLfloat sx, GLfloat sy)
{
    wcPt2D vertsNew;
    GLint k;

    for (k = 0; k < nVerts; k++) {
        vertsNew [k].x = verts [k].x * sx + fixedPt.x * (1 - sx);
        vertsNew [k].y = verts [k].y * sy + fixedPt.y * (1 - sy);
    }
    glBegin (GL_POLYGON);
    for (k = 0; k < nVerts; k++)
        glVertex2f (vertsNew [k].x, vertsNew [k].y);
    glEnd ( );
}

```

---

## 5.2 REPRESENTACIÓN MATRICIAL Y COORDENADAS HOMOGÉNEAS

---

Muchas aplicaciones gráficas implican secuencias de transformaciones geométricas. Una animación debería requerir que un objeto fuese trasladado y rotado tras cada incremento de movimiento. En diseño y aplicaciones de construcción de dibujos, se llevan a cabo traslaciones, rotaciones y cambios de escala para acoplar los componentes del dibujo dentro de sus propias posiciones. Y la visualización de las transformaciones implica secuencias de traslaciones y rotaciones para llevarnos desde la escena original especificada a la visualización en un dispositivo de salida. Aquí, consideramos cómo las representaciones de matrices discutidas en la sección anterior pueden reformularse, de tal forma que las secuencias de transformaciones puedan ser procesadas eficientemente.

Hemos visto en la Sección 5.1 que cada una de las tres transformaciones bidimensionales básicas (traslación, rotación y cambio de escala) pueden expresarse en forma de matriz general:

$$\mathbf{P}' = \mathbf{M}_1 \cdot \mathbf{P} + \mathbf{M}_2 \quad (5.15)$$

con posiciones de coordenadas  $\mathbf{P}$  y  $\mathbf{P}'$  representados en vectores columnas. La matriz  $\mathbf{M}_1$  es una matriz de 2 por 2 que contiene factores multiplicativos, y  $\mathbf{M}_2$  es una matriz columna de 2 elementos que contiene los tér-

minos traslacionales. Para la translación,  $\mathbf{M}_1$  es la matriz identidad. Para la rotación o el cambio de escala,  $\mathbf{M}_2$  contiene los términos traslacionales asociados con el punto de pivote o con el punto fijo de escalado. Para producir una secuencia de transformaciones con esas ecuaciones, como por ejemplo, un cambio de escala seguido de una rotación y luego una translación, podemos calcular las coordenadas transformadas haciendo una cosa cada vez. Primero, se cambia la escala de la posición de las coordenadas, luego dichas coordenadas se giran y, finalmente, las coordenadas rotadas son trasladadas. Sin embargo, una forma más eficiente de hacerlo, es combinar transformaciones de tal suerte que la posición final de las coordenadas se obtenga directamente a partir de las coordenadas iniciales, sin calcular valores de coordenadas intermedios. Podemos hacer esto, reformulando la Ecuación 5.15 para eliminar la operación de suma de matrices.

## Coordenadas homogéneas

Los términos multiplicativos y traslacionales para una transformación geométrica bidimensional pueden ser combinados dentro de una matriz sencilla, si expandimos la representación a matrices de 3 por 3. En ese caso, podemos usar la tercera columna de la matriz de transformación para los términos traslacionales, y todas las ecuaciones de transformación pueden expresarse como multiplicación de matrices. Pero para poder hacer esto, necesitamos además expandir la representación matricial para posiciones de coordenadas bidimensionales a una matriz columna de 3 elementos. Una técnica estándar para lograr esto consiste en expandir cada representación de posición-coordenada bidimensional  $(x, y)$  en representaciones de 3 elementos  $(x_h, y_h, h)$  llamadas **coordenadas homogéneas**, donde el **parámetro homogéneo**  $h$  es un valor distinto de cero tal que:

$$x = \frac{x_h}{h}, \quad y = \frac{y_h}{h} \quad (5.16)$$

Por tanto, una representación de coordenadas homogéneas bidimensionales, puede escribirse también como  $(h \cdot x, h \cdot y, h)$ . Para transformaciones geométricas, podemos elegir el parámetro homogéneo  $h$  para que sea cualquier valor distinto de cero. Así, hay un número infinito de representaciones homogéneas equivalentes para cada punto de coordenadas  $(x, y)$ . Una elección acertada es fijar  $h = 1$ . Cada posición bidimensional se representa con coordenadas homogéneas  $(x, y, 1)$ . Se necesitan otros valores para el parámetro  $h$ , por ejemplo en formulaciones de matrices para mostrar transformaciones tridimensionales.

El término *coordenadas homogéneas* se usa en matemáticas para referirse al efecto de esta representación en coordenadas cartesianas. Cuando un punto cartesiano  $(x, y)$  se convierte a representación homogénea  $(x_h, y_h, h)$  las ecuaciones que contienen  $x$  e  $y$ , tales como  $f(x, y) = 0$ , se convierten en ecuaciones homogéneas en los tres parámetros  $x_h, y_h$ , y  $h$ . Esto significa precisamente, que si cada uno de los tres parámetros es sustituido por cualquier valor,  $v$  veces, dicho valor  $v$  puede ser despejado de la ecuación.

Expresar posiciones en coordenadas homogéneas nos permite representar todas las ecuaciones de transformaciones geométricas como multiplicación de matrices, que es el método estándar usado en los sistemas gráficos. Las posiciones de coordenadas bidimensionales se representan con vectores columna de tres elementos, y las operaciones de transformación bidimensionales se representan como matrices de 3 por 3.

## Matriz de translación bidimensional

Usando la aproximación de coordenadas homogéneas, podemos representar las ecuaciones para una translación bidimensional de una posición de coordenadas usando la siguiente matriz de multiplicación.

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad (5.17)$$

Esta operación de translación puede escribirse en su forma abreviada:

$$\mathbf{P}' = \mathbf{T}(t_x, t_y) \cdot \mathbf{P} \quad (5.18)$$

con  $\mathbf{T}(t_x, t_y)$  como la matriz de traslación de 3 por 3 de la Ecuación 5.17. En situaciones donde no hay ambigüedad en los parámetros de traslación, podemos representar sencillamente la matriz de traslación como  $\mathbf{T}$ .

### Matriz de rotación bidimensional

De manera similar, las ecuaciones de transformación de rotación bidimensional sobre el origen de coordenadas pueden expresarse en forma de matriz,

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad (5.19)$$

o como:

$$\mathbf{P}' = \mathbf{R}(\theta) \cdot \mathbf{P} \quad (5.20)$$

El operador de transformación de rotación  $\mathbf{R}(\theta)$  es la matriz de 3 por 3 en la Ecuación 5.19, con el parámetro de rotación  $\theta$ . Podemos además escribir esta matriz de rotación simplemente como  $\mathbf{R}$ .

En algunas bibliotecas gráficas, una función de rotación bidimensional genera sólo rotaciones sobre el eje de coordenadas, como en la Ecuación 5.19. Una rotación sobre cualquier otro punto de pivote debe representarse como una secuencia de operaciones de transformación. Una alternativa en paquetes gráficos es ofrecer parámetros adicionales en la rutina de rotación para las coordenadas del punto de pivote. Una rutina de rotación que incluye parámetros del punto de pivote, luego establece una matriz general de rotación, sin la necesidad de invocar una sucesión de funciones de transformación.

### Matriz de cambio de escala bidimensional

Finalmente, una transformación de cambio de escala relativa al origen de coordenadas puede ahora expresarse como la matriz de multiplicación:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad (5.21)$$

o,

$$\mathbf{P}' = \mathbf{S}(s_x, s_y) \cdot \mathbf{P} \quad (5.22)$$

El operador  $\mathbf{S}(s_x, s_y)$  es la matriz de 3 por 3 en la Ecuación 5.21 con parámetros  $s_x$  y  $s_y$ . Y, en la mayoría de los casos, podemos representar la matriz de cambio de escala simplemente como  $\mathbf{S}$ .

Algunas bibliotecas ofrecen una función de cambio de escala que puede generar sólo un cambio de escala con respecto al origen de coordenadas, como en la Ecuación 5.21. En este caso, una transformación de cambio de escala relativa a otra posición de referencia es llevada a cabo como una sucesión de operaciones de transformación. Sin embargo, otros sistemas sí incluyen una rutina de cambio de escala general que puede construir matrices homogéneas para realizar cambios de escala con respecto a puntos fijos designados.

## 5.3 TRANSFORMACIONES INVERSAS

---

Para la traslación, obtenemos la matriz inversa mediante la negación de las distancias de traslación. Así, si tenemos distancias de traslación bidimensionales  $t_x$  y  $t_y$ , la matriz de traslación inversa es:

$$\mathbf{T}^{-1} = \begin{bmatrix} 1 & 0 & -t_x \\ 0 & 1 & -t_y \\ 0 & 0 & 1 \end{bmatrix} \quad (5.23)$$

Esto produce una traslación en la dirección opuesta, y el producto de la matriz de traslación y su inversa producen la matriz identidad.

Una rotación inversa se obtiene sustituyendo el ángulo de rotación por su negativo. Por ejemplo, una rotación bidimensional a través del ángulo  $q$  sobre el origen de coordenadas, tiene la matriz de transformación:

$$\mathbf{R}^{-1} = \begin{bmatrix} \cos \theta & \sin \theta & 0 \\ -\sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (5.24)$$

Los valores negativos para los ángulos de rotación generan rotaciones en el sentido de las agujas del reloj, así, la matriz identidad se produce cuando alguna matriz de rotación se multiplica por su inversa. Puesto que por el cambio de signo del ángulo de rotación sólo se ve afectada la función seno, la matriz inversa puede obtenerse también intercambiando filas por columnas. Esto es, podemos calcular la inversa de cualquier matriz de rotación  $\mathbf{R}$  evaluando su traspuesta ( $\mathbf{R}^{-1} = \mathbf{R}^T$ ).

Formamos la matriz inversa para cualquier transformación de escala sustituyendo los parámetros de escala por sus recíprocos. Para escalas bidimensionales con parámetros  $s_x$  y  $s_y$  aplicados respecto al origen de coordenadas, la matriz de transformación inversa es:

$$\mathbf{S}^{-1} = \begin{bmatrix} \frac{1}{s_x} & 0 & 0 \\ 0 & \frac{1}{s_y} & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (5.25)$$

La matriz inversa genera una transformación de escala opuesta, de tal forma que la multiplicación de cualquier matriz de escala por su inversa produce la matriz identidad.

## 5.4 TRANSFORMACIONES COMPUESTAS BIDIMENSIONALES

---

Usando la representación de matrices, podemos establecer una secuencia de transformaciones como **matriz de transformación compuesta** calculando el producto de las transformaciones individuales. Formando productos con las matrices de transformación es común referirse a ello como **concatenación**, o **composición**, de matrices. Desde una posición de coordenadas representada como una matriz columna homogénea, debemos premultiplicar la matriz columna por las matrices, representando una secuencia de transformaciones. Y, como muchas posiciones de una escena son normalmente transformadas por la misma secuencia, es más eficiente primero multiplicar la transformación de matrices para formar una única matriz compuesta. Así, si queremos aplicar dos transformaciones a la posición de un punto  $\mathbf{P}$ , la ubicación transformada se calcularía como:

$$\begin{aligned} \mathbf{P}' &= \mathbf{M}_2 \cdot \mathbf{M}_1 \cdot \mathbf{P} \\ &= \mathbf{M} \cdot \mathbf{P} \end{aligned} \quad (5.26)$$

La posición de coordenadas se transforma usando la matriz compuesta  $\mathbf{M}$ , mejor que aplicando las transformaciones individuales  $\mathbf{M}_1$  y luego  $\mathbf{M}_2$ .

## Traslaciones compuestas bidimensionales

Si dos vectores de traslación consecutivos  $(t_{1x}, t_{1y})$  y  $(t_{2x}, t_{2y})$  se aplican a una posición de coordenadas bidimensional  $\mathbf{P}$ , la ubicación transformada final  $\mathbf{P}'$ , se calcula como:

$$\begin{aligned}\mathbf{P}' &= \mathbf{T}(t_{2x}, t_{2y}) \cdot \{\mathbf{T}(t_{1x}, t_{1y}) \cdot \mathbf{P}\} \\ &= \{\mathbf{T}(t_{2x}, t_{2y}) \cdot \mathbf{T}(t_{1x}, t_{1y})\} \cdot \mathbf{P}\end{aligned}\quad (5.27)$$

donde  $\mathbf{P}$  y  $\mathbf{P}'$  se representan como vectores columna de coordenadas homogéneas de tres elementos. Podemos verificar estos resultados, calculando el producto de matrices para los dos agrupamientos asociados. También, la matriz de transformación compuesta para esta secuencia de traslaciones es:

$$\begin{bmatrix} 1 & 0 & t_{2x} \\ 0 & 1 & t_{2y} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & t_{1x} \\ 0 & 1 & t_{1y} \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_{1x} + t_{2x} \\ 0 & 1 & t_{1y} + t_{2y} \\ 0 & 0 & 1 \end{bmatrix} \quad (5.28)$$

o,

$$\mathbf{T}(t_{2x}, t_{2y}) \cdot \mathbf{T}(t_{1x}, t_{1y}) = \mathbf{T}(t_{1x} + t_{2x}, t_{1y} + t_{2y}) \quad (5.29)$$

lo cual demuestra que dos traslaciones sucesivas son aditivas.

## Rotaciones compuestas bidimensionales

Dos rotaciones sucesivas aplicadas a un punto  $\mathbf{P}$  producen la posición transformada:

$$\begin{aligned}\mathbf{P}' &= \mathbf{R}(\theta_2) \cdot \{\mathbf{R}(\theta_1) \cdot \mathbf{P}\} \\ &= \{\mathbf{R}(\theta_2) \cdot \mathbf{R}(\theta_1)\} \cdot \mathbf{P}\end{aligned}\quad (5.30)$$

Mediante la multiplicación de dos matrices de rotación, podemos verificar que dos rotaciones sucesivas son aditivas:

$$\mathbf{R}(\theta_2) \cdot \mathbf{R}(\theta_1) = \mathbf{R}(\theta_1 + \theta_2) \quad (5.31)$$

por tanto, las coordenadas rotadas finales de un punto pueden calcularse con la matriz de rotación compuesta como

$$\mathbf{P}' = \mathbf{R}(\theta_1 + \theta_2) \cdot \mathbf{P} \quad (5.32)$$

## Cambios de escala compuestos bidimensionales

Concatenar matrices de transformación para dos operaciones sucesivas de cambio de escala en dos dimensiones produce la siguiente matriz de cambio de escala compuesta:

$$\begin{bmatrix} s_{2x} & 0 & 0 \\ 0 & s_{2y} & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} s_{1x} & 0 & 0 \\ 0 & s_{1y} & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s_{1x} \cdot s_{2x} & 0 & 0 \\ 0 & s_{1y} \cdot s_{2y} & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (5.33)$$

o,

$$\mathbf{S}(s_{2x}, s_{2y}) \cdot \mathbf{S}(s_{1x}, s_{1y}) = \mathbf{S}(s_{1x} \cdot s_{2x}, s_{1y} \cdot s_{2y}) \quad (5.34)$$

La matriz resultante en este caso indica que operaciones de cambio de escala sucesivas son multiplicativas. Esto es, si quisieramos triplicar el tamaño de un objeto dos veces seguidas, el tamaño final sería nueve veces más grande que el original.

## Rotación general sobre un punto de pivote bidimensional

Cuando un paquete gráfico ofrece sólo una función de rotación con respecto al origen de coordenadas, podemos generar una rotación bidimensional sobre cualquier otro punto de pivote  $(x_r, y_r)$  representando la siguiente secuencia de operaciones traslación-rotación-traslación.

- (1) Trasladar el objeto de tal forma que la posición del punto de pivote se mueva al origen de coordenadas.
- (2) Rotar el objeto sobre el eje de coordenadas.
- (3) Trasladar el objeto de tal forma que el punto de pivote vuelva a su posición original.

Esta secuencia de transformaciones se ilustra en la Figura 5.9. La matriz de transformación compuesta para esta secuencia se obtiene con la concatenación:

$$\begin{aligned} & \begin{bmatrix} 1 & 0 & x_r \\ 0 & 1 & y_r \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -x_r \\ 0 & 1 & -y_r \\ 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} \cos \theta & -\sin \theta & x_r(1-\cos \theta) + y_r \sin \theta \\ \sin \theta & \cos \theta & y_r(1-\cos \theta) - x_r \sin \theta \\ 0 & 0 & 1 \end{bmatrix} \end{aligned} \quad (5.35)$$

que puede expresarse en la forma:

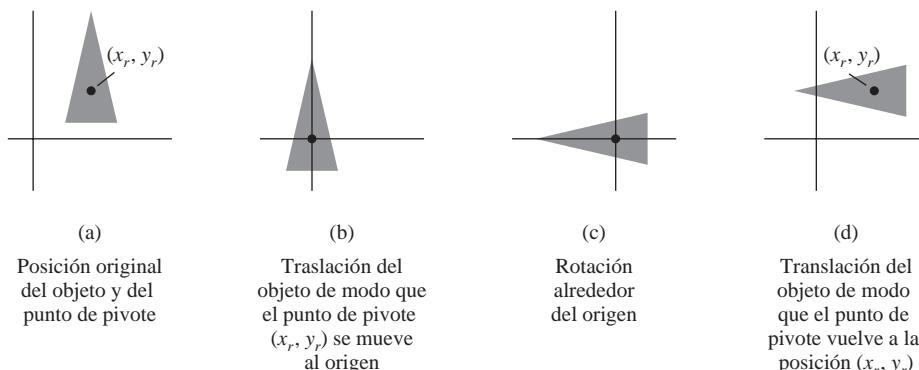
$$\mathbf{T}(x_r, y_r) \cdot \mathbf{R}(\theta) \cdot \mathbf{T}(-x_r, -y_r) = \mathbf{R}(x_r, y_r, \theta)$$

donde  $\mathbf{T}(-x_r, -y_r) = \mathbf{T}^{-1}(x_r, y_r)$ . En general, una función de rotación de una biblioteca gráfica, puede estructurarse para aceptar parámetros de coordenadas de un punto de pivote, así como de un ángulo de rotación, y para generar automáticamente la matriz de rotación de la Ecuación 5.35.

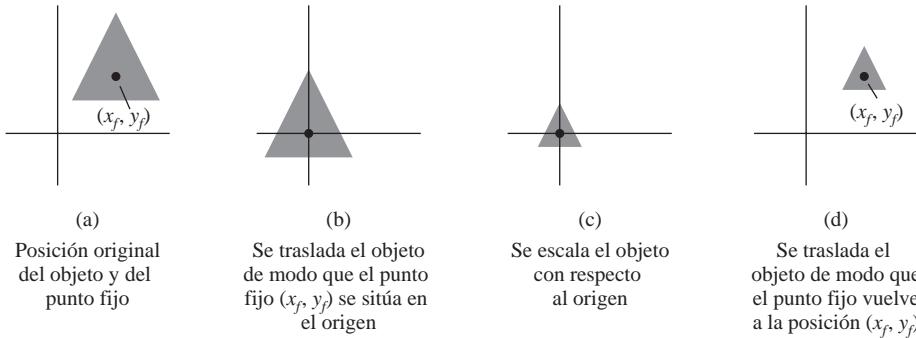
## Cambio de escala general de puntos fijos bidimensionales

La Figura 5.10 ilustra una secuencia de transformaciones para producir un cambio de escala bidimensional con respecto a una posición fija seleccionada  $(x_f, y_f)$  cuando tenemos una función que sólo puede realizar un cambio de escala respecto al origen de coordenadas. Esta secuencia es:

- (1) Trasladar el objeto de tal forma que el punto fijo coincida con el origen de coordenadas.



**FIGURA 5.9.** Secuencia de transformación para la rotación de un objeto sobre un punto de pivote especificado usando la matriz de rotación  $\mathbf{R}(\theta)$  de la transformación 5.19.



**FIGURA 5.10.** Secuencia de transformación para el cambio de escala de un objeto con respecto a una posición fija específica, usando la matriz de escala  $\mathbf{S}(s_x, s_y)$  de transformación 5.21.

- (2) Cambiar de escala un objeto con respecto al origen de coordenadas.
- (3) Usar la inversa de la traslación del paso (1) para devolver el objeto a su posición original.

La concatenación de matrices para estas tres operaciones produce la requerida matriz de cambio de escala:

$$\begin{bmatrix} 1 & 0 & x_f \\ 0 & 1 & y_f \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -x_f \\ 0 & 1 & -y_f \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & x_f(1-s_x) \\ 0 & s_y & y_f(1-s_y) \\ 0 & 0 & 1 \end{bmatrix} \quad (5.37)$$

o,

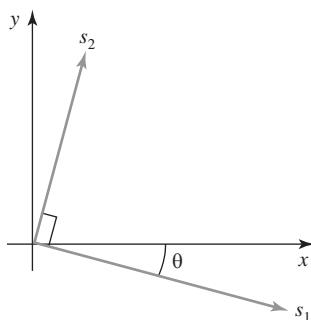
$$\mathbf{T}(x_f, y_f) \cdot \mathbf{S}(s_x, s_y) \cdot \mathbf{T}(-x_f, -y_f) = \mathbf{S}(x_f, y_f, s_x, s_y) \quad (5.38)$$

Esta transformación se genera automáticamente en sistemas que ofrecen una función de cambio de escala que acepta coordenadas para un punto fijo.

### Directrices generales para el cambio de escala bidimensional

Los parámetros  $s_x$  y  $s_y$  cambian la escala de objetos a lo largo de las direcciones  $x$  e  $y$ . Podemos cambiar de escala un objeto según otras direcciones mediante la rotación del objeto para alinear la dirección del cambio de escala deseado con los ejes de coordenadas antes de aplicar la transformación cambio de escala.

Supongamos que queremos aplicar factores de escala con valores especificados por los parámetros  $s_1$  y  $s_2$  en las direcciones mostradas en la Figura 5.11. Para llevar a cabo el cambio de escala, sin cambiar la orientación del objeto, primero se lleva a cabo la rotación de forma que las direcciones para  $s_1$  y  $s_2$  coincidan con los ejes  $x$  y  $y$ , respectivamente. Después, se aplica la transformación de cambio de escala  $\mathbf{S}(s_1, s_2)$  seguida de una



**FIGURA 5.11.** Parámetros de cambio de escala  $s_1$  y  $s_2$  a lo largo de direcciones ortogonales definidas por el desplazamiento angular  $\theta$ .

rotación opuesta a los puntos de retorno de sus orientaciones originales. La matriz compuesta resultante del producto de estas tres transformaciones es:

$$\mathbf{R}^{-1}(\theta) \cdot \mathbf{S}(s_1, s_2) \cdot \mathbf{R}(\theta) = \begin{bmatrix} s_1 \cos^2 \theta + s_2 \sin^2 \theta & (s_2 - s_1) \cos \theta \sin \theta & 0 \\ (s_2 - s_1) \cos \theta \sin \theta & s_1 \sin^2 \theta + s_2 \cos^2 \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (5.39)$$

Como ejemplo de esta transformación, giramos un cuadrado para convertirlo en un paralelogramo (Figura 5.12) estrechándolo a lo largo de la diagonal desde  $(0,0)$  hasta  $(1,1)$ . Primero rotamos la diagonal sobre el eje  $y$  usando  $\theta = 45^\circ$ , luego duplicamos su longitud con los valores de escala  $s_1 = 1$  y  $s_2 = 2$ , y después lo rotamos de nuevo para devolver la diagonal a su orientación original.

En la Ecuación 5.39, asumimos que ese cambio de escala iba a ser realizado con relación al origen. Podemos llevar esta operación de cambio de escala un paso más allá y concatenar la matriz con los operadores de traslación, de tal forma que la matriz compuesta incluiría los parámetros para la especificación de una posición fija de cambio de escala.

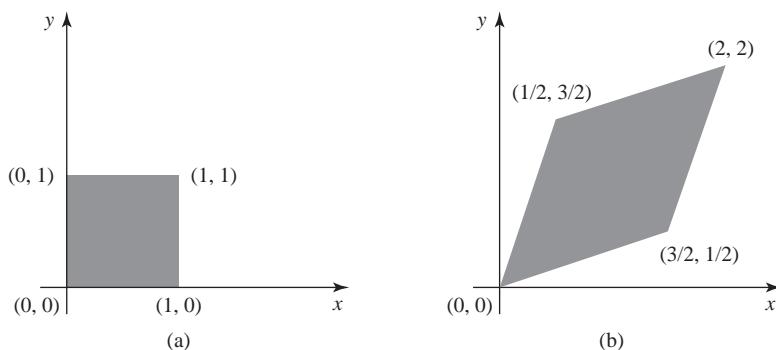
### Propiedades de la concatenación de matrices

La multiplicación de matrices es asociativa. Para tres matrices cualesquiera,  $\mathbf{M}_1$ ,  $\mathbf{M}_2$  y  $\mathbf{M}_3$ , la matriz producto  $\mathbf{M}_3 \cdot \mathbf{M}_2 \cdot \mathbf{M}_1$  puede obtenerse multiplicando primero  $\mathbf{M}_3$  y  $\mathbf{M}_2$  o multiplicando primero  $\mathbf{M}_2$  y  $\mathbf{M}_1$ :

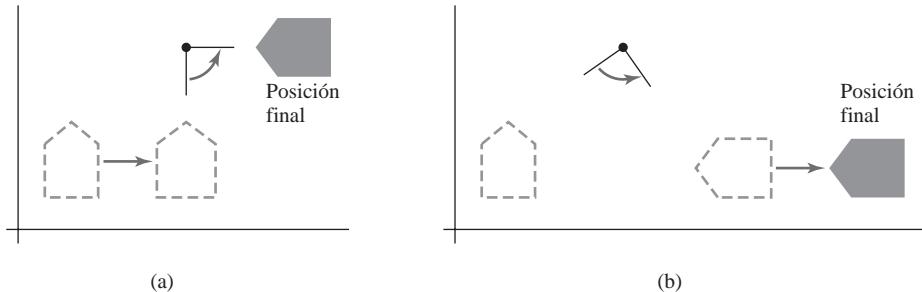
$$\mathbf{M}_3 \cdot \mathbf{M}_2 \cdot \mathbf{M}_1 = (\mathbf{M}_3 \cdot \mathbf{M}_2) \cdot \mathbf{M}_1 = \mathbf{M}_3 \cdot (\mathbf{M}_2 \cdot \mathbf{M}_1) \quad (5.40)$$

Por tanto, dependiendo del orden en el que se hayan especificado las transformaciones, podemos construir una matriz compuesta, bien multiplicando de izquierda a derecha (premultiplicando) o bien multiplicando de derecha a izquierda (postmultiplicando). Algunos paquetes gráficos requieren que las transformaciones se hagan especificando el orden en el que deben ser aplicadas. En este caso, invocaríamos primero la transformación  $\mathbf{M}_1$ , luego  $\mathbf{M}_2$  y después  $\mathbf{M}_3$ . A medida que se llama de manera sucesiva a cada rutina de transformación, su matriz es concatenada a la izquierda del producto de matrices previo. Otros sistemas gráficos, sin embargo, postmultiplican las matrices, así que esta secuencia de transformaciones tendría que invocarse en el orden inverso: la última transformación invocada (que para este ejemplo es  $\mathbf{M}_1$ ) es la primera en aplicarse, y la primera transformación que fue llamada ( $\mathbf{M}_3$  para nuestro ejemplo) es la última en aplicarse.

Por otra parte, el producto de transformaciones no puede ser commutativo. El producto de las matrices  $\mathbf{M}_1 \cdot \mathbf{M}_2$  en general no es igual que  $\mathbf{M}_2 \cdot \mathbf{M}_1$ . Esto significa que si queremos trasladar y rotar un objeto, debemos tener cuidado con el orden en se evalúa la matriz compuesta (Figura 5.13). Para algunos casos especiales, tales como una secuencia de transformaciones todas del mismo tipo, la multiplicación de matrices de



**FIGURA 5.12.** Un cuadrado (a) se convierte en un paralelogramo (b) utilizando la matriz de transformación compuesta 5.39, con  $s_1 = 1$ ,  $s_2 = 2$  y  $\theta = 45^\circ$ .



**FIGURA 5.13.** Invertir el orden en el que se lleva a cabo la secuencia de transformaciones puede afectar a la posición transformada de un objeto. En (a), un objeto primero se traslada en la dirección  $x$  y luego se rota en el sentido contrario al de las agujas del reloj con un ángulo de  $45^\circ$ . En (b), el objeto primero se rota  $45^\circ$  en el sentido contrario al de las agujas del reloj y después es trasladado en la dirección  $x$ .

transformación es conmutativa. Como ejemplo, dos rotaciones sucesivas pueden llevarse a cabo en cualquier orden y la posición final será la misma. Esta propiedad conmutativa también se aplica para dos traslaciones sucesivas o dos cambios de escala sucesivos. Otro par de operaciones conmutativo es la rotación y el cambio de escala uniforme ( $s_x = s_y$ ).

### Transformaciones compuestas bidimensionales generales y eficiencia de cálculo

Una transformación bidimensional, que representa cualquier combinación de traslaciones, rotaciones y cambios de escala se puede expresar como:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} rs_{xx} & rs_{xy} & trs_x \\ rs_{yx} & rs_{yy} & trs_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad (5.41)$$

Los cuatro elementos  $rs_{jk}$  son los términos multiplicativos rotación-escala en la transformación, lo que implica sólo ángulos de rotación y factores de escala. Los elementos  $trs_x$  y  $trs_y$ , son los términos traslacionales, que contienen combinaciones de distancias de traslación, coordenadas de puntos de pivote y puntos fijos, ángulos de rotación y parámetros de escala. Por ejemplo, si se va a cambiar de escala y rotar un objeto sobre las coordenadas de su centroide ( $x_c, y_c$ ) y luego se va a trasladar, los valores de los elementos de la matriz de transformación compuesta son:

$$\mathbf{T}(t_x, t_y) \cdot \mathbf{R}(x_c, y_c, \theta) \cdot \mathbf{S}(x_c, y_c, s_x, s_y)$$

$$= \begin{bmatrix} s_x \cos\theta & -s_y \sin\theta & x_c(1-s_x \cos\theta) + y_c s_y \sin\theta + t_x \\ s_x \sin\theta & s_y \cos\theta & y_c(1-s_y \cos\theta) - x_c s_x \sin\theta + t_y \\ 0 & 0 & 1 \end{bmatrix} \quad (5.42)$$

Aunque la Ecuación de matrices 5.41 requiere nueve multiplicaciones y seis sumas, los cálculos explícitos para las coordenadas transformadas son:

$$x' = x \cdot rs_{xx} + y \cdot rs_{xy} + trs_x \quad y' = x \cdot rs_{yx} + y \cdot rs_{yy} + trs_y \quad (5.43)$$

Por tanto, realmente necesitamos realizar cuatro multiplicaciones y cuatro sumas para transformar las posiciones de coordenadas. Éste es el máximo número de cálculos requerido para cualquier secuencia de transformación, una vez que las matrices individuales se han concatenado y se han evaluado los elementos de

la matriz compuesta. Sin concatenación, las transformaciones individuales se aplicarían una cada vez, y el número de cálculos podría incrementarse significativamente. Por tanto, una implementación eficiente para las operaciones de transformación es formular matrices de transformación, concatenar cualquier secuencia de transformación y calcular las coordenadas transformadas usando las Ecuaciones 5.43. En sistemas paralelos, las multiplicaciones directas de matrices con la matriz de transformación compuesta de la Ecuación 5.41 pueden ser igualmente eficientes.

Dado que los cálculos de rotación requieren evaluaciones trigonométricas y varias multiplicaciones para cada punto transformado, la eficiencia de cálculo puede convertirse en algo importante a considerar en las transformaciones de rotación. En animaciones y otras aplicaciones que implican muchas transformaciones repetidas y pequeños ángulos de rotación, podemos usar aproximaciones y cálculos iterativos para reducir los cálculos en las ecuaciones de transformación compuestas. Cuando el ángulo de rotación es pequeño, las funciones trigonométricas pueden ser sustituidas con valores aproximados basados en unos pocos primeros términos de su desarrollo en serie de potencias. Para ángulos suficientemente pequeños (inferiores a  $10^\circ$ )  $\cos \theta$  es, aproximadamente, 1.0 y  $\sin \theta$  tiene un valor muy próximo al valor de  $\theta$  radianes. Si estamos rotando en pequeños pasos angulares sobre el origen, por ejemplo, podemos igualar  $\cos \theta$  a 1.0 y reducir los cálculos de transformación a cada paso a dos multiplicaciones y dos sumas para cada juego de coordenadas que se quieran rotar. Estos cálculos de rotación son:

$$x' = x - y \sin \theta, \quad y' = x \sin \theta + y \quad (5.44)$$

donde  $\sin \theta$  se evalúa una vez para todos los pasos, asumiendo que el ángulo de rotación no cambia. El error introducido en cada paso por esta aproximación disminuye a la vez que disminuye el ángulo de rotación. Pero incluso con pequeños ángulos de rotación, el error acumulado a lo largo de muchos pasos puede ser bastante grande. Podemos controlar el error acumulado estimando el error en  $x'$  e  $y'$  a cada paso y reinicializando la posición del objeto cuando el error acumulado se vuelve demasiado grande. Algunas aplicaciones de animación reinician automáticamente las posiciones de un objeto y fijan intervalos, por ejemplo, cada  $360^\circ$  o cada  $180^\circ$ .

Las transformaciones compuestas a menudo implican matrices inversas. Por ejemplo, las secuencias de transformación para direcciones de escalado generales y para algunas reflexiones e inclinaciones (Sección 5.5) requieren rotaciones inversas. Como hemos podido ver, las representaciones de la matriz inversa para las transformaciones geométricas básicas pueden generarse con procedimientos sencillos. Una matriz de traslación inversa se obtiene cambiando los signos de las distancias de traslación, y una matriz de rotación inversa se obtiene mediante una matriz traspuesta (o cambiando el signo de los términos en seno). Estas operaciones son mucho más simples que los cálculos directos de matriz inversa.

## Transformación bidimensional de sólido-rígido

Si una matriz de transformación incluye sólo parámetros de traslación y rotación, es una **matriz de transformación de sólido-rígido**. La forma general para una matriz de transformación bidimensional de sólido-rígido es:

$$\begin{bmatrix} r_{xx} & r_{xy} & tr_x \\ r_{yx} & r_{yy} & tr_y \\ 0 & 0 & 1 \end{bmatrix} \quad (5.45)$$

donde los cuatro elementos  $r_{jk}$  son los términos multiplicativos de la rotación, y los elementos  $tr_x$  y  $tr_y$  son los términos translacionales. Un cambio en una posición de coordenadas de un sólido-rígido, a veces también se denomina transformación de **movimiento rígido**. Todos los ángulos y distancias entre posiciones de coordenadas son inalterables mediante una transformación. Además, la matriz 5.45 tiene la propiedad de que su submatriz de 2 por 2 superior izquierda es la *matriz ortogonal*. Esto significa que si consideramos cada fila (o

cada columna) de la submatriz como un vector, entonces los dos vectores fila  $(r_{xx}, r_{xy})$  (o los dos vectores columna) forman un juego ortogonal de vectores unidad. Tal juego de vectores, también se puede denominar juego de vectores *ortonormal*. Cada vector tiene como longitud la unidad:

$$r_{xx}^2 + r_{xy}^2 = r_{yx}^2 + r_{yy}^2 = 1 \quad (5.46)$$

y los vectores son perpendiculares (su producto escalar es 0):

$$r_{xx}r_{yx} + r_{xy}r_{yy} = 0 \quad (5.47)$$

Por tanto, si estos vectores unidad se transforman mediante la rotación de la submatriz, entonces el vector  $(r_{xx}, r_{xy})$  se convierte en un vector unidad a lo largo del eje  $x$  y el vector  $(r_{yx}, r_{yy})$  se transforma en un vector unidad a lo largo del eje  $y$  del sistema de coordenadas:

$$\begin{bmatrix} r_{xx} & r_{xy} & 0 \\ r_{yx} & r_{yy} & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} r_{xx} \\ r_{yx} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \quad (5.48)$$

$$\begin{bmatrix} r_{xx} & r_{xy} & 0 \\ r_{yx} & r_{yy} & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} r_{yx} \\ r_{yy} \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} \quad (5.49)$$

Por ejemplo, la siguiente transformación de sólido-rígido primero rota un objeto un ángulo  $\theta$  alrededor del punto de pivote  $(x_r, y_r)$  y después traslada el objeto.

$$T(t_x, t_y) \cdot R(x_r, y_r, \theta) = \begin{bmatrix} \cos\theta & -\sin\theta & x_r(1-\cos\theta) + y_r \sin\theta + t_x \\ \sin\theta & \cos\theta & y_r(1-\cos\theta) - x_r \sin\theta + t_y \\ 0 & 0 & 1 \end{bmatrix} \quad (5.50)$$

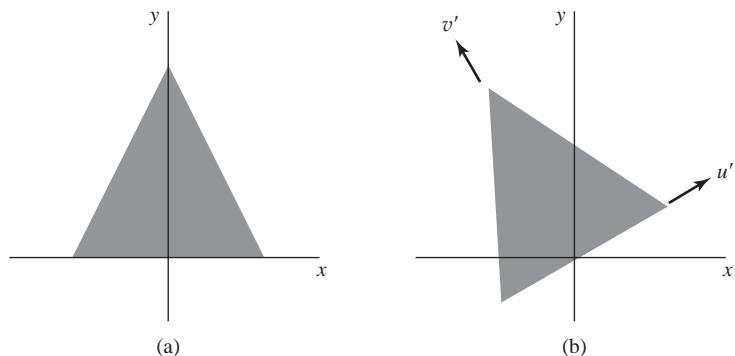
Aquí, los vectores unidad ortogonales en la submatriz superior izquierda son  $(\cos\theta, -\sin\theta)$  y  $(\sin\theta, \cos\theta)$  y,

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos\theta \\ -\sin\theta \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \quad (5.51)$$

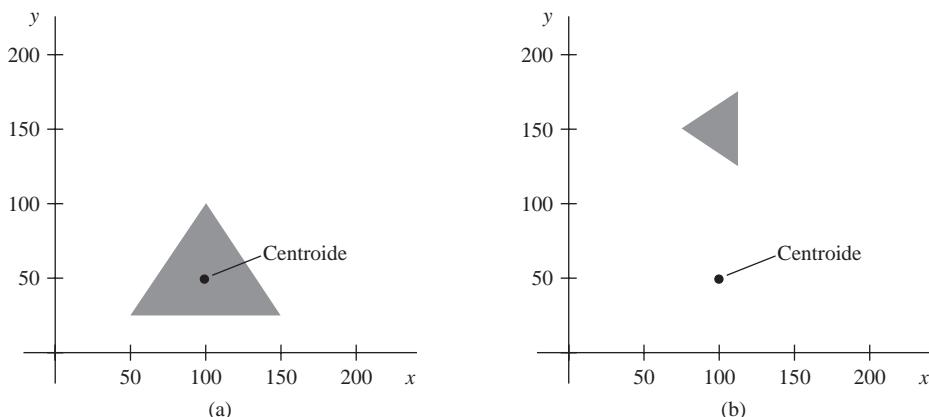
De manera similar, el vector unidad  $(\sin\theta, \cos\theta)$  se convierte, por la matriz de transformación precedente, en el vector unidad  $(0,1)$  en la dirección  $y$ .

## Construcción de matrices de rotación bidimensionales

La propiedad ortogonal de la rotación de matrices resulta útil para construir la matriz cuando conocemos la orientación final de un objeto, en lugar de la cantidad de rotaciones angulares necesarias para colocar el objeto en aquella posición. Esta información de orientación podría determinarse por la alineación de ciertos objetos en una escena o por posiciones de referencia entre sistemas de coordenadas. Por ejemplo, podríamos querer rotar un objeto para alinear su eje de simetría con la dirección de visualización (de la cámara) o podríamos querer rotar un objeto de tal forma que estuviera encima de otro objeto. La Figura 5.14 muestra un objeto que va a ser alineado con la dirección de los vectores unidad  $\mathbf{u}'$  y  $\mathbf{v}'$ . Asumiendo que la orientación del objeto original, como se muestra en la Figura 5.14(a), se alinea con el eje de coordenadas, construimos la



**FIGURA 5.14.** La matriz de rotación para hacer girar un objeto desde una posición (a) a una posición (b) puede construirse con los valores de los vectores unidad de orientación  $\mathbf{u}'$  y  $\mathbf{v}'$  relativos a la orientación original.



**FIGURA 5.15.** Un triángulo (a) es transformado a la posición (b) usando los cálculos de la matriz compuesta del procedimiento `transformVerts2D`.

transformación deseada asignando los elementos de  $\mathbf{u}'$  a la primera fila de la matriz de rotación y los elementos de  $\mathbf{v}'$  a la segunda fila. En una aplicación de modelado, por ejemplo, podemos usar este método para obtener la matriz de transformación dentro de un sistema de coordenadas de objetos locales cuando sabemos cuál va a ser su orientación dentro de las coordenadas universales de la escena. Una transformación similar es la conversión de descripciones de objeto desde un sistema de coordenadas a otro, en las Secciones 5.8 y 5.15 se estudian estos métodos más detalladamente.

### Ejemplo de programación de matrices bidimensionales compuestas

Un ejemplo de implementación para una secuencia de transformaciones geométricas se da en el siguiente programa. Inicialmente, la matriz compuesta, `compMatrix`, se construye como la matriz identidad. En este ejemplo, se usa una concatenación de izquierda a derecha para construir la matriz de transformación compuesta y se invocan las rutinas de transformación en el orden en el que son ejecutadas. A medida que cada una de las rutinas de transformación básicas (cambio de escala, rotación y traslación) se invoca, se establece una matriz para aquella transformación y se concatena por la izquierda con la matriz compuesta. Una vez que todas las transformaciones se han especificado, la transformación compuesta se aplica para transformar un triángulo. Primero se cambia la escala del triángulo con respecto a la posición de su centroide (Apéndice A) luego, se rota sobre su centroide y, por último, se traslada. La Figura 5.15 muestra las posiciones original y final de un

triángulo que se transforma mediante esta secuencia. Las rutinas OpenGL se usan para mostrar la posición inicial y final del triángulo.

```
#include <GL/glut.h>
#include <stdlib.h>
#include <math.h>

/* Establece el tamaño inicial de la ventana de visualización. */
GLsizei winWidth = 600, winHeight = 600;

/* Establece el rango de las coordenadas del universo. */
GLfloat xwcMin = 0.0, xwcMax = 225.0;
GLfloat ywcMin = 0.0, ywcMax = 225.0;

class wcPt2D {
public:
    GLfloat x, y;
};

typedef GLfloat Matrix3x3 [3][3];

Matrix3x3 matComposite;

const GLdouble pi = 3.14159;

void init (void)
{
    /* Establece el color de la ventana de visualización en blanco. */
    glClearColor (1.0, 1.0, 1.0, 0.0);
}

/* Construye la matriz identidad 3 por 3. */
void matrix3x3SetIdentity (Matrix3x3 matIdent3x3)
{
    GLint row, col;

    for (row = 0; row < 3; row++)
        for (col = 0; col < 3; col++)
            matIdent3x3 [row][col] = (row == col);
}

/* Premultiplica la matriz m1 por la matriz m2, almacenando el resultado en m2. */
void matrix3x3PreMultiply (Matrix3x3 m1, Matrix3x3 m2)
{
    GLint row, col;
    Matrix3x3 matTemp;

    for (row = 0; row < 3; row++)
        for (col = 0; col < 3 ; col++)
            matTemp [row][col] = m1 [row][0] * m2 [0][col] + m1 [row][1] *
                                m2 [1][col] + m1 [row][2] * m2 [2][col];
```

```

        for (row = 0; row < 3; row++)
            for (col = 0; col < 3; col++)
                m2 [row][col] = matTemp [row][col];
    }

void translate2D (GLfloat tx, GLfloat ty)
{
    Matrix3x3 matTransl;

    /* Inicializa la matriz de translación con la matriz identidad. */
    matrix3x3SetIdentity (matTransl);

    matTransl [0][2] = tx;
    matTransl [1][2] = ty;

    /* Concatena matTransl con la matriz compuesta. */
    matrix3x3PreMultiply (matTransl, matComposite);
}

void rotate2D (wcPt2D pivotPt, GLfloat theta)
{
    Matrix3x3 matRot;

    /* Inicializa la matriz de rotación con la matriz identidad. */
    matrix3x3SetIdentity (matRot);

    matRot [0][0] = cos (theta);
    matRot [0][1] = -sin (theta);
    matRot [0][2] = pivotPt.x * (1 - cos (theta)) + pivotPt.y * sin (theta);
    matRot [1][0] = sin (theta);
    matRot [1][1] = cos (theta);
    matRot [1][2] = pivotPt.y * (1 - cos (theta)) - pivotPt.x * sin (theta);

    /* Concatena matRot con la matriz compuesta. */
    matrix3x3PreMultiply (matRot, matComposite);
}

void scale2D (GLfloat sx, GLfloat sy, wcPt2D fixedPt)
{
    Matrix3x3 matScale;

    /* Inicializa la matriz de cambio de escala con la matriz identidad. */
    matrix3x3SetIdentity (matScale);

    matScale [0][0] = sx;
    matScale [0][2] = (1 - sx) * fixedPt.x;
    matScale [1][1] = sy;
    matScale [1][2] = (1 - sy) * fixedPt.y;

    /* Concatenate matScale with the composite matrix. */

    matrix3x3PreMultiply (matScale, matComposite);
}

/* Usando la matriz compuesta, se calculan las coordenadas transformadas. */

```

```
void transformVerts2D (GLint nVerts, wcPt2D * verts)
{
    GLint k;
    GLfloat temp;

    for (k = 0; k < nVerts; k++) {
        temp = matComposite [0][0] * verts [k].x + matComposite [0][1] *
               verts [k].y + matComposite [0][2];
        verts [k].y = matComposite [1][0] * verts [k].x + matComposite [1][1] *
                      verts [k].y + matComposite [1][2];
        verts [k].x = temp;
    }
}

void triangle (wcPt2D *verts)
{
    GLint k;

    glBegin (GL_TRIANGLES);
    for (k = 0; k < 3; k++)
        glVertex2f (verts [k].x, verts [k].y);
    glEnd ( );
}

void displayFcn (void)
{
    /* Define la posición inicial del triángulo. */
    GLint nVerts = 3;
    wcPt2D verts [3] = { {50.0, 25.0}, {150.0, 25.0}, {100.0, 100.0} };

    /* Calcula la posición del centroide del triángulo. */
    wcPt2D centroidPt;

    GLint k, xSum = 0, ySum = 0;
    for (k = 0; k < nVerts; k++) {
        xSum += verts [k].x;
        ySum += verts [k].y;
    }
    centroidPt.x = GLfloat (xSum) / GLfloat (nVerts);
    centroidPt.y = GLfloat (ySum) / GLfloat (nVerts);

    /* Establece los parámetros de la transformación geométrica. */
    wcPt2D pivPt, fixedPt;
    pivPt = centroidPt;
    fixedPt = centroidPt;

    GLfloat tx = 0.0, ty = 100.0;
    GLfloat sx = 0.5, sy = 0.5;
    GLdouble theta = pi/2.0;

    /* Borra la ventana de visualización. */
}
```

```
glClear (GL_COLOR_BUFFER_BIT);

/* Establece el color de relleno inicial en azul. */
glColor3f (0.0, 0.0, 1.0);

/* Muestra un triángulo azul. */
triangle (verts);

/* Inicializa la matriz compuesta con la matriz identidad. */
matrix3x3SetIdentity (matComposite);

/* Construye la matriz compuesta para la secuencia de transformación. */

scale2D (sx, sy, fixedPt); // Primera transformación: escala.

rotate2D (pivPt, theta); // Segunda transformación: rotación

translate2D (tx, ty); // Transformación final: traslación.

/* Aplica la matriz compuesta a los vértices del triángulo. */
transformVerts2D (nVerts, verts);

/* Establece el color para el triángulo transformado.*/
glColor3f (1.0, 0.0, 0.0);

/* Muestra el triángulo transformado en rojo.*/
triangle (verts);

glFlush ( );

}

void winReshapeFcn (GLint newWidth, GLint newHeight)
{

    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ( );
    gluOrtho2D (xwcMin, xwcMax, ywcMin, ywcMax);

    glClear (GL_COLOR_BUFFER_BIT);
}

void main (int argc, char ** argv)
{

    glutInit (&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    glutInitWindowPosition (50, 50);
    glutInitWindowSize (winWidth, winHeight);
    glutCreateWindow ("Geometric Transformation Sequence");

    init ( );
    glutDisplayFunc (displayFcn);
    glutReshapeFunc (winReshapeFcn);
    glutMainLoop ( );
}
```

## 5.5 OTRAS TRANSFORMACIONES BIDIMENSIONALES

Las transformaciones básicas tales como la traslación, la rotación y el cambio de escala son componentes estándares de las bibliotecas gráficas. Algunos paquetes ofrecen algunas transformaciones adicionales que pueden ser útiles en ciertas aplicaciones. Dos de dichas transformaciones son la reflexión y la inclinación.

### Reflexión

Una transformación que produce la imagen de un objeto en un espejo se llama **reflexión**. Para una reflexión bidimensional, esta imagen se genera respecto a un **eje de reflexión** rotando el objeto 180° sobre dicho eje de reflexión. Podemos elegir un eje de reflexión en el plano  $xy$  o perpendicular al plano  $xy$ . Cuando el eje de reflexión es una línea en el plano  $xy$ , la trayectoria de la rotación está también en el plano  $xy$ . A continuación se proporcionan ejemplos de algunas reflexiones comunes.

La reflexión respecto de la línea  $y = 0$  (el eje  $x$ ) se logra con la matriz de transformación:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (5.52)$$

Esta transformación conserva los valores  $x$ , pero «da la vuelta» a las posiciones de coordenadas de valores  $y$ . La orientación resultante de un objeto después de haber sido reflejado sobre el eje  $x$  se muestra en la Figura 5.16. Para imaginar la trayectoria de la transformación de rotación para esta reflexión, podemos pensar en el objeto plano moviéndose fuera del plano  $xy$  y girando 180° a través de un espacio tridimensional alrededor del eje  $x$  y colocado de nuevo sobre el plano  $xy$  al otro lado del eje  $x$ .

Una reflexión sobre la línea  $x = 0$  (el eje  $y$ ) vuelca las coordenadas  $x$  mientras que mantiene las mismas coordenadas  $y$ . La matriz para esta transformación es:

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (5.53)$$

La Figura 5.17 ilustra el cambio de posición de un objeto que ha sido reflejado respecto de la línea  $x = 0$ . La rotación equivalente en este caso es 180° a través del espacio tridimensional sobre el eje  $y$ .

Damos la vuelta tanto a las coordenadas  $x$  como a las  $y$  de un punto, mediante la reflexión relativa de un eje que es perpendicular al plano  $xy$  y que pasa por el origen de coordenadas. Esta reflexión a veces se deno-

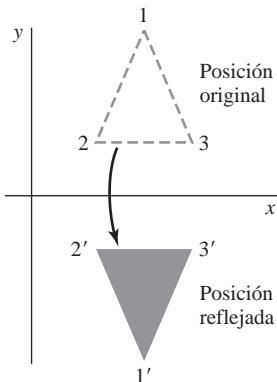


FIGURA 5.16. Reflexión de un objeto respecto al eje  $x$ .

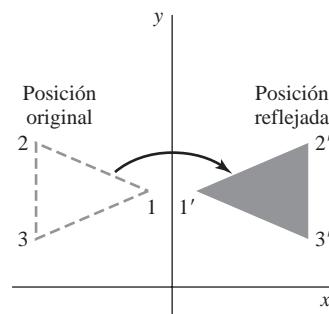


FIGURA 5.17. Reflexión de un objeto respecto al eje  $y$ .

mina reflexión relativa al origen de coordenadas, y es equivalente a reflejar con respecto a ambos ejes de coordenadas. La matriz de representación para esta reflexión es:

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (5.54)$$

En la Figura 5.18 se muestra un ejemplo de reflexión respecto del origen. La matriz de reflexión 5.54 es la misma que la matriz de rotación  $\mathbf{R}(\theta)$  con  $\theta = 180^\circ$ . Sencillamente estamos girando el objeto en el plano  $xy$  media vuelta alrededor del origen.

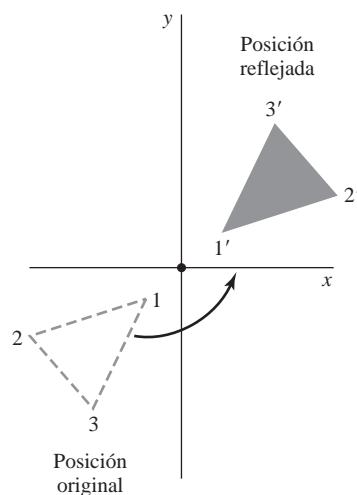
La matriz de reflexión 5.54 puede generalizarse para la reflexión de cualquier punto en el plano  $xy$  (Figura 5.19). Esta reflexión es lo mismo que la rotación de  $180^\circ$  en el plano  $xy$  alrededor de un punto de reflexión.

Si elegimos el eje de reflexión como la línea diagonal  $y = x$ , (Figura 5.20) la matriz de reflexión es:

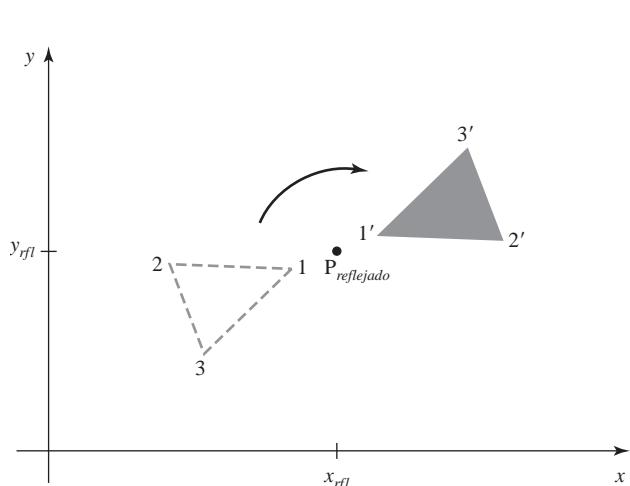
$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (5.55)$$

Podemos obtener esta matriz concatenando una secuencia de rotaciones y reflexiones de matrices sobre los ejes de coordenadas. Una posible secuencia se muestra en la Figura 5.21. Aquí, primero llevamos a cabo una rotación en el sentido de las agujas del reloj con respecto al origen a través de un ángulo de  $45^\circ$ , que rota la línea  $y = x$  y sobre el eje  $x$ . A continuación realizamos una reflexión con respecto al eje  $x$ . El paso final consiste en girar la línea  $y = x$  de vuelta a su posición original con una rotación de  $45^\circ$  en sentido contrario al de las agujas del reloj. Otra secuencia de transformaciones equivalente consiste en reflejar primero el objeto sobre el eje  $x$  y luego rotarlo  $90^\circ$  en el sentido de las agujas del reloj.

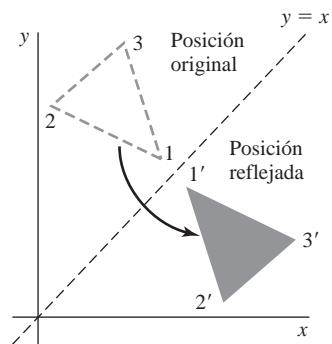
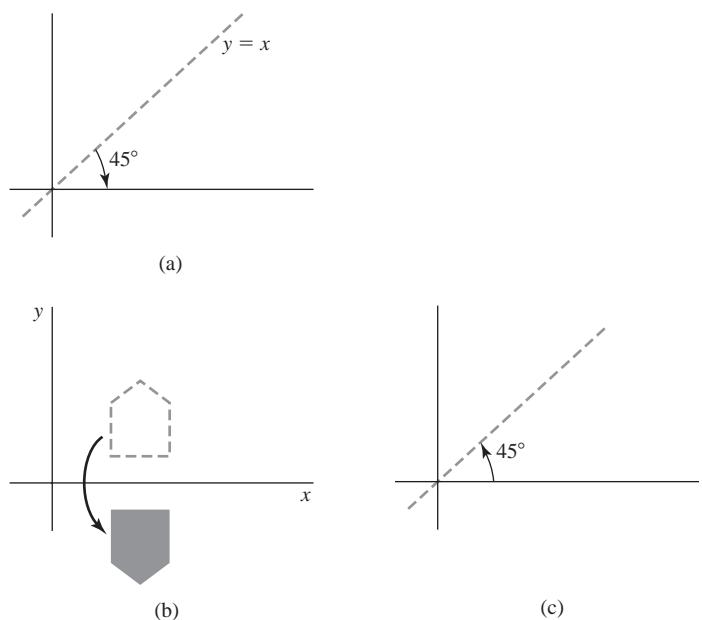
Para obtener una matriz de transformación sobre la diagonal  $y = -x$ , podemos concatenar matrices para la secuencia de transformación: (1) rotación de  $45^\circ$  en el sentido de las agujas del reloj, (2) reflexión sobre el eje  $y$ , y (3) rotación de  $45^\circ$  en el sentido de las agujas del reloj. La matriz de transformación resultante es:



**FIGURA 5.18.** Reflexión de un objeto respecto al origen de coordenadas. Esta transformación puede realizarse con una rotación en el plano  $xy$  sobre el origen de coordenadas.



**FIGURA 5.19.** Reflexión de un objeto respecto a un eje perpendicular al plano  $xy$  que pasa por el punto  $P_{reflejado}$ .

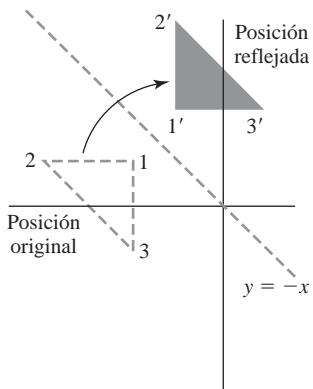
FIGURA 5.20. Reflexión de un objeto con respecto a la línea  $y = x$ .FIGURA 5.21. Secuencia de transformaciones para producir una reflexión respecto a la línea  $y = x$ : una rotación de  $45^\circ$  en el sentido de las agujas del reloj (a) una reflexión sobre el eje  $x$  (b) y una rotación de  $45^\circ$  en el sentido contrario al de las agujas del reloj (c).

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (5.56)$$

La Figura 5.22 muestra las posiciones original y final para un objeto transformado con esta matriz de reflexión.

Las reflexiones sobre cualquier línea  $y = mx + b$  en el plano  $xy$  pueden realizarse con una combinación de transformaciones traslación-rotación-reflexión. En general, primero trasladamos la línea de tal forma que pase por el origen. Luego podemos girar la línea hacia uno de los ejes de coordenadas y reflejarla sobre dicho eje. Finalmente, reestablecemos la línea a su posición original con las transformaciones inversas de rotación y traslación.

Podemos implementar reflexiones con respecto a los ejes de coordenadas o al origen de coordenadas como transformaciones de escala con factores de escala negativos. Además, los elementos de la matriz de reflexión pueden definirse con valores distintos de  $\pm 1$ . Un parámetro de reflexión de una magnitud superior a 1 cambia la imagen del espejo por un punto más alejado del eje de reflexión, y un parámetro cuya magnitud es inferior a 1 trae la imagen del espejo a un punto más cercano al eje de reflexión. Así, un objeto reflejado puede también agrandarse, reducirse o distorsionarse.

FIGURA 5.22. Reflexión con respecto a la línea  $y = -x$ .

## Inclinar

Una transformación que distorsiona la forma de un objeto de tal manera que la forma obtenida aparece como si el objeto estuviera compuesto por capas internas que se hubieran obtenido resbalando unas sobre otras es lo que se denomina **inclinación**. Dos transformaciones comunes para producir una inclinación son aquellas que desplazan los valores de las coordenadas  $x$  y las que desplazan los valores de  $y$ .

Una inclinación en la dirección- $x$  respecto al eje  $x$  se produce con la matriz de transformación:

$$\begin{bmatrix} 1 & sh_x & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (5.57)$$

la cual transforma la posición de coordenadas como sigue:

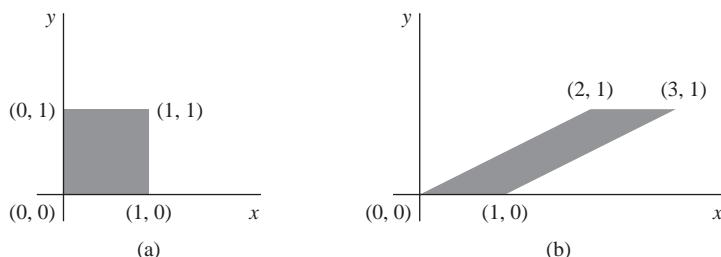
$$x' = x + sh_x \cdot y, \quad y' = y \quad (5.58)$$

Cualquier número real puede asignarse a los parámetros de inclinación  $sh_x$ . Entonces, una posición de coordenadas  $(x, y)$  se cambia horizontalmente por una cantidad proporcional a su distancia perpendicular (valor  $y$ ) desde el eje  $x$ . Establecer el parámetro  $sh_x$  por ejemplo con el valor 2, cambia el cuadrado de la Figura 5.23 por un paralelogramo. Los valores negativos para  $sh_x$  cambian las posiciones de las coordenadas hacia la izquierda.

Podemos generar una inclinación en la dirección- $x$  respecto a otras líneas de referencia con

$$\begin{bmatrix} 1 & sh_x & -sh_x \cdot y_{\text{ref}} \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (5.59)$$

Ahora, las posiciones de las coordenadas se transforman en:

FIGURA 5.23. Un cuadrado (a) se convierte en un paralelogramo (b) utilizando la matriz de inclinación 5.57 en la dirección- $x$  con  $sh_x = 2$ .

$$x' = x + sh_x(y - y_{\text{ref}}), \quad y' = y \quad (5.60)$$

Un ejemplo de esta transformación de inclinación se da en la Figura 5.24 para un valor de  $\frac{1}{2}$  del parámetro de inclinación respecto a la línea  $y_{\text{ref}} = -1$ .

Una inclinación en la dirección-y respecto a la línea  $x = x_{\text{ref}}$  se genera con la matriz de transformación:

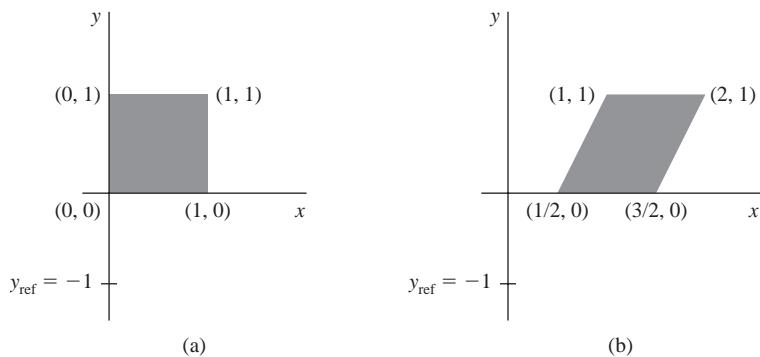
$$\begin{bmatrix} 1 & 0 & 0 \\ sh_y & 1 & -sh_y \cdot x_{\text{ref}} \\ 0 & 0 & 1 \end{bmatrix} \quad (5.61)$$

la cual genera los siguientes valores de coordenadas transformadas:

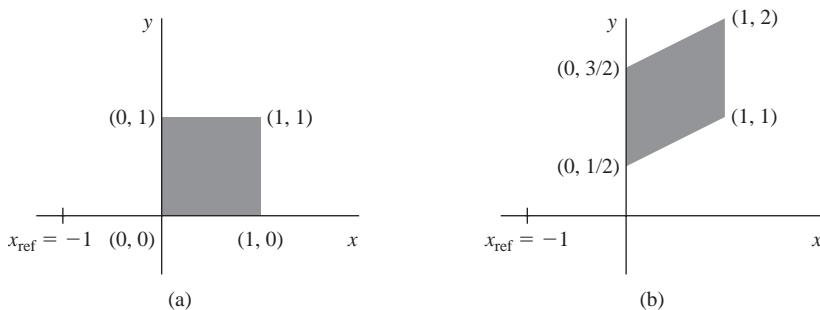
$$x' = x, \quad y' = y + sh_y(x - x_{\text{ref}}) \quad (5.62)$$

Esta transformación cambia la posición de coordenadas verticalmente en una cantidad proporcional a su distancia de la línea de referencia  $x = x_{\text{ref}}$ . La Figura 5.25 ilustra la conversión de un cuadrado en un paralelogramo con  $sh_y = 0.5$  y  $x_{\text{ref}} = -1$ .

Las operaciones de inclinación pueden expresarse como secuencias de transformaciones básicas. La matriz de inclinación en la dirección-x 5.57, por ejemplo, puede representarse como una transformación compuesta que implica una serie de matrices de rotación y escala. Esta transformación compuesta escala el cuadrado de la Figura 5.23 a lo largo de su diagonal, mientras que mantiene las longitudes originales y las orientaciones de los extremos paralelos al eje  $x$ . Los cambios en las posiciones de los objetos relativos a las líneas de inclinación de referencia son equivalentes a las traslaciones.



**FIGURA 5.24.** Un cuadrado (a) se transforma en un paralelogramo desplazado (b) con  $sh_x = 0.5$  y  $y_{\text{ref}} = -1$  en la matriz de inclinación 5.59.



**FIGURA 5.25.** Un cuadrado(a) se convierte en un paralelogramo desplazado (b) usando los siguientes valores de los parámetros  $sh_y = 0.5$  y  $x_{\text{ref}} = -1$  en la matriz de inclinación en la dirección-y 5.61.

## 5.6 MÉTODOS DE RASTERIZACIÓN PARA TRANSFORMACIONES GEOMÉTRICAS

Las características de los sistemas de rasterización sugieren un método alternativo para realizar ciertas transformaciones bidimensionales. Los sistemas de rasterización almacenan información de dibujo, como patrones de color en el búfer de imagen. Por tanto, algunas transformaciones de objetos simples pueden realizarse rápidamente manipulando un array de valores de píxeles. Se necesitan pocas operaciones aritméticas, así que las transformaciones de píxel son particularmente eficientes.

Como hemos podido ver en la Sección 3.19, las funciones que manipulan los arrays de píxeles rectangulares se denominan *operaciones de rasterización*, y mover un bloque de valores de píxel de una posición a otra se denomina *transferencia de bloque*, *bitblt* o *pixelblt*. Normalmente, pueden encontrarse las rutinas para realizar algunas operaciones de rasterización en los paquetes gráficos.

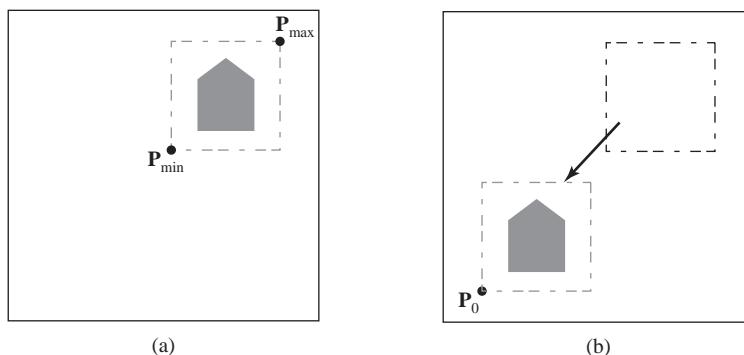
La Figura 5.26 ilustra una traslación bidimensional implementada como una transferencia de bloque de un área de refresco de búfer. Todas las características del bit, mostradas en el área rectangular, se copian como un bloque dentro de otra parte de un búfer de imagen. Podemos borrar el patrón en la ubicación original asignando el color de fondo a todos los píxeles contenidos en dicho bloque (suponiendo que el patrón que va a borrarse no se solapa con otros objetos de la escena).

Los incrementos de  $90^\circ$  en la rotación se llevan a cabo fácilmente reorganizando los elementos de un array de píxeles. Podemos rotar un objeto o un prototipo bidimensional  $90^\circ$  en el sentido contrario al de las agujas del reloj invirtiendo los valores de los píxeles en cada fila del array, y después intercambiando filas por columnas. Una rotación de  $180^\circ$  se obtiene invirtiendo el orden de los elementos en cada fila del array, y después invirtiendo el orden de las filas. La Figura 5.27 muestra las manipulaciones de array que pueden usarse para rotar el bloque de píxeles  $90^\circ$  y  $180^\circ$ .

Para rotaciones de arrays que no son múltiplos de  $90^\circ$  es necesario realizar algún procesamiento adicional. El procedimiento general se ilustra en la Figura 5.28. Cada área destinada a píxeles se mapea sobre el array rotado y se calcula la cantidad superpuesta con las áreas rotadas de píxeles. Puede entonces calcularse un color para un píxel de destino, promediando los colores de los píxeles de origen superpuestos y calculando su peso en función del porcentaje de área superpuesta. O, podríamos usar un método de aproximación, como el que se usa en el efecto de suavizado (*antialiasing*), para determinar el color de los píxeles de destino.

Podemos usar métodos similares para cambiar la escala de un bloque de píxeles. Las áreas de píxeles en el bloque original se cambian de escala, usando valores específicos para  $s_x$  y  $s_y$ , y luego se mapean sobre un conjunto de píxeles de destino. El color de cada píxel de destino es asignado entonces, de acuerdo con su área de solapamiento con las áreas de los píxeles cambiados de escala (Figura 5.29).

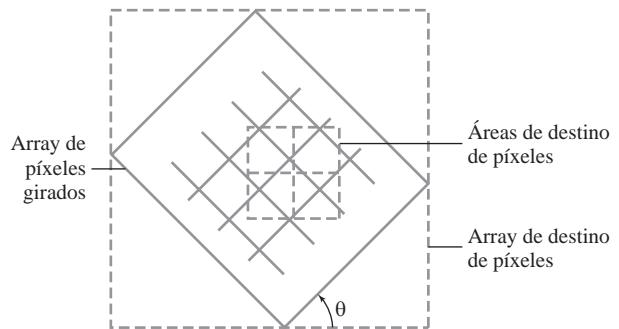
Un objeto puede reflejarse usando transformaciones de rasterización, que invierten los valores de filas y columnas en el bloque de píxeles, combinadas con traslaciones. Las inclinaciones se producen mediante desplazamientos de las posiciones de los valores del array a lo largo de filas o columnas.



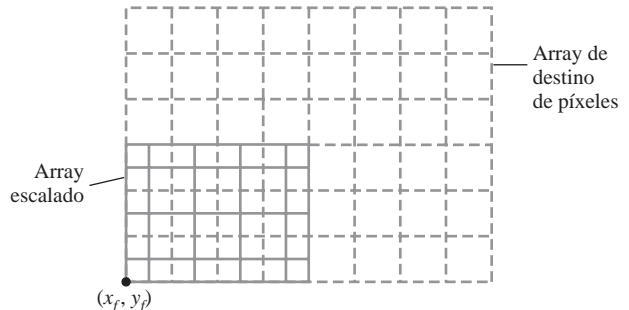
**FIGURA 5.26.** Traslación de un objeto desde la posición de la pantalla (a) a la posición de destino mostrada en (b), moviendo un bloque rectangular de valores de píxeles. Las posiciones de coordenadas  $P_{\min}$  y  $P_{\max}$  especifican los límites del bloque rectangular que va a ser movido y  $P_0$  es la posición de referencia de destino.

$$\begin{array}{c}
 \text{(a)} \quad \left[ \begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{array} \right] \\
 \text{(b)} \quad \left[ \begin{array}{cccc} 3 & 6 & 9 & 12 \\ 2 & 5 & 8 & 11 \\ 1 & 4 & 7 & 10 \end{array} \right] \\
 \text{(c)} \quad \left[ \begin{array}{ccc} 12 & 11 & 10 \\ 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{array} \right]
 \end{array}$$

**FIGURA 5.27.** Rotación de un array de valores de píxeles. El array original se muestra en (a), las posiciones de los elementos del array después de una rotación de  $90^\circ$  en el sentido contrario al de las agujas del reloj se muestran en (b) y las posiciones de los elementos del array después de una rotación de  $180^\circ$  se muestran en (c).



**FIGURA 5.28.** Una rotación de líneas para un bloque rectangular de píxeles puede llevarse a cabo mapeando las áreas de destino de píxeles sobre el bloque rotado.



**FIGURA 5.29.** Mapeo de áreas de destino de píxeles sobre un array cambiado de escala de valores de píxeles. Los factores de escala  $s_x = s_y = 0.5$  se aplican respecto a un punto fijo  $(x_f, y_f)$ .

## 5.7 TRANSFORMACIONES DE RASTERIZACIÓN EN OpenGL

En la Sección 3.19 presentamos la mayoría de las funciones OpenGL para la realización de operaciones de rasterización. Una traslación de un array rectangular de valores de píxeles de color, de un área de búfer a otra, puede llevarse a cabo en OpenGL como una operación de copia:

```
glCopyPixels (xmin, ymin, width, height, GL_COLOR)
```

Los primeros cuatro parámetros de esta función dan la localización y las dimensiones del bloque de píxeles. Y la constante simbólica de OpenGL, `GL_COLOR`, especifica que son valores de color que van a ser copiados. Este array de píxeles va a ser copiado en un área rectangular de un búfer de refresco, cuya esquina inferior izquierda está en la localización especificada por la posición de rasterización actual. Los valores de los píxeles de color se copian bien de los valores RGBA o bien de los índices de la tabla de color, dependiendo de la configuración actual para el modo de color. Tanto las regiones que van a ser copiadas (el origen) como el área de destino, deberían encontrarse completamente dentro de los límites de las coordenadas de la pantalla. Esta traslación puede llevarse a cabo en cualquier búfer de OpenGL de los que se usan para el refresco, o incluso entre diferentes búferes. Un búfer de origen para la función `glCopyPixels` se elige con la rutina `glReadBuffer` y el búfer de destino se selecciona con la rutina `glDrawBuffer`.

Podemos rotar un bloque de valores de píxeles de color en incrementos de 90 grados, primero guardando el bloque en un array, cambiando después los elementos del array y situándolos de nuevo en el búfer de refresh. Como vimos en la Sección 3.19, un bloque de valores de color RGB, que esté en un búfer, puede guardarse en un array con la función:

```
glReadPixels (xmin, ymin, width, height, GL_RGB, GL_UNSIGNED_BYTE, colorArray);
```

Si los índices de la tabla de color se almacenan en las posiciones de los píxeles, podemos sustituir la constante `GL_RGB` por `GL_COLOR_INDEX`. Para rotar los valores de color, cambiamos las filas y las columnas del array de color, tal y como se describe en la sección anterior. Después, ponemos el array rotado de nuevo en el búfer con:

```
glDrawPixels (width, height, GL_RGB, GL_UNSIGNED_BYTE, colorArray);
```

La esquina inferior izquierda de este array se sitúa en la posición de rastreo actual. Podemos seleccionar el búfer de origen que contiene el bloque original de valores de píxeles con `glReadBuffer`, y designamos un búfer de destino con `glDrawBuffer`.

Una transformación de escala bidimensional puede desarrollarse como una operación de rasterización en OpenGL, especificando los factores de escala y después invocando bien `glCopyPixels` o `glDrawPixels`. Para las operaciones de rasterización establecemos los factores de escala con:

```
glPixelZoom (sx, sy);
```

donde los parámetros `sx` y `sy` pueden asignarse a cualquier valor de un punto flotante distinto de cero. Los valores positivos superiores a 1.0, incrementan el tamaño de un elemento en el array de origen, y valores positivos inferiores a 1.0 decrementan el tamaño del elemento. Un valor negativo para `sx`, `sy`, o para ambos, produce una reflexión así como un cambio de escala de los elementos del array. Así, si `sx = sy = -3.0`, el array de origen se refleja con respecto a la posición de barrido actual y cada elemento de color del array se mapea a un bloque de píxeles destino de 3 por 3. Si el centro de un píxel destino se encuentra dentro de un área rectangular de un elemento de color cambiado de escala perteneciente a un array, se asigna el color de ese elemento del array. A los píxeles de destino cuyos centros están en los límites izquierdo o superior del array de elementos cambiados de escala, también se les asigna el color de aquel elemento. El valor predeterminado tanto para `sx` como para `sy` es 1.0.

También podemos combinar transformaciones de rasterización con las operaciones lógicas vistas en la Sección 3.19, para producir varios efectos. Con el operador *or exclusive*, por ejemplo, dos copias sucesivas de un array de píxeles al mismo área de búfer, reestablece los valores que estaban presentes originalmente en dicha área. Esta técnica puede usarse en una aplicación de animación para trasladar un objeto a través de una escena sin alterar los píxeles del fondo.

## 5.8 TRANSFORMACIONES ENTRE SISTEMAS DE COORDENADAS BIDIMENSIONALES

---

Las aplicaciones de gráficos por computadora implican transformaciones de coordenadas de un marco de referencia a otro durante varias etapas del procesamiento de la escena. Las rutinas de visualización transforman descripciones de objetos de las coordenadas universales a unas coordenadas de dispositivo de salida. Para aplicaciones de modelado y diseño, los objetos individuales se definen típicamente en sus propias referencias cartesianas locales. Estas descripciones de coordenadas locales deben, por tanto, transformarse en posiciones y orientaciones dentro del sistema de coordenadas total de la escena. Un programa para facilitar la gestión de la disposición de oficinas, por ejemplo, tiene descripciones de coordenadas individuales para sillas y mesas y otros muebles que pueden colocarse en el plano de la planta, con múltiples copias de sillas y otros elementos en diferentes posiciones.

Además, las escenas a veces se describen en marcos de referencia no cartesianos que aprovechan las simetrías de los objetos. Las descripciones de coordenadas en estos sistemas deben convertirse a coordenadas uni-

versales cartesianas para ser procesadas. Algunos ejemplos de sistemas no cartesianos son las coordenadas polares, las coordenadas esféricas, las coordenadas elípticas y las coordenadas parabólicas. Las relaciones entre los sistemas de referencia cartesianos y algunos sistemas no cartesianos comunes se dan en el Apéndice A. Aquí, sólo consideramos las transformaciones involucradas en la conversión de un marco cartesiano bidimensional a otro.

La Figura 5.30 muestra un sistema cartesiano  $x'y'$  especificado con el origen de coordenadas  $(x_0, y_0)$  y un ángulo de orientación  $\theta$  en un marco de referencia cartesiano  $xy$ . Para transformar las descripciones del objeto de las coordenadas  $xy$  a las coordenadas  $x'y'$ , establecemos una transformación que superpone los ejes  $x'y'$  sobre los ejes  $xy$ . Esto se realiza en dos pasos:

- (1) Traslación de tal forma que el origen  $(x_0, y_0)$  del sistema  $x'y'$  se mueva al origen  $(0,0)$  del sistema  $xy$ .
- (2) Rotación del eje  $x'$  sobre el eje  $x$ .

La transformación del origen de coordenadas se lleva a cabo con la matriz de transformación:

$$\mathbf{T}(-x_0, -y_0) = \begin{bmatrix} 1 & 0 & -x_0 \\ 0 & 1 & -y_0 \\ 0 & 0 & 1 \end{bmatrix} \quad (5.63)$$

La orientación de los dos sistemas después de la operación de traslación deberían aparecer como en la Figura 5.31. Para conseguir que los ejes de los dos sistemas coincidan, hacemos una rotación en el sentido de las agujas del reloj:

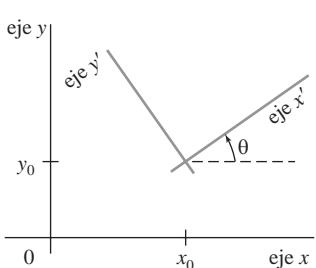
$$\mathbf{R}(-\theta) = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (5.64)$$

Concatenar estas dos matrices de transformación, nos da como resultado la matriz compuesta completa para transformar las descripciones del objeto desde el sistema  $xy$  al sistema  $x'y'$ :

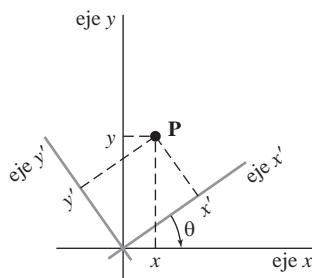
$$\mathbf{M}_{xy,x'y'} = \mathbf{R}(-\theta) \cdot \mathbf{T}(-x_0, -y_0) \quad (5.65)$$

Un método alternativo para describir la orientación del sistema de coordenadas  $x'y'$  consiste en especificar un vector  $\mathbf{V}$  que indique la dirección para el eje positivo  $y'$ , tal como se muestra en la Figura 5.32. Podemos especificar el vector  $\mathbf{V}$  como un punto en el marco de referencia  $xy$  relativo al origen del sistema  $xy$ , el cual podemos convertir en el vector unidad

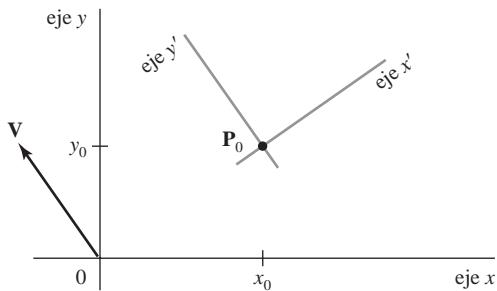
$$\mathbf{v} = \frac{\mathbf{V}}{|\mathbf{V}|} = (v_x, v_y) \quad (5.66)$$



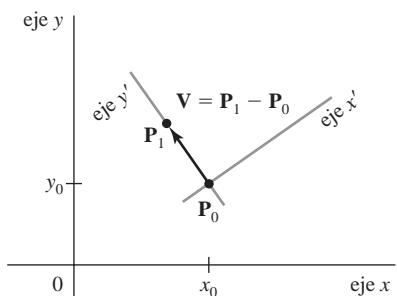
**FIGURA 5.30.** Un sistema cartesiano  $x'y'$  posicionado en  $(x_0, y_0)$  con orientación  $\theta$  en un sistema cartesiano  $xy$ .



**FIGURA 5.31.** Posición de los marcos de referencia mostrados en la Figura 5.30 después de trasladar el origen del sistema  $x'y'$  al origen de coordenadas del sistema  $xy$ .



**FIGURA 5.32.** Sistema cartesiano  $x'y'$  con origen en  $\mathbf{P}_0 = (x_0, y_0)$  y eje  $y'$  paralelo al vector  $\mathbf{V}$ .



**FIGURA 5.33.** Un sistema cartesiano  $x'y'$  definido por dos posiciones de coordenadas,  $\mathbf{P}_0$  y  $\mathbf{P}_1$ , dentro de un marco de referencia  $xy$ .

Y obtenemos el vector unidad  $\mathbf{u}$  a lo largo del eje  $x'$ , aplicando una rotación de  $90^\circ$  en el sentido de las agujas del reloj al vector  $\mathbf{v}$ :

$$\mathbf{u} = (v_y, -v_x) = (u_x, u_y) \quad (5.67)$$

En la Sección 5.4, vimos que los elementos de cualquier matriz de rotación podían expresarse como elementos de un conjunto de vectores ortonormales. Por tanto, la matriz para rotar el sistema  $x'y'$  y hacerlo coincidir con el sistema  $xy$ , puede escribirse como:

$$R = \begin{bmatrix} u_x & u_y & 0 \\ v_x & v_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (5.68)$$

Por ejemplo, supóngase que elegimos la orientación  $\mathbf{V} = (-1, 0)$  para el eje  $y'$ . Entonces el eje  $x'$  está en la dirección positiva de  $y$  y la matriz para la transformación de rotación es:

$$\begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

De igual manera, podemos obtener esta matriz de rotación de la Ecuación 5.64 estableciendo como ángulo de orientación  $\theta = 90^\circ$ .

En una aplicación interactiva, sería más conveniente elegir una dirección relativa a la posición  $\mathbf{P}_0$  para  $\mathbf{V}$ , que especificarlo respecto al origen de coordenadas  $xy$ . Los vectores unidad  $\mathbf{u}$  y  $\mathbf{v}$  estarían entonces orientados como se muestra en la Figura 5.33. Los componentes de  $\mathbf{v}$  se obtienen ahora como:

$$\mathbf{v} = \frac{\mathbf{P}_1 - \mathbf{P}_0}{|\mathbf{P}_1 - \mathbf{P}_0|} \quad (5.69)$$

y  $\mathbf{u}$  se obtiene como la perpendicular a  $\mathbf{v}$  que forma un sistema cartesiano a derechas.

## 5.9 TRANSFORMACIONES GEOMÉTRICAS EN UN ESPACIO TRIDIMENSIONAL

---

Los métodos para transformaciones geométricas en tres dimensiones pueden obtenerse extendiendo los métodos bidimensionales, incluyendo consideraciones para la coordenada  $z$ . Ahora, trasladamos un objeto especificando un vector de traslación tridimensional, que determina cuánto va a ser movido el objeto en cada una de las tres direcciones de coordenadas. De la misma forma, cambiamos la escala de un objeto eligiendo un factor de escala para cada una de las tres coordenadas cartesianas. Pero la extensión de los métodos de rotación bidimensionales a los de tres dimensiones es menos directa.

Cuando hemos visto las rotaciones bidimensionales en el plano  $xy$ , necesitábamos considerar sólo las rotaciones sobre los ejes que fueran perpendiculares al plano  $xy$ . En un espacio tridimensional, podemos seleccionar cualquier orientación espacial para la rotación de ejes. Algunos paquetes gráficos soportan rotaciones tridimensionales como una composición de tres rotaciones, una para cada uno de los tres ejes cartesianos. Alternativamente, podemos establecer unas ecuaciones generales de rotación, dando la orientación de rotación del eje y el ángulo de rotación requerido.

Una posición tridimensional, expresada en coordenadas homogéneas, se representa como un vector columna de cuatro elementos. Así, cada operando de la transformación geométrica es ahora una matriz de 4 por 4, que premultiplica un vector columna de coordenadas. Y, al igual que en dos dimensiones, cualquier secuencia de transformaciones se representa como una matriz simple, formada por la concatenación de matrices para las transformaciones individuales de la secuencia. Cada matriz sucesiva en una secuencia de transformación se concatena a la izquierda de las matrices de transformación previas.

## 5.10 TRASLACIONES TRIDIMENSIONALES

---

Una posición  $\mathbf{P} = (x, y, z)$  en un espacio tridimensional, se traslada a la posición  $\mathbf{P}' = (x', y', z')$  añadiendo las distancias de traslación  $t_x$ ,  $t_y$  y  $t_z$  a las coordenadas cartesianas de  $\mathbf{P}$ :

$$x' = x + t_x, \quad y' = y + t_y, \quad z' = z + t_z \quad (5.70)$$

La Figura 5.34 ilustra la traslación de un punto tridimensional.

Podemos expresar estas operaciones de traslación tridimensionales en matrices de la forma de la Ecuación 5.17. Pero ahora las posiciones de coordenadas,  $\mathbf{P}$  y  $\mathbf{P}'$ , se representan en coordenadas homogéneas con matrices columna de cuatro elementos y el operador de traslación  $\mathbf{T}$  es una matriz de 4 por 4:

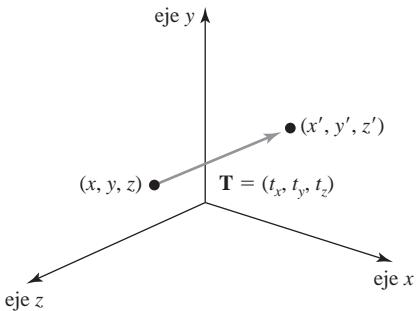
$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (5.71)$$

o,

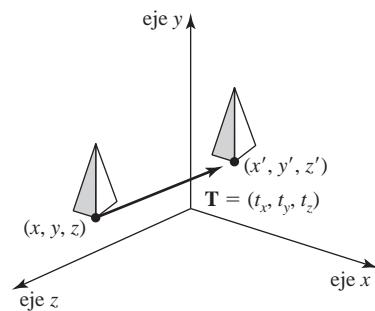
$$\mathbf{P}' = \mathbf{T} \cdot \mathbf{P} \quad (5.72)$$

Un objeto se traslada en tres dimensiones, transformando cada una de las posiciones de coordenadas de definición para el objeto, y reconstruyendo después el objeto en la nueva localización. Para un objeto representado como un conjunto de superficies poligonales, trasladamos cada vértice de cada superficie (Figura 5.35) y volvemos a mostrar las caras del polígono en las posiciones trasladadas.

El siguiente fragmento de programa ilustra la construcción de una matriz de traslación, dando unos parámetros traslacionales como conjunto de entrada. Para construir las matrices en estos procedimientos se usan métodos similares a los vistos en el programa de ejemplo de la Sección 5.4.



**FIGURA 5.34.** Movimiento de una posición de coordenadas con el vector de traslación  $\mathbf{T} = (t_x, t_y, t_z)$ .



**FIGURA 5.35.** Cambio de la posición de un objeto tridimensional usando el vector de traslación  $\mathbf{T}$ .

```

typedef GLfloat Matrix4x4 [4][4];

/* Construye la matriz identidad 4 por 4. */
void matrix4x4SetIdentity (Matrix4x4 matIdent4x4)
{
    GLint row, col;

    for (row = 0; row < 4; row++)
        for (col = 0; col < 4 ; col++)
            matIdent4x4 [row][col] = (row == col);

    void translate3D (GLfloat tx, GLfloat ty, GLfloat tz)
    {
        Matrix4x4 matTransl3D;

        /* Inicializa la matriz de traslación con la matriz identidad. */
        matrix4x4SetIdentity (matTransl3D);

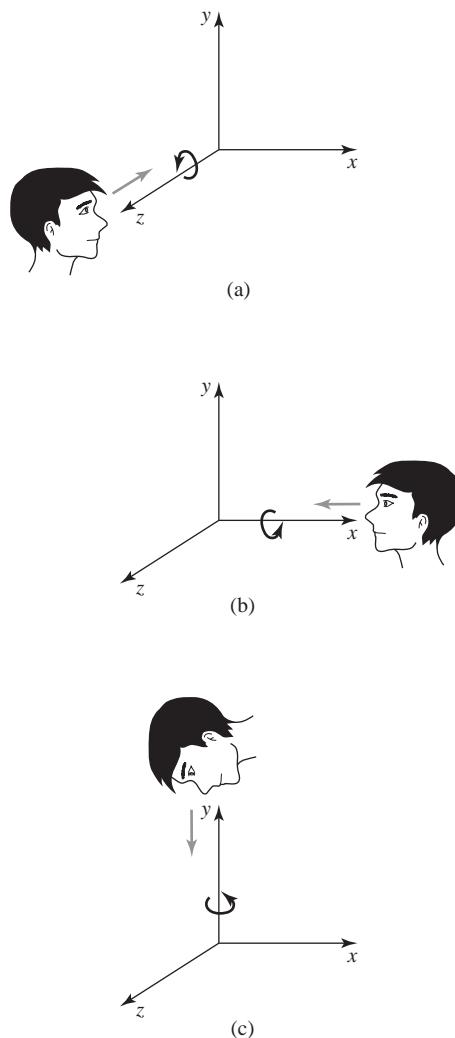
        matTransl3D [0][3] = tx;
        matTransl3D [1][3] = ty;
        matTransl3D [2][3] = tz;
    }
}

```

La inversa de una matriz de traslación tridimensional se obtiene usando los mismos procedimientos que se aplicaron en la traslación bidimensional. Esto es, negamos las distancias de traslación  $t_x$ ,  $t_y$  y  $t_z$ . Esto produce una traslación en la dirección opuesta, y el producto de la matriz de traslación y su inversa es la matriz identidad.

## 5.11 ROTACIONES TRIDIMENSIONALES

Podemos rotar un objeto sobre cualquier eje en el espacio, pero la forma más fácil de llevar a cabo una rotación de ejes, es aquella que es paralela a los ejes de coordenadas cartesianos. También, podemos usar combi-



**FIGURA 5.36.** Las rotaciones positivas alrededor de un eje de coordenadas se realizan en el sentido contrario a las agujas del reloj, cuando se está mirando a lo largo de la mitad positiva de los ejes con respecto al origen.

naciones de rotaciones de ejes de coordenadas (con las translaciones apropiadas) para especificar una rotación sobre cualquier otra línea en el espacio. Por tanto, primero consideramos las operaciones implicadas en las rotaciones de los ejes de coordenadas, y luego veremos los cálculos necesarios para otros ejes de rotación.

Por convenio, los ángulos de rotación positivos producen rotaciones en el sentido contrario al de las agujas del reloj sobre un eje de coordenadas, asumiendo que estamos mirando en la dirección negativa a lo largo de dicho eje de coordenadas (Figura 5.36). Esto concuerda con nuestra discusión anterior acerca de las rotaciones en dos dimensiones, donde las rotaciones positivas en el plano  $xy$  se hacen en sentido contrario a las agujas del reloj sobre un punto de pivot (un eje que es paralelo al eje  $z$ ).

### Rotaciones de ejes de coordenadas tridimensionales

Las ecuaciones de **rotación del eje-z** bidimensionales pueden extenderse fácilmente a tres dimensiones:

$$\begin{aligned}x' &= x \cos \theta - y \sin \theta \\y' &= x \sin \theta + y \cos \theta \\z' &= z\end{aligned}\tag{5.73}$$

El parámetro  $\theta$  especifica el ángulo de rotación sobre el eje  $z$ , y los valores de la coordenada- $z$  no se pueden cambiar con esta transformación. En la forma de coordenadas homogéneas, las ecuaciones para la rotación tridimensional del eje- $z$  son:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (5.74)$$

y las podemos escribir de manera más compacta como:

$$\mathbf{P}' = \mathbf{R}_z(\theta) \cdot \mathbf{P} \quad (5.75)$$

La Figura 5.37 ilustra la rotación de un objeto alrededor del eje  $z$ .

Las ecuaciones de transformación para rotaciones alrededor de los otros dos ejes de coordenadas pueden obtenerse con una permutación cíclica de los parámetros de coordenadas  $x$ ,  $y$  y  $z$  en las Ecuaciones 5.73:

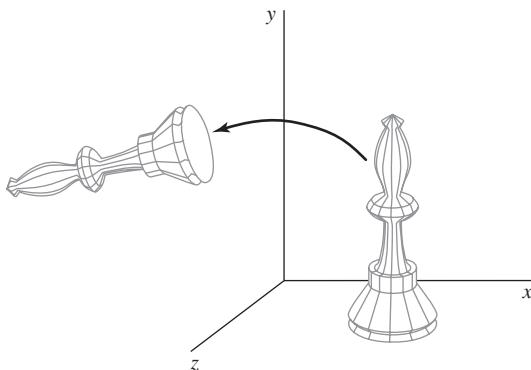
$$x \rightarrow y \rightarrow z \rightarrow x \quad (5.76)$$

Así, para obtener las transformaciones de rotación del eje- $x$  y el eje- $y$ , sustituimos cíclicamente  $x$  por  $y$ ,  $y$  por  $z$ , y  $z$  por  $x$ , tal como se ilustra en la Figura 5.38.

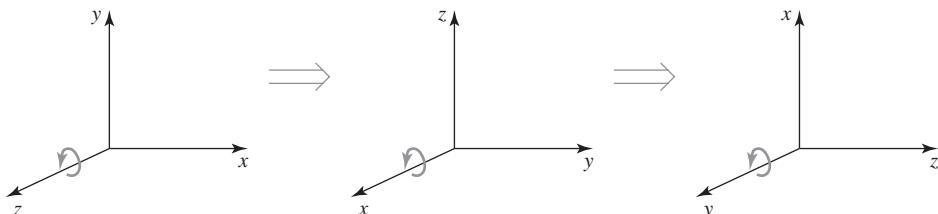
Sustituyendo las permutaciones de 5.76 por las Ecuaciones 5.73, obtenemos las ecuaciones para una **rotación del eje- $x$** :

$$\begin{aligned} y' &= y \cos \theta - z \sin \theta \\ z' &= y \sin \theta + z \cos \theta \\ x' &= x \end{aligned} \quad (5.77)$$

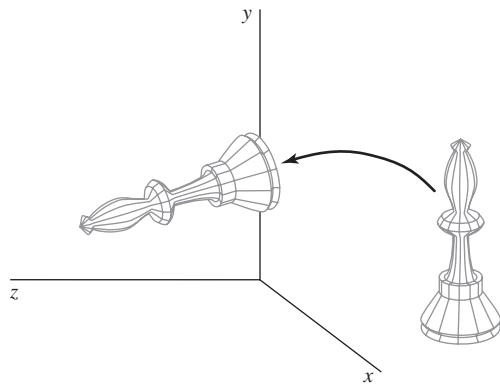
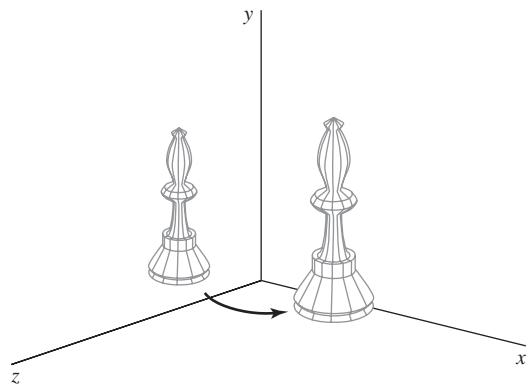
La rotación de un objeto alrededor del eje  $x$  se muestra en la Figura 5.39.



**FIGURA 5.37.** Rotación de un objeto sobre el eje  $z$ .



**FIGURA 5.38.** Permutación cíclica de los ejes de coordenadas cartesianas para producir los tres juegos de ecuaciones de rotación de ejes de coordenadas.

**FIGURA 5.39.** Rotación de un objeto alrededor del eje x.**FIGURA 5.40.** Rotación de un objeto alrededor del eje y.

Una permutación cíclica de coordenadas en las Ecuaciones 5.77 proporciona las ecuaciones de transformación para una **rotación del eje-y**:

$$\begin{aligned}z' &= z \cos \theta - x \sin \theta \\x' &= z \sin \theta + x \cos \theta \\y' &= y\end{aligned}\tag{5.78}$$

Un ejemplo de rotación del eje-y se muestra en la Figura 5.40.

Una matriz de rotación tridimensional inversa se obtiene de la misma manera que las rotaciones inversas en dos dimensiones. Basta con sustituir el ángulo  $\theta$  por  $-\theta$ . Los valores negativos para los ángulos de rotación generan rotaciones en el sentido de las agujas del reloj y la matriz identidad se obtiene multiplicando cualquier matriz de rotación por su inversa. Mientras sólo la función seno se vea afectada por el cambio de signo del ángulo de rotación, la matriz inversa puede obtenerse también intercambiando filas por columnas. Es decir, podemos calcular la inversa de cualquier matriz de rotación  $\mathbf{R}$  formando su traspuesta ( $\mathbf{R}^{-1} = \mathbf{R}^T$ ).

### Rotaciones tridimensionales generales

Una matriz de rotación, para cualquier eje que no coincide con un eje de coordenadas, puede realizarse como una transformación compuesta incluyendo combinaciones de traslaciones y rotaciones de ejes de coordenadas. Primero, movemos el eje de rotación designado dentro de uno de los ejes de coordenadas. Después aplicamos la matriz de rotación apropiada para ese eje de coordenadas. El último paso en la secuencia de transformación es devolver el eje de rotación a su posición original.

En el caso especial de que un objeto vaya a ser rotado alrededor de un eje que es paralelo a uno de los ejes de coordenadas, conseguimos la rotación deseada con la siguiente secuencia de transformaciones.

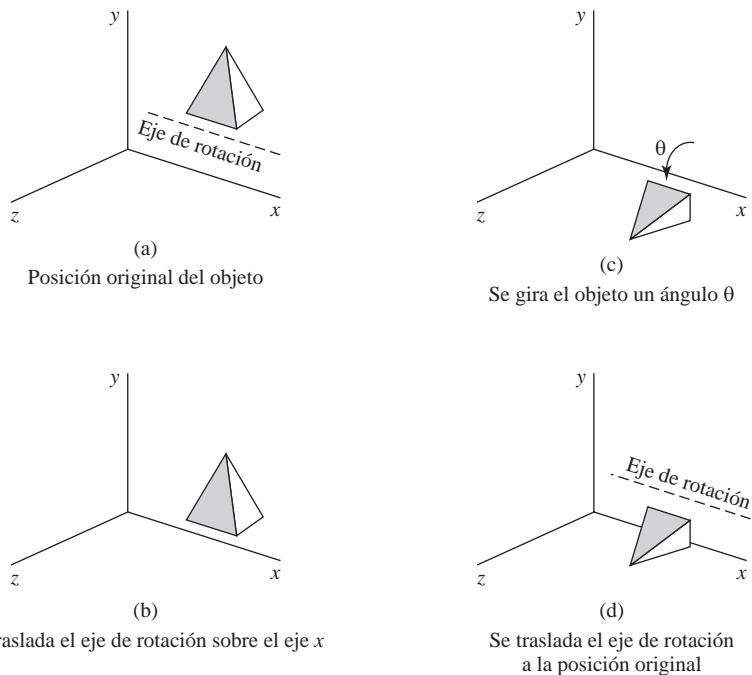
- (1) Traslado del objeto de tal forma que el eje de rotación coincida con el eje de coordenadas paralelo.
- (2) Se realiza la rotación especificada sobre ese eje.
- (3) Traslado del objeto de tal forma que el eje de rotación se mueva de nuevo a su posición original.

Los pasos de esta secuencia se ilustran en la Figura 5.41. Una posición de coordenadas  $\mathbf{P}$  se transforma con la secuencia mostrada en esta figura como:

$$\mathbf{P}' = \mathbf{T}^{-1} \cdot \mathbf{R}_x(\theta) \cdot \mathbf{T} \cdot \mathbf{P}\tag{5.79}$$

donde la matriz de rotación compuesta para la transformación es:

$$\mathbf{R}(\theta) = \mathbf{T}^{-1} \cdot \mathbf{R}_x(\theta) \cdot \mathbf{T}\tag{5.80}$$



**FIGURA 5.41.** Secuencia de transformaciones para la rotación de un objeto sobre un eje que es paralelo al eje  $x$ .

Esta matriz compuesta es de la misma forma que la secuencia de transformaciones bidimensional para la rotación sobre un eje paralelo al eje  $z$  (un punto de pivote que no está en el origen de coordenadas).

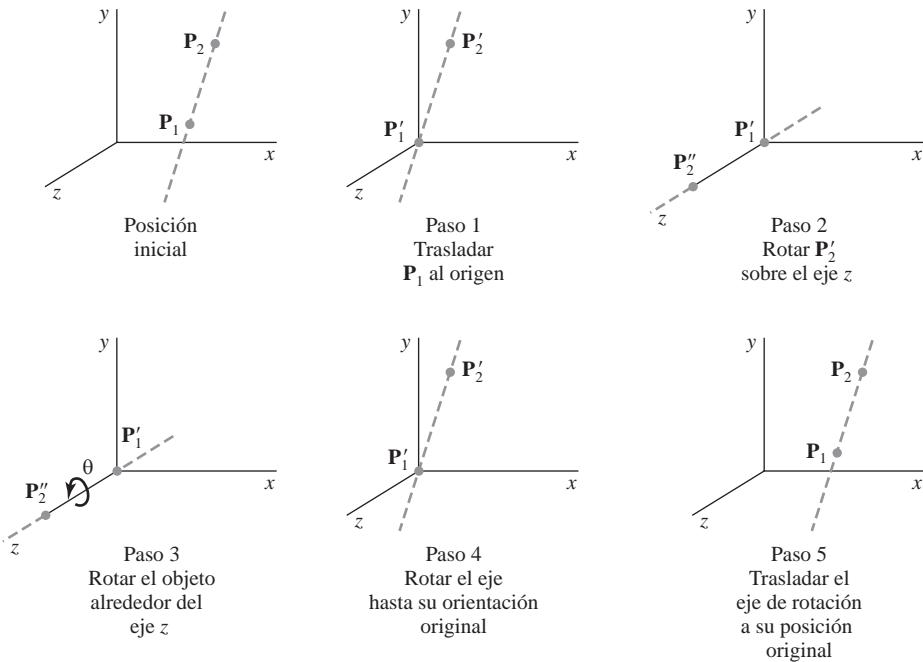
Cuando un objeto va a ser rotado sobre un eje que no es paralelo a uno de los ejes de coordenadas, necesitamos desarrollar algunas transformaciones adicionales. En este caso, también necesitamos rotaciones para alinear el eje de rotación con un eje de coordenadas seleccionado y luego devolver el eje de rotación a su orientación original. Dando las especificaciones para la rotación de ejes y del ángulo de rotación, podemos llevar a cabo la rotación requerida en cinco pasos:

- (1) Trasladar el objeto de tal forma que el eje de rotación pase a través del origen de coordenadas.
- (2) Rotar el objeto de forma que el eje de rotación coincida con uno de los ejes de coordenadas.
- (3) Realizar la rotación especificada sobre el eje de coordenadas seleccionado.
- (4) Aplicar las rotaciones inversas para devolver al eje de rotación su orientación original.
- (5) Aplicar la traslación inversa para devolver el eje de rotación a su posición espacial original.

Podemos transformar el eje de rotación dentro de cualquiera de los tres ejes de coordenadas. El eje- $z$  es a menudo una elección conveniente, y después consideraremos una secuencia de transformación usando la matriz de rotación del eje- $z$  (Figura 5.42).

Un eje de rotación puede definirse con dos posiciones de coordenadas, como en la Figura 5.43, o con un punto de coordenadas y ángulos de dirección (o cosenos de dirección) entre el eje de rotación y dos de los ejes de coordenadas. Asumimos que el eje de rotación se define con dos puntos, como se ilustra, y que la dirección de rotación va a ser en el sentido contrario a las agujas del reloj cuando se mira a lo largo del eje de  $\mathbf{P}_2$  a  $\mathbf{P}_1$ . Las componentes del vector del eje de rotación se calculan entonces del siguiente modo:

$$\begin{aligned} \mathbf{V} &= \mathbf{P}_2 - \mathbf{P}_1 \\ &= (x_2 - x_1, y_2 - y_1, z_2 - z_1) \end{aligned} \quad (5.81)$$



**FIGURA 5.42.** Cinco pasos de transformación para obtener una matriz compuesta para la rotación alrededor de un eje arbitrario, con el eje de rotación proyectado sobre el eje  $z$ .

Y el vector unidad del eje de rotación  $\mathbf{u}$  es:

$$\mathbf{u} = \frac{\mathbf{V}}{|\mathbf{V}|} = (a, b, c) \quad (5.82)$$

donde los componentes  $a$ ,  $b$  y  $c$  son los cosenos de dirección para la rotación del eje:

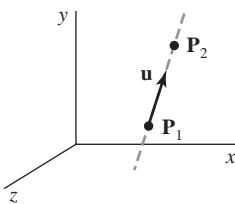
$$a = \frac{x_2 - x_1}{|\mathbf{V}|}, \quad b = \frac{y_2 - y_1}{|\mathbf{V}|}, \quad c = \frac{z_2 - z_1}{|\mathbf{V}|} \quad (5.83)$$

Si la rotación se va a realizar en sentido contrario (en el sentido de las agujas del reloj cuando se mira de  $\mathbf{P}_2$  a  $\mathbf{P}_1$ ) entonces deberíamos invertir el vector de eje  $\mathbf{V}$  y el vector unidad  $\mathbf{u}$  de tal manera que apuntasen en la dirección de  $\mathbf{P}_2$  a  $\mathbf{P}_1$ .

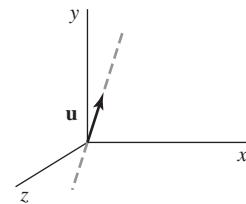
El primer paso en la secuencia de rotación es establecer la matriz de traslación que recoloca el eje de rotación, de tal forma que pasa a través del origen de coordenadas. Mientras queramos una rotación en sentido contrario a las agujas del reloj cuando miramos a lo largo del eje de  $\mathbf{P}_2$  a  $\mathbf{P}_1$  (Figura 5.43) movemos el punto  $\mathbf{P}_1$  al origen. (Si la rotación ha sido especificada en la dirección opuesta, deberemos mover  $\mathbf{P}_2$  al origen). Esta matriz de traslación es:

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & -x_1 \\ 0 & 1 & 0 & -y_1 \\ 0 & 0 & 1 & -z_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (5.84)$$

La cual recoloca el eje de rotación y el objeto tal y como se muestra en la Figura 5.44.



**FIGURA 5.43.** Un eje de rotación (línea de puntos) definido por los puntos  $P_1$  y  $P_2$ . La dirección para el vector de eje unidad  $\mathbf{u}$  se determina especificando la dirección de rotación.



**FIGURA 5.44.** Traslación de un eje de rotación al origen de coordenadas.

A continuación hay que formular las transformaciones que colocarán el eje de rotación sobre el eje  $z$ . Podemos usar las rotaciones del eje de coordenadas para llevar a cabo este alineamiento en dos pasos, y hay disponibles un cierto número de modos de desarrollar estos dos pasos. Para este ejemplo, primero giramos alrededor del eje  $x$  y después alrededor del eje  $y$ . La rotación del eje  $x$  obtiene el vector  $\mathbf{u}$  dentro del plano  $xz$ , y la rotación del eje  $y$  cambia la dirección de  $\mathbf{u}$  sobre el eje  $z$ . Estas dos rotaciones se ilustran en la Figura 5.45 para una posible orientación del vector  $\mathbf{u}$ .

Mientras los cálculos de rotación impliquen funciones seno y coseno, podemos usar operaciones de vectores estándar (Apéndice A) para obtener los elementos de las dos matrices de rotación. Puede utilizarse un producto escalar de vectores para determinar el término coseno y un producto vectorial de vectores para calcular el término seno.

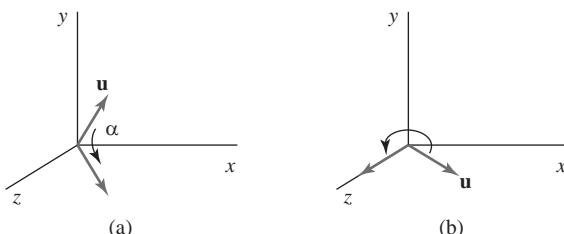
Establecemos la matriz de transformación para la rotación alrededor del eje  $x$ , determinando los valores para el seno y el coseno del ángulo de rotación necesario para obtener  $\mathbf{u}$  dentro del plano  $yz$ . Este ángulo de rotación es el ángulo entre la proyección de  $\mathbf{u}$  en el plano  $yz$  y el eje positivo de  $z$  (Figura 5.46). Si representamos la proyección de  $\mathbf{u}$  en el plano  $yz$  como el vector  $\mathbf{u}' = (0, b, c)$  entonces el coseno del ángulo de rotación  $\alpha$  puede determinarse a partir del producto escalar de  $\mathbf{u}'$  y el vector unitario  $\mathbf{u}_z$  a lo largo del eje  $z$ :

$$\cos \alpha = \frac{\mathbf{u}' \cdot \mathbf{u}_z}{|\mathbf{u}'||\mathbf{u}_z|} = \frac{c}{d} \quad (5.85)$$

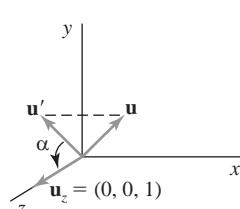
donde  $d$  es el módulo de  $\mathbf{u}'$ :

$$d = \sqrt{b^2 + c^2} \quad (5.86)$$

De manera similar, podemos determinar el seno de  $\alpha$  a partir del producto vectorial de  $\mathbf{u}'$  y  $\mathbf{u}_z$ . La forma independiente de las coordenadas de este producto vectorial es:



**FIGURA 5.45.** El vector unidad  $\mathbf{u}$  se gira sobre el eje  $x$  para dejarlo en el plano  $xz$  (a), y después se gira alrededor del eje  $y$  para alinearlo con el eje  $z$  (b).



**FIGURA 5.46.** La rotación de  $\mathbf{u}$  alrededor del eje  $x$  dentro del plano  $xz$  se lleva a cabo rotando  $\mathbf{u}'$  (que es la proyección de  $\mathbf{u}$  en el plano  $yz$ ) a través del ángulo  $\alpha$  sobre el eje  $z$ .

$$\mathbf{u}' \times \mathbf{u}_z = \mathbf{u}_x | \mathbf{u}' | | \mathbf{u}_z | \sin \alpha \quad (5.87)$$

y la forma cartesiana para el producto vectorial nos da:

$$\mathbf{u}' \times \mathbf{u}_z = \mathbf{u}_x \cdot b \quad (5.88)$$

Igualando las Ecuaciones 5.87 y 5.88, y sabiendo que  $|\mathbf{u}_z| = 1$  y  $|\mathbf{u}'| = d$ , tenemos:

$$d \sin \alpha = b$$

o,

$$\sin \alpha = \frac{b}{d} \quad (5.89)$$

Ahora que hemos determinado los valores para  $\cos \alpha$  y  $\sin \alpha$  en función de las componentes del vector  $\mathbf{u}$ , podemos establecer los elementos de la matriz para la rotación de este vector sobre el eje  $x$  y dentro del plano  $xz$ :

$$\mathbf{R}_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{c}{d} & -\frac{b}{d} & 0 \\ 0 & \frac{b}{d} & \frac{c}{d} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (5.90)$$

El siguiente paso en la formulación de la secuencia de transformaciones es determinar la matriz que cambiará en sentido contrario a las agujas del reloj el vector unidad en el plano  $xz$  alrededor del eje  $y$  sobre el eje positivo  $z$ . La Figura 5.47 muestra la orientación del vector unidad en el plano  $xz$ , resultante de la rotación sobre el eje  $x$ . Este vector, etiquetado como  $\mathbf{u}''$ , tiene el valor  $a$  para su componente  $x$ , mientras que la rotación sobre el eje  $x$  deja la componente  $x$  invariable. Su componente  $z$  es  $d$  (el módulo de  $\mathbf{u}'$ ) porque el vector  $\mathbf{u}'$  ha sido rotado sobre el eje  $z$ . Y la componente  $y$  de  $\mathbf{u}''$  es 0, porque ahora se encuentra en el plano  $xz$ . De nuevo podemos determinar el coseno del ángulo de rotación  $\beta$  a partir del producto escalar de los vectores unidad  $\mathbf{u}''$  y  $\mathbf{u}_z$ . Así,

$$\cos \beta = \frac{\mathbf{u}'' \cdot \mathbf{u}_z}{|\mathbf{u}''| |\mathbf{u}_z|} = d \quad (5.91)$$

mientras  $|\mathbf{u}_z| = |\mathbf{u}''| = 1$ . Comparando la forma independiente de las coordenadas del producto vectorial:

$$\mathbf{u}'' \times \mathbf{u}_z = \mathbf{u}_y | \mathbf{u}'' | | \mathbf{u}_z | \sin \beta \quad (5.92)$$

con la forma cartesiana:

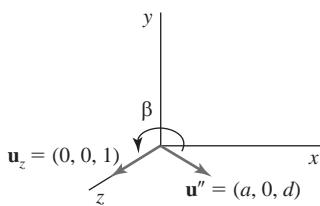
$$\mathbf{u}'' \times \mathbf{u}_z = \mathbf{u}_y (-a) \quad (5.93)$$

encontramos que,

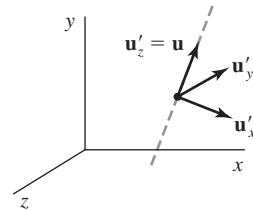
$$\sin \beta = -a \quad (5.94)$$

Por tanto, la matriz de transformación para la rotación de  $\mathbf{u}''$  sobre el eje  $y$  es

$$\mathbf{R}_y(\beta) = \begin{bmatrix} d & 0 & -a & 0 \\ 0 & 1 & 0 & 0 \\ a & 0 & d & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad (5.95)$$



**FIGURA 5.47.** Rotación de un vector unidad  $\mathbf{u}''$  (el vector  $\mathbf{u}$  después de la rotación dentro del plano  $xz$ ) sobre el eje  $y$ . Un ángulo de rotación positivo  $\beta$  alinea  $\mathbf{u}''$  con el vector  $\mathbf{u}_z$ .



**FIGURA 5.48.** Sistema de coordenadas local para un eje de rotación definido por el vector unidad  $\mathbf{u}$ .

Con las transformaciones de matrices 5.84, 5.90 y 5.95, alineamos el eje de rotación con el eje positivo  $z$ . El ángulo de rotación especificado  $\theta$  puede ahora aplicarse como una rotación alrededor del eje  $z$ :

$$\mathbf{R}_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (5.96)$$

Para completar la rotación requerida sobre el eje dado, necesitamos transformar el eje de rotación de vuelta a su posición original. Esto se hace aplicando la inversa de las transformaciones 5.84, 5.90 y 5.95. La matriz de transformación para la rotación sobre un eje arbitrario puede entonces expresarse como la composición de estas siete transformaciones individuales:

$$\mathbf{R}(\theta) = \mathbf{T}^{-1} \cdot \mathbf{R}_x^{-1}(\alpha) \cdot \mathbf{R}_y^{-1}(\beta) \cdot \mathbf{R}_z(\theta) \cdot \mathbf{R}_y(\beta) \cdot \mathbf{R}_x(\alpha) \cdot \mathbf{T} \quad (5.97)$$

Un método algo más rápido, pero quizás menos intuitivo, para obtener la matriz de rotación compuesta  $\mathbf{R}_y(\beta) \cdot \mathbf{R}_x(\alpha)$  es hacer uso del hecho de que la matriz compuesta para cualquier secuencia de rotaciones tridimensionales es de la forma:

$$\mathbf{R} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (5.98)$$

La submatriz de 3 por 3 superior izquierda de esta matriz es ortogonal. Esto significa que las filas (o las columnas) de esta submatriz forman un conjunto de vectores unidad ortogonales que son rotados con la matriz  $\mathbf{R}$  sobre los ejes  $x$ ,  $y$  y  $z$  respectivamente:

$$\mathbf{R} \cdot \begin{bmatrix} r_{11} \\ r_{12} \\ r_{13} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}, \quad \mathbf{R} \cdot \begin{bmatrix} r_{21} \\ r_{22} \\ r_{23} \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}, \quad \mathbf{R} \cdot \begin{bmatrix} r_{31} \\ r_{32} \\ r_{33} \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} \quad (5.99)$$

Por tanto, podemos establecer un sistema de coordenadas local con uno de sus ejes alineado con el eje de rotación. Entonces los vectores unidad para los tres ejes de coordenadas se usan para construir las columnas

de la matriz de rotación. Asumiendo que el eje de rotación no es paralelo a ninguno de los ejes de coordenadas, podríamos formar el siguiente conjunto de vectores unidad locales (Figura 5.48).

$$\begin{aligned}\mathbf{u}'_z &= \mathbf{u} \\ \mathbf{u}'_y &= \frac{\mathbf{u} \times \mathbf{u}_x}{|\mathbf{u} \times \mathbf{u}_x|} \\ \mathbf{u}'_x &= \mathbf{u}'_y \times \mathbf{u}'_z\end{aligned}\quad (5.100)$$

Si expresamos los elementos de los vectores unidad locales para la rotación de ejes como:

$$\begin{aligned}\mathbf{u}'_x &= (u'_{x1}, u'_{x2}, u'_{x3}) \\ \mathbf{u}'_y &= (u'_{y1}, u'_{y2}, u'_{y3}) \\ \mathbf{u}'_z &= (u'_{z1}, u'_{z2}, u'_{z3})\end{aligned}\quad (5.101)$$

entonces la matriz compuesta requerida, que es igual al producto de  $\mathbf{R}_y(\beta) \cdot \mathbf{R}_x(\alpha)$  es:

$$\mathbf{R} = \begin{bmatrix} u'_{x1} & u'_{x2} & u'_{x3} & 0 \\ u'_{y1} & u'_{y2} & u'_{y3} & 0 \\ u'_{z1} & u'_{z2} & u'_{z3} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}\quad (5.102)$$

Esta matriz transforma los vectores unidad  $\mathbf{u}'_x$ ,  $\mathbf{u}'_y$  y  $\mathbf{u}'_z$  en los ejes  $x$ ,  $y$  y  $z$ , respectivamente. Y esto alinea el eje de rotación con el eje  $z$ , porque  $\mathbf{u}'_z = \mathbf{u}$ .

## Métodos de cuaternios para rotaciones tridimensionales

Un método más eficiente para generar una rotación sobre un eje arbitrario seleccionado consiste en usar una representación cuaternia (Apéndice A) para la transformación de rotación. Los cuaternios, que son extensiones de los números complejos bidimensionales, son útiles en una serie de procedimientos de gráficos por computador, incluyendo la generación de objetos fractales. Requieren menos espacio de almacenamiento que las matrices de 4 por 4, y es más sencillo escribir procedimientos cuaternios para secuencias de transformaciones. Esto es particularmente importante en animaciones, las cuales a menudo requieren secuencias de movimientos complicadas e interpolación de movimiento entre dos posiciones de un objeto dadas.

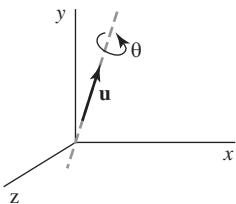
Una forma de caracterizar un cuaternionio es como un par ordenado, formada por una *parte escalar* y una *parte vectorial*:

$$q = (s, \mathbf{v})$$

también podemos pensar en un cuaternionio como en un número complejo de mayor orden con una parte real (la parte escalar) y tres partes complejas (los elementos del vector  $\mathbf{v}$ ). Una rotación sobre cualquier eje pasando por el origen de coordenadas, se lleva a cabo estableciendo primero un cuaternionio unidad con las partes escalar y vectorial:

$$s = \cos \frac{\theta}{2}, \quad \mathbf{v} = \mathbf{u} \sin \frac{\theta}{2} \quad (5.103)$$

donde  $\mathbf{u}$  es el vector unidad a lo largo del eje de rotación seleccionado y  $\theta$  es el ángulo de rotación especificado sobre este eje (Figura 5.49). Cualquier posición de un punto  $\mathbf{P}$  que va a ser rotado por este cuaternionio puede representarse en notación cuaternial como:



**FIGURA 5.49.** Parámetros de cuaternios unidad  $\theta$  y  $\mathbf{u}$  para la rotación sobre un eje especificado.

$$\mathbf{P} = (0, \mathbf{p})$$

con las coordenadas del punto como la parte vectorial  $\mathbf{p} = (x, y, z)$ . La rotación del punto entonces se lleva a cabo con la operación cuaternial:

$$\mathbf{P}' = q\mathbf{P}q^{-1} \quad (5.104)$$

donde  $q^{-1} = (s, -\mathbf{v})$  es la inversa del cuaternion unidad  $q$  con las partes escalar y vectorial dadas en las Ecuaciones 5.103. Esta transformación produce el siguiente nuevo cuaternion.

$$\mathbf{P}' = (0, \mathbf{p}') \quad (5.105)$$

El segundo término en este par ordenado es la posición del punto rotado  $\mathbf{p}'$ , el cual es evaluado con el producto escalar y el producto vectorial del vector como:

$$\mathbf{P}' = s^2\mathbf{p} + \mathbf{v}(\mathbf{p} \cdot \mathbf{v}) + 2s(\mathbf{v} \times \mathbf{p}) + \mathbf{v} \times (\mathbf{v} \times \mathbf{p}) \quad (5.106)$$

Los valores para los parámetros  $s$  y  $\mathbf{v}$  se obtienen de las expresiones 5.103. Muchos sistemas de gráficos por computadora usan implementaciones hardware eficientes de estos cálculos vectoriales para desarrollar rápidas rotaciones de objetos tridimensionales.

La transformación dada por la Ecuación 5.104 es equivalente a la rotación sobre un eje que pasa a través del origen de coordenadas. Esto es lo mismo que la secuencia de transformaciones de rotación de las Ecuaciones 5.97 que alinea el eje de rotación con el eje  $z$ , realizan una rotación alrededor de  $z$ , y después devuelven el eje de rotación a su orientación original en el origen de coordenadas.

Podemos evaluar los términos de las Ecuaciones 5.106 usando la definición de multiplicación de cuaternios que se da en el Apéndice A. Además, designando los componentes de la parte vectorial de  $q$  como  $\mathbf{v} = (a, b, c)$  obtenemos los elementos para la matriz de rotación compuesta  $\mathbf{R}_x^{-1}(\alpha) \cdot \mathbf{R}_y^{-1}(\beta) \cdot \mathbf{R}_z(\theta) \cdot \mathbf{R}_y(\beta) \cdot \mathbf{R}_x(\alpha)$  en la forma de 3 por 3 como:

$$\mathbf{M}_R(\theta) = \begin{bmatrix} 1-2b^2-2c^2 & 2ab-2sc & 2ac+2sb \\ 2ab+2sc & 1-2a^2-2c^2 & 2bc-2sa \\ 2ac-2sb & 2bc+2sa & 1-2a^2-2b^2 \end{bmatrix} \quad (5.107)$$

Los cálculos implicados en esta matriz pueden reducirse drásticamente sustituyendo los valores específicos por los parámetros  $a, b, c$  y  $s$ , y luego usando las siguientes identidades trigonométricas para simplificar los términos.

$$\cos^2 \frac{\theta}{2} - \sin^2 \frac{\theta}{2} = 1 - 2 \sin^2 \frac{\theta}{2} = \cos \theta, \quad 2 \cos \frac{\theta}{2} \sin \frac{\theta}{2} = \sin \theta$$

Así, podemos reescribir la matriz 5.107 como:

$$\mathbf{M}_R(\theta) = \begin{bmatrix} u_x^2(1-\cos \theta) + \cos \theta & u_x u_y (1-\cos \theta) - u_z \sin \theta & u_x u_z (1-\cos \theta) + u_y \sin \theta \\ u_y u_x (1-\cos \theta) + u_z \sin \theta & u_y^2(1-\cos \theta) + \cos \theta & u_y u_z (1-\cos \theta) - u_x \sin \theta \\ u_z u_x (1-\cos \theta) - u_y \sin \theta & u_z u_y (1-\cos \theta) + u_x \sin \theta & u_z^2(1-\cos \theta) + \cos \theta \end{bmatrix} \quad (5.108)$$

donde  $\mathbf{u}_x$ ,  $\mathbf{u}_y$  y  $\mathbf{u}_z$  son los componentes del vector del eje unidad  $\mathbf{u}$ .

Para completar la secuencia de transformaciones para rotar sobre un eje de rotación situado aleatoriamente, necesitamos incluir las traslaciones que mueven el eje de rotación al eje de coordenadas y lo devuelven a su posición original. Así, la expresión completa de la rotación cuaternal, correspondiente a la Ecuación 5.97, es:

$$\mathbf{R}(\theta) = \mathbf{T}^{-1} \cdot \mathbf{M}_R \cdot \mathbf{T} \quad (5.109)$$

Como ejemplo, podemos desarrollar una rotación sobre el eje  $z$  estableciendo un vector para el eje de rotación  $\mathbf{u}$  al vector unidad  $(0,0,1)$ . Sustituyendo los componentes de este vector en la matriz 5.108, obtenemos una versión de la matriz de rotación del eje- $z$  de 3 por 3,  $\mathbf{R}_z(\theta)$ , en las ecuaciones de transformación 5.74. De manera similar, sustituyendo los valores de rotación del cuaternion unidad en las ecuaciones de transformación 5.104, se obtienen los valores de las coordenadas rotadas de las Ecuaciones 5.73.

En el siguiente código, se dan ejemplos de procedimientos que podrían usarse para construir una matriz de rotación tridimensional. La representación cuaternal de la Ecuación 5.109 se usa para establecer los elementos de la matriz para una rotación tridimensional general.

```

class wcPt3D {
public:
    GLfloat x, y, z;
};

typedef float Matrix4x4 [4][4];

Matrix4x4 matRot;

/* Construye la matriz identidad 4 por 4. */
void matrix4x4SetIdentity (Matrix4x4 matIdent4x4)
{
    GLint row, col;

    for (row = 0; row < 4; row++)
        for (col = 0; col < 4 ; col++)
            matIdent4x4 [row][col] = (row == col);
}

/* Premultiplica la matriz m1 por la matriz m2, almacena el resultado en m2. */
void matrix4x4PreMultiply (Matrix4x4 m1, Matrix4x4 m2)
{
    GLint row, col;
    Matrix4x4 matTemp;

    for (row = 0; row < 4; row++)
        for (col = 0; col < 4 ; col++)
            matTemp [row][col] = m1 [row][0] * m2 [0][col] + m1 [row][1] *
                m2 [1][col] + m1 [row][2] * m2 [2][col] +
                m1 [row][3] * m2 [3][col];

    for (row = 0; row < 4; row++)
        for (col = 0; col < 4; col++)
            m2 [row][col] = matTemp [row][col];
}

void translate3D (GLfloat tx, GLfloat ty, GLfloat tz)

```

```

{
    Matrix4x4 matTransl3D;

    /* Inicializa la matriz de translación con la matriz identidad. */
    matrix4x4SetIdentity (matTransl3D);

    matTransl3D [0][3] = tx;
    matTransl3D [1][3] = ty;
    matTransl3D [2][3] = tz;

    /* Concatena la matriz de translación con matRot. */
    matrix4x4PreMultiply (matTransl3D, matRot);

}

void rotate3D (wcPt3D p1, wcPt3D p2, GLfloat radianAngle)
{

    Matrix4x4 matQuaternionRot;
    GLfloat axisVectLength = sqrt ((p2.x - p1.x) * (p2.x - p1.x) +
                                    (p2.y - p1.y) * (p2.y - p1.y) +
                                    (p2.z - p1.z) * (p2.z - p1.z));
    GLfloat cosA = cos (radianAngle);
    GLfloat oneC = 1 - cosA;
    GLfloat sinA = sin (radianAngle);
    GLfloat ux = (p2.x - p1.x) / axisVectLength;
    GLfloat uy = (p2.y - p1.y) / axisVectLength;
    GLfloat uz = (p2.z - p1.z) / axisVectLength;

    /* Configura la matriz de translación para mover p1 al origen. */
    translate3D (-p1.x, -p1.y, -p1.z);

    /* Inicializa matQuaternionRot con la matriz identidad. */
    matrix4x4SetIdentity (matQuaternionRot);

    matQuaternionRot [0][0] = ux*ux*oneC + cosA;
    matQuaternionRot [0][1] = ux*uy*oneC - uz*sinA;
    matQuaternionRot [0][2] = ux*uz*oneC + uy*sinA;
    matQuaternionRot [1][0] = uy*ux*oneC + uz*sinA;
    matQuaternionRot [1][1] = uy*uy*oneC + cosA;
    matQuaternionRot [1][2] = uy*uz*oneC - ux*sinA;
    matQuaternionRot [2][0] = uz*ux*oneC - uy*sinA;
    matQuaternionRot [2][1] = uz*uy*oneC + ux*sinA;
    matQuaternionRot [2][2] = uz*uz*oneC + cosA;

    /* Combina matQuaternionRot con la matriz de translación. */
    matrix4x4PreMultiply (matQuaternionRot, matRot);

    /* Configura la matriz matTransl3D inversa y la concatena con el
     * producto de las dos matrices anteriores.*/
}

```

```

translate3D (p1.x, p1.y, p1.z);
}

void displayFcn (void)
{
    /* Introducir los parámetros de rotación. */

    /* Inicializa matRot con la matriz identidad: */
    matrix4x4SetIdentity (matRot);

    /* Pasar los parámetros de rotación al procedimiento rotate3D. */

    /* Mostrar el objeto girado. */
}

```

## 5.12 CAMBIO DE ESCALA TRIDIMENSIONAL

La expresión de la matriz para la transformación de cambio de escala tridimensional de una posición  $\mathbf{P} = (x, y, z)$  relativa al origen de coordenadas es una simple extensión de un cambio de escala bidimensional. Simplemente incluimos el parámetro para el cambio de escala de la coordenada- $z$  en la matriz de transformación:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (5.110)$$

La transformación de cambio de escala tridimensional para una posición de un punto puede representarse como:

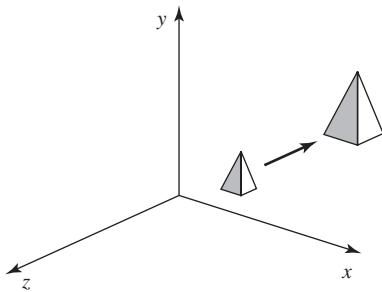
$$\mathbf{P}' = \mathbf{S} \cdot \mathbf{P} \quad (5.111)$$

donde a los parámetros de escala  $s_x$ ,  $s_y$  y  $s_z$ , se les asignan cualesquiera valores positivos. Las expresiones explícitas para la transformación de cambio de escala respecto del origen son:

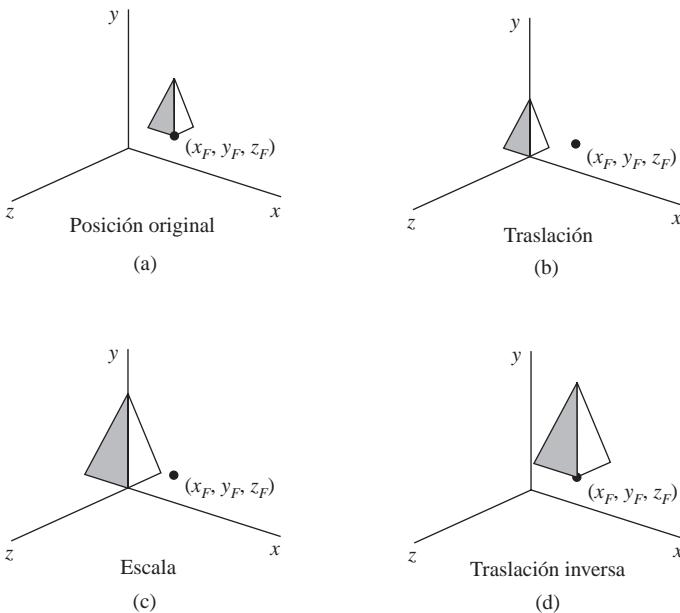
$$x' = x \cdot s_x, \quad y' = y \cdot s_y, \quad z' = z \cdot s_z \quad (5.112)$$

Cambiar la escala de un objeto con la transformación dada por las Ecuaciones 5.110 cambia la posición del objeto respecto del origen de coordenadas. Un valor del parámetro superior a 1 mueve un punto alejándolo del origen en la correspondiente dirección de coordenadas. De la misma manera, un valor de parámetro inferior a 1 mueve un punto acercándolo al origen en esa dirección de coordenadas. Además, si los parámetros de escala no son todos iguales, las dimensiones relativas del objeto transformado cambian. La forma original de un objeto se conserva realizando un *cambio de escala uniforme*:  $s_x = s_y = s_z$ . El resultado de aplicar un cambio de escala uniforme sobre un objeto con cada parámetro de escala igual a 2 se ilustra en la Figura 5.50.

Dado que algunos paquetes gráficos sólo ofrecen una rutina que realiza cambios de escala respecto al origen de coordenadas, podemos construir siempre una transformación de cambio de escala con respecto a cualquier *posición fija* seleccionada ( $x_f, y_f, z_f$ ) usando la siguiente secuencia de transformaciones.



**FIGURA 5.50.** Duplicar el tamaño de un objeto con la transformación 5.110 también mueve el objeto alejándolo del origen.



**FIGURA 5.51.** Secuencia de transformaciones para el cambio de escala de un objeto respecto a un punto fijo seleccionado usando la Ecuación 5.110.

- (1) Trasladar el punto fijo al origen.
- (2) Aplicar la transformación de cambio de escala respecto al origen de coordenadas usando la Ecuación 5.110.
- (3) Trasladar el punto fijo de vuelta a su posición original.

Esta secuencia de transformaciones se muestra en la Figura 5.51. La representación de la matriz para un punto fijo de cambio de escala arbitrario puede expresarse como la concatenación de estas transformaciones de traslación - cambio de escala - traslación:

$$\mathbf{T}(x_f, y_f, z_f) \cdot \mathbf{S}(s_x, s_y, s_z) \cdot \mathbf{T}(-x_f, -y_f, -z_f) = \begin{bmatrix} s_x & 0 & 0 & (1-s_x)x_f \\ 0 & s_y & 0 & (1-s_y)y_f \\ 0 & 0 & s_z & (1-s_z)z_f \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (5.113)$$

Podemos establecer procedimientos programados para la construcción de matrices de escalado tridimensional, usando tanto la secuencia traslación-escalado-traslación como la incorporación directa de las coordenadas del punto fijo. En el siguiente código de ejemplo, demostramos una construcción directa de una matriz de escalado tridimensional relativa a un punto fijo seleccionado usando los cálculos de la Ecuación 5.113.

```
class wcPt3D
{
    private:
        GLfloat x, y, z;

    public:
        /* Constructor predeterminado:
         * Inicializa la posición en (0.0, 0.0, 0.0).
         */
        wcPt3D () {
            x = y = z = 0.0;
        }

        void setCoords (GLfloat xCoord, GLfloat yCoord, GLfloat zCoord) {
            x = xCoord;
            y = yCoord;
            z = zCoord;
        }

        GLfloat getx () const {
            return x;
        }

        GLfloat gety () const {
            return y;
        }

        GLfloat getz () const {
            return z;
        }
    };

    typedef float Matrix4x4 [4][4];

    void scale3D (GLfloat sx, GLfloat sy, GLfloat sz, wcPt3D fixedPt)
    {

        Matrix4x4 matScale3D;

        /* Inicializa la matriz de cambio de escala con la matriz identidad. */
        matrix4x4SetIdentity (matScale3D);

        matScale3D [0][0] = sx;
        matScale3D [0][3] = (1 - sx) * fixedPt.getx ();
        matScale3D [1][1] = sy;
        matScale3D [1][3] = (1 - sy) * fixedPt.gety ();
        matScale3D [2][2] = sz;
        matScale3D [2][3] = (1 - sz) * fixedPt.getz ();
    }
}
```

Una inversa, la matriz de cambio de escala tridimensional se establece para la Ecuación 5.110 o la Ecuación 5.113 sustituyendo cada parámetro de escala ( $s_x$ ,  $s_y$  y  $s_z$ ) por su recíproco. Pero esta transformación inversa es indefinida si a cualquier parámetro de escala se le asigna el valor 0. La matriz inversa genera una transformación de cambio de escala opuesta y la concatenación de una matriz de cambio de escala tridimensional con su inversa da lugar a la matriz identidad.

## 5.13 TRANSFORMACIONES COMPUESTAS TRIDIMENSIONALES

---

Al igual que en las transformaciones bidimensionales, formamos una transformación tridimensional multiplicando las representaciones matriciales para las operaciones individuales en la secuencia de transformación. Cualquiera de las secuencias de transformación bidimensionales vistas en la Sección 5.4, tales como el cambio de escala en direcciones sin coordenadas, puede llevarse a cabo en el espacio tridimensional.

Podemos implementar una secuencia de transformaciones concatenando las matrices individuales de derecha a izquierda o de izquierda a derecha, dependiendo del orden en el que las representaciones de matrices se hayan especificado. Por supuesto, el término del extremo derecho en un producto de matrices es siempre la primera transformación que hay que aplicar al objeto y el término situado más a la izquierda es siempre la última transformación. Necesitamos usar este orden para el producto de matrices porque las posiciones de coordenadas se representan como vectores columna de cuatro elementos, los cuales se premultiplican con la matriz de transformación compuesta de 4 por 4.

El siguiente programa ofrece rutinas de ejemplo para la construcción de matrices de transformación compuestas tridimensionales. Las tres transformaciones geométricas básicas se combinan en un orden determinado para producir una única matriz compuesta, la cual se inicializa con la matriz identidad. En este ejemplo, primero se realiza la rotación, luego el cambio de escala y, por último, la traslación. Elegimos una evaluación de izquierda a derecha de la matriz compuesta, de tal forma que las transformaciones se van llamando en el orden en el que van a ser aplicadas. Así, a medida que se construye cada matriz, se concatena por el lado izquierdo de la matriz compuesta actual para formar el producto actualizado de la matriz.

```

class wcPt3D {
public:
    GLfloat x, y, z;
};

typedef GLfloat Matrix4x4 [4][4];

Matrix4x4 matComposite;

/* Construye la matriz identidad 4 por 4. */
void matrix4x4SetIdentity (Matrix4x4 matIdent4x4)
{
    GLint row, col;
    for (row = 0; row < 4; row++)
        for (col = 0; col < 4 ; col++)
            matIdent4x4 [row][col] = (row == col);
}

/* Premultiplica la matriz m1 por la matriz m2, almacena el resultado en m2. */
void matrix4x4PreMultiply (Matrix4x4 m1, Matrix4x4 m2)
{
}

```

```

GLint row, col;
Matrix4x4 matTemp;

for (row = 0; row < 4; row++)
    for (col = 0; col < 4 ; col++)
        matTemp [row][col] = m1 [row][0] * m2 [0][col] + m1 [row][1] *
                            m2 [1][col] + m1 [row][2] * m2 [2][col] +
                            m1 [row][3] * m2 [3][col];
    for (row = 0; row < 4; row++)
        for (col = 0; col < 4; col++)
            m2 [row][col] = matTemp [row][col];
}

/* Procedimiento para generar la matriz de traslación 3D. */
void translate3D (GLfloat tx, GLfloat ty, GLfloat tz)
{
    Matrix4x4 matTransl3D;

    /* Inicializa la matriz de traslación con la matriz identidad. */
    matrix4x4SetIdentity (matTransl3D);

    matTransl3D [0][3] = tx;
    matTransl3D [1][3] = ty;
    matTransl3D [2][3] = tz;

    /* Concatena matTransl3D con la matriz compuesta. */
    matrix4x4PreMultiply (matTransl3D, matComposite);
}

/* Procedimiento para generar una matriz de rotación de cuaternios. */
void rotate3D (wcPt3D p1, wcPt3D p2, GLfloat radianAngle)
{
    Matrix4x4 matQuatRot;

    float axisVectLength = sqrt ((p2.x - p1.x) * (p2.x - p1.x) +
                                (p2.y - p1.y) * (p2.y - p1.y) +
                                (p2.z - p1.z) * (p2.z - p1.z));
    float cosA = cosf (radianAngle);
    float oneC = 1 - cosA;
    float sinA = sinf (radianAngle);
    float ux = (p2.x - p1.x) / axisVectLength;
    float uy = (p2.y - p1.y) / axisVectLength;
    float uz = (p2.z - p1.z) / axisVectLength;

    /* Define la matriz de traslación para mover p1 al origen
     * y concatena la matriz de traslación con matComposite. */
    translate3D (-p1.x, -p1.y, -p1.z);

    /* Inicializa matQuatRot con la matriz identidad. */
    matrix4x4SetIdentity (matQuatRot);

    matQuatRot [0][0] = ux*ux*oneC + cosA;
}

```

```

matQuatRot [0][1] = ux*uy*oneC - uz*sina;
matQuatRot [0][2] = ux*uz*oneC + uy*sina;
matQuatRot [1][0] = uy*ux*oneC + uz*sina;
matQuatRot [1][1] = uy*uy*oneC + cosa;
matQuatRot [1][2] = uy*uz*oneC - ux*sina;
matQuatRot [2][0] = uz*ux*oneC - uy*sina;
matQuatRot [2][1] = uz*uy*oneC + ux*sina;
matQuatRot [2][2] = uz*uz*oneC + cosa;

/* Concatena matQuatRot con la matriz compuesta. */
matrix4x4PreMultiply (matQuatRot, matComposite);

/* Construye la matriz de traslación inversa para p1 y la
 * concatena con la matriz compuesta. */
translate3D (p1.x, p1.y, p1.z);
}

/* Procedimiento para generar la matriz de cambio de escala 3D. */
void scale3D (GLfloat sx, GLfloat sy, GLfloat sz, wcPt3D fixedPt)
{
    Matrix4x4 matScale3D;

    /* Inicializa la matriz de cambio de escala con la matriz identidad. */
    matrix4x4SetIdentity (matScale3D);

    matScale3D [0][0] = sx;
    matScale3D [0][3] = (1 - sx) * fixedPt.x;
    matScale3D [1][1] = sy;
    matScale3D [1][3] = (1 - sy) * fixedPt.y;
    matScale3D [2][2] = sz;
    matScale3D [2][3] = (1 - sz) * fixedPt.z;

    /* Concatena matScale3D con la matriz compuesta. */
    matrix4x4PreMultiply (matScale3D, matComposite);
}

void displayFcn (void)
{
    /* Introducir la descripción del objeto. */
    /* Introducir los parámetros de traslación, rotación y cambio de escala. */
    /* Establecer las rutinas de transformación de visualización 3D. */
    /* Inicializar la matriz matComposite con la matriz identidad: */
    matrix4x4SetIdentity (matComposite);

    /* Invocar a las rutinas de transformación en el orden en que
     * van a ser aplicadas: */
    rotate3D (p1, p2, radianAngle); // Primera transformación: rotación.
    scale3D (sx, sy, sz, fixedPt); // Segunda transformación: cambio de escala.
    translate3D (tx, ty, tz); // Transformación final: traslación.

    /* Llamada a las rutinas que muestran los objetos transformados. */
}

```

## 5.14 OTRAS TRANSFORMACIONES TRIDIMENSIONALES

---

Además de la traslación, la rotación y el cambio de escala, las otras transformaciones vistas para las aplicaciones bidimensionales son también útiles en muchas situaciones tridimensionales. Estas transformaciones adicionales incluyen la reflexión, la inclinación y las transformaciones entre sistemas de coordenadas de referencia.

### Reflexiones tridimensionales

Una reflexión en un espacio tridimensional puede desarrollarse con respecto a un *eje de reflexión* determinado o con respecto a un *plano de reflexión*. En general, las matrices de reflexión tridimensionales se establecen de manera similar a las mismas de dos dimensiones. Las reflexiones respecto a un eje dado son equivalentes a rotaciones de 180° sobre dicho eje. Las reflexiones con respecto al plano son equivalentes a rotaciones de 180° en un espacio cuatridimensional. Cuando el plano de reflexión es un plano de coordenadas ( $xy$ ,  $xz$ , o  $yz$ ) podemos pensar en la transformación como en una conversión entre un sistema a izquierdas y un sistema a derechas (Apéndice A).

Un ejemplo de una reflexión que convierte especificaciones de coordenadas de un sistema a derechas a un sistema a izquierdas (o viceversa) se muestra en la Figura 5.52. Esta transformación cambia el signo de las coordenadas  $z$ , dejando los valores para las coordenadas  $x$  e  $y$  invariables. La representación de la matriz para esta reflexión relativa al plano  $xy$  es:

$$M_{z\text{reflejada}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (5.114)$$

Las matrices de transformación para invertir las coordenadas  $x$  o las coordenadas  $y$  se definen de manera similar, como reflexiones relativas al plano  $yz$  o al plano  $xz$ , respectivamente. Las reflexiones sobre otros planos pueden obtenerse como una combinación de rotaciones y reflexiones en el plano de coordenadas.

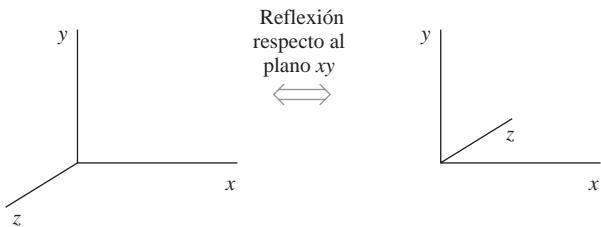
### Inclinaciones tridimensionales

Estas transformaciones pueden usarse para modificar las formas de los objetos, tal como se hace en las aplicaciones bidimensionales. Además, se aplican en transformaciones de vistas para proyecciones de perspectivas. Las transformaciones de inclinación relativas a los ejes  $x$  e  $y$  son las mismas que las vistas en la Sección 5.5. Para aplicaciones tridimensionales, podemos generar inclinaciones respecto al eje  $z$ .

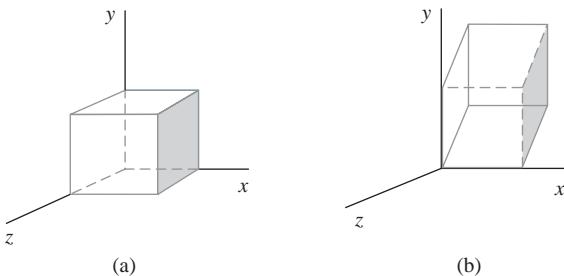
Una transformación de inclinación general en el eje  $z$  relativa a una posición de referencia determinada se obtiene con la siguiente matriz.

$$M_{z\text{inclinación}} = \begin{bmatrix} 1 & 0 & sh_{zx} & -sh_{zx} \cdot z_{\text{ref}} \\ 0 & 1 & sh_{zy} & -sh_{zy} \cdot z_{\text{ref}} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (5.115)$$

A los parámetros de inclinación  $sh_{zx}$  y  $sh_{zy}$  se les puede asignar cualquier valor real. El efecto de esta matriz de transformación es alterar los valores de las coordenadas  $x$  e  $y$  en una cantidad que sea proporcional a la distancia desde  $z_{\text{ref}}$ , mientras que se deja la coordenada  $z$  invariable. Las áreas del plano que son perpendiculares al eje  $z$  se varían así en una cantidad igual a  $z - z_{\text{ref}}$ . Un ejemplo de los efectos de esta matriz de inclinación en un cubo se muestra en la Figura 5.53 para los valores de inclinación  $sh_{zx} = sh_{zy} = 1$  y una posición de referencia  $z_{\text{ref}} = 0$ . Las matrices de transformación tridimensional para una inclinación en el eje  $x$  y



**FIGURA 5.52.** Conversión de especificaciones de coordenadas entre un sistema a derechas y otro a izquierdas puede llevarse a cabo con la transformación de reflexión.



**FIGURA 5.53.** Un cubo (a) inclinado respecto al origen (b) mediante la matriz de transformación 5.115, con  $sh_{zx} = sh_{zy} = 1$ .

una inclinación del eje  $y$  es similar a las matrices bidimensionales. Simplemente necesitamos añadir una fila o una columna para los parámetros de inclinación de las coordenadas  $z$ .

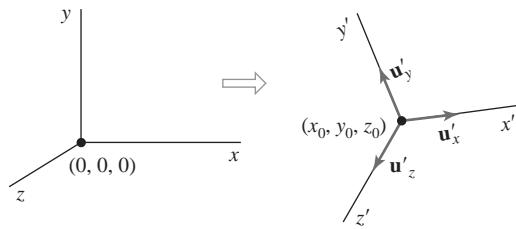
## 5.15 TRANSFORMACIONES ENTRE SISTEMAS DE COORDENADAS TRIDIMENSIONALES

En la Sección 5.8 examinamos las operaciones necesarias para transferir una descripción de una escena bidimensional desde un sistema de referencia a otro. Las transformaciones de sistemas de coordenadas se emplean en paquetes de gráficos por computadora para construir (modelar) escenas e implementar rutinas de visualización tanto para aplicaciones bidimensionales como tridimensionales. Como vimos en la Sección 5.8, una matriz de transformación para transferir la descripción de una escena bidimensional de un sistema de coordenadas a otro se construye con operaciones para superponer los ejes de coordenadas de los dos sistemas. Los mismos procedimientos son aplicables a transformaciones de escenas tridimensionales.

De nuevo consideramos sólo marcos de referencia cartesianos y asumimos que un sistema  $x'y'z'$  se define con respecto a un sistema  $xyz$ . Para transferir las descripciones de las coordenadas  $xyz$  al sistema  $x'y'z'$ , primero establecemos una traslación que lleva el origen de coordenadas  $x'y'z'$  a la posición del origen  $xyz$ . A continuación se realiza la secuencia de rotaciones que alinea los correspondientes ejes de coordenadas. Si se usan diferentes escalas en los dos sistemas de coordenadas, una transformación de cambio de escala también es necesaria para compensar las diferencias en los intervalos de coordenadas.

La Figura 5.54 muestra un sistema de coordenadas  $x'y'z'$  con origen en  $(x_0, y_0, z_0)$  y vectores unidad de ejes definidos respecto a un marco de referencia  $xyz$ . El origen de coordenadas del sistema  $x'y'z'$  se mueve hasta hacerlo coincidir con el origen  $xyz$  usando la matriz de traslación  $T(-x_0, -y_0, -z_0)$ . Podemos usar los vectores unidad de ejes para formar la matriz de rotación del eje de coordenadas:

$$\mathbf{R} = \begin{bmatrix} u'_{x1} & u'_{x2} & u'_{x3} & 0 \\ u'_{y1} & u'_{y2} & u'_{y3} & 0 \\ u'_{z1} & u'_{z2} & u'_{z3} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (5.116)$$



**FIGURA 5.54.** Un sistema de coordenadas  $x'y'z'$  definido dentro de un sistema  $xyz$ . Una descripción de una escena es transferida a la nueva referencia de coordenadas usando una secuencia de transformación que superpone el marco  $x'y'z'$  sobre los ejes  $xyz$ .

la cual transforma los vectores unidad  $\mathbf{u}'_x$ ,  $\mathbf{u}'_y$  y  $\mathbf{u}'_z$  dentro de los ejes  $x$ ,  $y$  y  $z$ , respectivamente. La secuencia completa de transformaciones de coordenadas se da entonces con la matriz compuesta  $\mathbf{R} \cdot \mathbf{T}$ . Esta matriz transforma correctamente las descripciones de coordenadas de un sistema cartesiano a otro, incluso si un sistema es a izquierdas y el otro a derechas.

## 5.16 TRANSFORMACIONES AFINES

Una transformación de coordenadas de la forma:

$$\begin{aligned} x' &= a_{xx}x + a_{xy}y + a_{xz}z + b_x \\ y' &= a_{yx}x + a_{yy}y + a_{yz}z + b_y \\ z' &= a_{zx}x + a_{zy}y + a_{zz}z + b_z \end{aligned} \quad (5.117)$$

se llama **transformación afín**. Cada una de las coordenadas transformadas  $x'$ ,  $y'$  y  $z'$ , es una función lineal del origen de coordenadas  $x$ ,  $y$  y  $z$ , y los parámetros  $a_{ij}$  y  $b_k$  son constantes determinadas por el tipo de transformación. Las transformaciones afines (en dos dimensiones, tres dimensiones o más dimensiones) tienen las propiedades generales de que las líneas paralelas se transforman en líneas paralelas y los puntos finitos se asignan a puntos finitos.

Traslación, rotación, cambio de escala, reflexión e inclinación son ejemplos de transformaciones afines. Siempre podemos expresar cualquier transformación afín como alguna composición de estas cinco transformaciones. Otro ejemplo de transformación afín es la conversión de descripciones de coordenadas para una escena de un sistema de referencia a otro, mientras esta transformación pueda describirse como combinación de traslación y rotación. Una transformación afín que sólo implica transformación, rotación y reflexión conserva los ángulos y longitudes, así como líneas paralelas. Para cada una de estas tres transformaciones, la longitud de la línea y el ángulo entre cualesquiera dos líneas se mantiene igual después de la transformación.

## 5.17 FUNCIONES DE TRANSFORMACIONES GEOMÉTRICAS EN OpenGL

En la biblioteca del núcleo (*core*) de OpenGL hay disponible una función distinta para cada una de las transformaciones geométricas, y todas las transformaciones se especifican en tres dimensiones. Para llevar a cabo una traslación, invocamos la rutina de traslación y establecemos los componentes para el vector de traslación tridimensional. En la función de rotación, especificamos el ángulo y la orientación para el eje de rotación que hace la intersección con el origen de coordenadas. Y una función de cambio de escala se usa para establecer las tres coordenadas de los factores de escala relativos al origen de coordenadas. En cada caso, la rutina de transformación usa una matriz de 4 por 4 que se aplica a las coordenadas de los objetos a los que se hace referencia después de la llamada a la transformación.

## Transformaciones básicas en OpenGL

Una matriz de traslación de 4 por 4 se construye con la siguiente rutina:

```
glTranslate* (tx, ty, tz);
```

A los parámetros de traslación `tx`, `ty` y `tz` se les puede asignar cualquier valor real, y el código del sufijo a ser anexado es o bien `f` (float) o `d` (double). Para aplicaciones bidimensionales, establecemos `tz = 0.0`. Y una posición bidimensional se representa como una matriz columna de 4 elementos con la componente `z` igual a 0.0. La matriz de traslación generada por esta función se usa para transformar las posiciones de los objetos definidos después de que se haya invocado a esta función. Por ejemplo, para trasladar posiciones definidas de coordenadas 25 unidades en la dirección `x` y -10 unidades en la dirección `y` utilizamos la instrucción

```
glTranslatef (25.0, -10.0, 0.0);
```

De manera similar, una matriz de rotación de 4 por 4 se genera con:

```
glRotate* (theta, vx, vy, vz);
```

donde el vector `v = (vx, vy, vz)` puede tomar cualquier valor en punto flotante para sus componentes. Este vector define la orientación para la rotación del elemento que pasa a través del origen de coordenadas. Si `v` no se especifica como vector unidad, entonces se normaliza automáticamente antes de que los elementos de la matriz de rotación sean calculados. El código de sufijo puede ser bien `f` o bien `d`, y el parámetro `theta` va a ser asignado al ángulo de rotación en grados, cuya rutina lo convierte en radianes para los cálculos trigonométricos. Esta función genera una matriz de rotación usando los cálculos cuaterniales de la Ecuación 5.108, la cual se aplica a las posiciones definidas después de que sea llamada esta función. Por ejemplo, la instrucción:

```
glRotatef (90.0, 0.0, 0.0, 1.0);
```

define una matriz para rotaciones de 90° sobre el eje `z`.

Obtenemos una matriz de cambio de escala de 4 por 4 con respecto al origen de coordenadas con la siguiente rutina.

```
glScale* (sx, sy, sz);
```

El código sufijo es de nuevo `f` o `d`, y los parámetros de escala pueden tener como valor números reales. Por tanto, esta función generará también reflexiones cuando se asignen valores negativos a los parámetros de escala. Por ejemplo, la siguiente instrucción genera una matriz que aplica un cambio de escala con el factor 2 en la dirección `x`, con el factor 3 en la dirección `y`, y aplica una reflexión con respecto al eje `x`.

```
glScalef (2.0, -3.0, 1.0);
```

Un valor de cero para cualquier parámetro de escala puede causar un error de procesamiento, porque la matriz inversa no puede calcularse. La matriz de cambio de escala-reflexión se aplica a los objetos que se definen con posterioridad.

## Operaciones con matrices en OpenGL

En la Sección 2.9 vimos que la rutina `glMatrixMode` se usaba para establecer el *modo proyección*, el cual designaba la matriz que iba a ser usada para la transformación de proyección. Esta transformación determina cómo una escena va a proyectarse sobre la pantalla. Usamos la misma rutina para fijar la matriz para las transformaciones geométricas. Pero en este caso la matriz es referenciada como matriz *modelview* (vista de modelo), y se emplea para almacenar y combinar las transformaciones geométricas. Además se usa para combinar las transformaciones geométricas con la transformación de un sistema de coordenadas de visualización. Específicamente, el modo *modelview* con la instrucción:

```
glMatrixMode (GL_MODELVIEW);
```

la cual designa la matriz *modelview* de 4 por 4 como la **matriz actual**. Las rutinas de transformación de OpenGL discutidas en la sección anterior se usan para modificar la matriz *modelview*, la cual se aplica para transformar posiciones de coordenadas en una escena. Otros dos modos que podemos establecer con la función `glMatrixMode` son el modo textura (*texture mode*) y el modo color (*color mode*). La matriz de textura se usa para mapear patrones de textura a superficies y la matriz de color se usa para convertir de un modelo de color a otro. Veremos las transformaciones de visualizaciones, proyección, textura y color en los siguientes capítulos. Por el momento, limitaremos esta exposición a los detalles de transformaciones geométricas. El argumento predeterminado para la función `glMatrixMode` es `GL_MODELVIEW`.

Una vez que estamos en el modo *modelview* (o cualquier otro modo) una llamada a una rutina de transformación genera una matriz que se multiplica por la matriz actual para ese modo. Además, podemos asignar valores a los elementos de una matriz actual, y hay dos funciones en las bibliotecas de OpenGL para este propósito. Con la siguiente función, asignamos la matriz identidad a la matriz actual.

```
glLoadIdentity ( );
```

Alternativamente, podemos asignar otros valores a los elementos de una matriz actual usando:

```
glLoadMatrix* (elements16);
```

Un subíndice simple, un array de 16 elementos de valores en punto flotante, se especifica con el parámetro `elements16`, y un código de sufijo, `f` o `d`, se usa para designar tipos de datos. Los elementos en este array deben especificarse por columnas. Es decir, primero se enumeran los cuatro elementos de la primera columna, y luego los cuatro elementos de la segunda columna, la tercera columna y, finalmente, la cuarta columna. Para ilustrar este orden, inicializamos la matriz *modelview* con el siguiente código.

```
glMatrixMode (GL_MODELVIEW);

GLfloat elems [16];
GLint k;

for (k = 0; k < 16; k++)
    elems [k] = float (k);
glLoadMatrixf (elems);
```

que da lugar a la matriz:

$$\mathbf{M} = \begin{bmatrix} 0.0 & 4.0 & 8.0 & 12.0 \\ 1.0 & 5.0 & 9.0 & 13.0 \\ 2.0 & 6.0 & 10.0 & 14.0 \\ 3.0 & 7.0 & 11.0 & 15.0 \end{bmatrix}$$

También podemos concatenar una matriz específica con la matriz actual:

```
glMultMatrix* (otherElements16);
```

De nuevo, el código de sufijo es o bien `f` o `d`, y el parámetro `otherElements16` es un array de subíndices simples de 16 elementos que enumera los elementos de alguna otra matriz por columnas. La matriz actual se *postmultiplica* con la matriz especificada en `glMultMatrix`, y este producto sustituye a la matriz actual. Así, asumiendo que la matriz actual es la matriz *modelview*, la cual designamos como **M**, la matriz *modelview* actualizada se calcula como:

$$\mathbf{M} = \mathbf{M} \cdot \mathbf{M}'$$

donde **M'** representa la matriz cuyos elementos se especifican con el parámetro `otherElements16` en la instrucción `glMultMatrix` anterior.

La función `glMultMatrix` también puede usarse para establecer cualquier secuencia de transformaciones con matrices definidas individualmente. Por ejemplo,

```

glMatrixMode (GL_MODELVIEW);

glLoadIdentity (); // Establece la matriz actual como la matriz identidad.
glMultMatrixf (elemsM2); // Postmultiplica la matriz identidad con la matriz M2.
glMultMatrixf (elemsM1); // Postmultiplica M2 con la matriz M1.

```

genera la siguiente matriz *modelview* actual.

$$\mathbf{M} = \mathbf{M}_2 \cdot \mathbf{M}_1$$

La primera transformación que hay que aplicar en esta secuencia es la última especificada en el código. Así, si establecemos una secuencia de transformaciones en un programa OpenGL, podemos pensar en las transformaciones individuales como si estuvieran cargadas en una pila, de tal forma que la última operación especificada es la primera en aplicarse. Esto no es lo que sucede en realidad, pero la analogía con la pila puede ayudar a recordarlo, en un programa OpenGL, una secuencia de transformaciones se aplica en el orden opuesto del que se especificó.

También es importante tener en mente que OpenGL almacena matrices por columnas. Y una referencia a un elemento de una matriz, tal como  $m_{jk}$ , en OpenGL, es una referencia al elemento de la columna  $j$  y la fila  $k$ . Esto es a la inversa que el convenio estándar matemático, donde el número de fila se referencia primero. Pero podemos evitar errores comunes en las referencias de filas y columnas especificando siempre matrices en OpenGL como arrays de 16 elementos, con un solo subíndice y recordando que los elementos deben enumerarse por columnas.

## Pilas de matrices en OpenGL

Para cada uno de los cuatro modos (*modelview*, proyección, textura y color) que podemos seleccionar con la función `glMatrixMode`, OpenGL mantiene una pila de matrices. Inicialmente, cada pila contiene sólo la matriz identidad. En cualquier momento durante el procesamiento de una escena, la matriz en la cima de cada pila es la «matriz actual» para dicho modo. Después de que hayamos especificado las transformaciones geométricas y de visualización, la cima de la **pila de matrices de *modelview*** es una matriz compuesta de 4 por 4 que combina las transformaciones de visualización y las diversas transformaciones geométricas que queramos aplicar a la escena. En algunos casos, podemos querer crear múltiples vistas y secuencias de transformación, y luego guardar la matriz compuesta para cada una. Por tanto, OpenGL soporta una pila de *modelview* de profundidad al menos 32, y algunas implementaciones deben permitir más de 32 matrices que guardar en la pila de *modelview*. Podemos determinar el número de posiciones disponibles en la pila de *modelview* para una implementación particular de OpenGL con

```
glGetIntegerv (GL_MAX_MODELVIEW_STACK_DEPTH, stackSize);
```

que devuelve un único valor entero al array `stackSize`. Los otros tres modo de matrices tienen un mínimo de profundidad de pila de 2, y podemos determinar la profundidad máxima disponible para cada una en una implementación particular usando una de las siguientes constantes simbólicas de OpenGL: `GL_MAX_PROJECTION_STACK_DEPTH`, `GL_MAX_TEXTURE_STACK_DEPTH`, o `GL_MAX_COLOR_STACK_DEPTH`.

También podemos averiguar cuántas matrices hay actualmente en la pila con:

```
glGetIntegerv (GL_MODELVIEW_STACK_DEPTH, numMats);
```

Inicialmente, la pila de *modelview* contiene sólo la matriz identidad, de tal forma que esta función devuelve el valor 1 si realizamos la consulta antes de que haya tenido lugar ningún procesamiento sobre la pila. Hay disponibles constantes simbólicas similares para determinar el número de matrices que hay actualmente en las otras tres pilas.

Tenemos dos funciones disponibles en OpenGL para procesar las matrices que hay en una pila. Estas funciones de procesamiento de la pila son más eficientes que manipular la pila de matrices de forma individual, particularmente cuando las funciones de la pila se implementan por hardware. Por ejemplo, una implementa-

ción hardware puede copiar múltiples elementos de matrices simultáneamente. Y podemos mantener una matriz identidad en la pila, de tal forma que inicializaciones de la matriz actual pueden llevarse a cabo más rápidamente que usando repetidas llamadas a `glLoadIdentity`.

Con la siguiente función, copiamos la matriz actual en la cima de la pila activa y almacenamos esa copia en la segunda posición de la pila.

```
glPushMatrix ( );
```

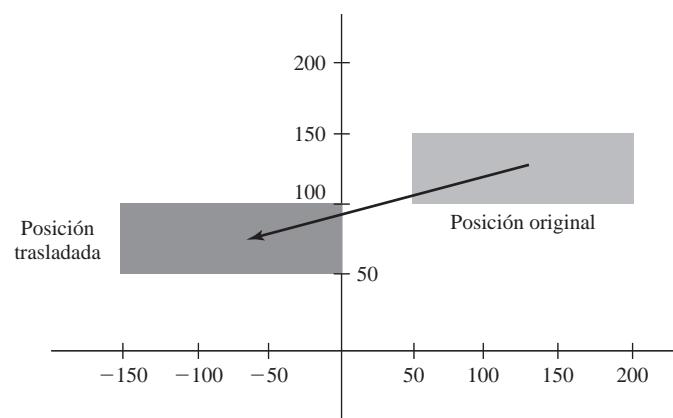
Esto nos da matrices duplicadas en las dos posiciones de la cima de la pila. La otra función de la pila es:

```
glPopMatrix ( );
```

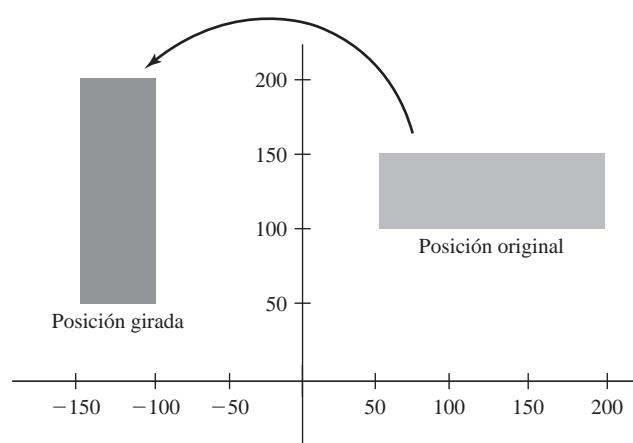
la cual destruye la matriz de la cima de la pila, y la segunda matriz en la pila se convierte en la matriz actual. Para extraer el contenido de la cima de la pila, debe haber al menos dos matrices en la pila. En cualquier otro caso se generará un error.

## Ejemplos de programas de transformaciones geométricas OpenGL

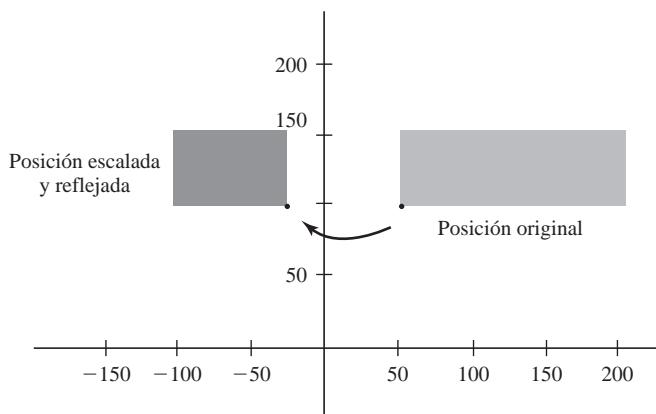
En el siguiente fragmento de código, aplicamos cada una de las transformaciones geométricas básicas, una cada vez, a un rectángulo. Inicialmente, la matriz de *modelview* es la matriz identidad y muestra un rectángulo azul (gris claro en las figuras). A continuación, se establece el color actual como rojo, se especifican los parámetros de traslación bidimensionales y se muestra el rectángulo rojo (gris más oscuro en las figuras) trasladado (Figura 5.55).



**FIGURA 5.55.** Traslación de un rectángulo usando la función OpenGL `glTranslatef (-200.0, -50.0, 0.0)`.



**FIGURA 5.56.** Rotación de un rectángulo sobre el eje x usando la función de OpenGL `glRotatef (90.0, 0.0, 1.0)`.



**FIGURA 5.57.** Cambio de escala y reflexión de un rectángulo usando la función OpenGL `glScalef (-0.5, 1.0, 1.0)`

Mientras no queramos combinar transformaciones, lo siguiente será establecer la matriz actual como la matriz identidad. Luego se construye una matriz de rotación y se concatena con la matriz actual (la matriz identidad). Cuando el rectángulo original es referenciado de nuevo, se gira sobre el eje  $z$  y se muestra en color rojo (gris oscuro en la figura) (Figura 5.56). Repetimos este proceso una vez más para generar el rectángulo cambiado de escala y reflejado que se muestra en la Figura 5.57.

```

glMatrixMode (GL_MODELVIEW);

	glColor3f (0.0, 0.0, 1.0);
	glRecti (50, 100, 200, 150); // Muestra un rectángulo azul.

	glColor3f (1.0, 0.0, 0.0);
	glTranslatef (-200.0, -50.0, 0.0); // Establece los parámetros de traslación.

	glRecti (50, 100, 200, 150); // Muestra un rectángulo rojo trasladado.
	glLoadIdentity (); // Carga la matriz identidad como matriz actual.

	glRotatef (90.0, 0.0, 0.0, 1.0); // Establece una rotación de 90 grados alrededor
	// del eje z .
	glRecti (50, 100, 200, 150); // Muestra un rectángulo rojo girado.

	glLoadIdentity (); // Carga la matriz identidad como matriz actual.
	glScalef (-0.5, 1.0, 1.0); // Establece los parámetros de reflexión y escala.
	glRecti (50, 100, 200, 150); // Muestra un rectángulo rojo transformado.

```

Normalmente, es más eficiente usar las funciones de procesado de pilas que usar las funciones de manipulación de matrices. Esto es particularmente cierto cuando queremos hacer varios cambios en las vistas o en las transformaciones geométricas. En el siguiente código, repetimos las transformaciones del rectángulo del ejemplo anterior usando el procesado de la pila en lugar de la función `glLoadIdentity`.

```

glMatrixMode (GL_MODELVIEW);

	glColor3f (0.0, 0.0, 1.0); // Establece el color actual en azul.

```

```

glRecti (50, 100, 200, 150); // Presenta un rectángulo azul.

glPushMatrix ( ); // Hace una copia de la matriz identidad (superior).
glColor3f (1.0, 0.0, 0.0); // Establece el color actual en rojo.

glTranslatef (-200.0, -50.0, 0.0); // Establece los parámetros de traslación.
glRecti (50, 100, 200, 150); // Muestra un rectángulo rojo trasladado.

glPopMatrix ( ); // Recorre la matriz de traslación.
glPushMatrix ( ); // Hace una copia de la matriz identidad (superior).

glRotatef (90.0, 0.0, 0.0, 1.0); // Define rotación de 90 grados sobre el eje z.
glRecti (50, 100, 200, 150); // Muestra un rectángulo rojo girado.

glPopMatrix ( ); // Recorre la matriz de rotación.
glScalef (-0.5, 1.0, 1.0); // Establece los parámetros de reflexión y escala.
glRecti (50, 100, 200, 150); // Muestra un rectángulo rojo transformado.

```

Para nuestro programa de ejemplo final de transformación geométrica, damos una versión de OpenGL para código de transformaciones compuestas tridimensionales en la Sección 5.13. Como OpenGL postmultiplica matrices de transformación según se las va llamando, debemos ahora invocar las transformaciones en el orden opuesto al que vayan a ser aplicadas. Así, cada llamada a una transformación subsiguiente concatena la designada matriz de transformación a la derecha de la matriz compuesta. Como no hemos explorado aún las rutinas de visualización tridimensional de OpenGL (Capítulo 7) este programa podría completarse usando operaciones de visualización bidimensional de OpenGL y aplicando las transformaciones geométricas a objetos en el plano *xy*.

---

```

class wcPt3D {
public:
    GLfloat x, y, z;
};

/* Procedimiento para generar una matriz para girar alrededor de
 * un eje definido por los puntos p1 y p2.*/
void rotate3D (wcPt3D p1, wcPt3D p2, GLfloat thetaDegrees)
{
    /* Establece las componentes del vector de rotación según el eje. */
    float vx = (p2.x - p1.x);
    float vy = (p2.y - p1.y);
    float vz = (p2.z - p1.z);

    /* Especifica la secuencia traslación-rotación-traslación en orden inverso: */
    glTranslatef (p1.x, p1.y, p1.z); // Mueve p1 a su posición original.
    /* Rotación alrededor del eje que pasa por el origen: */
    glRotatef (thetaDegrees, vx, vy, vz);
    glTranslatef (-p1.x, -p1.y, -p1.z); // Traslada p1 al origen.
}

```

```

/* Procedimiento para generar una matriz para la transformación de cambio
 * de escala con respecto a un punto fijo arbitrario.
 */
void scale3D (GLfloat sx, GLfloat sy, GLfloat sz, wcPt3D fixedPt)
{
    /* Especifica la secuencia traslación-escalado-traslación en orden inverso: */
    /* (3) Traslada el punto fijo a su posición original: */
    glTranslatef (fixedPt.x, fixedPt.y, fixedPt.z);
    glScalef (sx, sy, sz); // (2) Escalado con respecto al origen.

    /* (1) Traslada un punto fijo al origen de coordenadas: */
    glTranslatef (-fixedPt.x, -fixedPt.y, -fixedPt.z);
}

void displayFcn (void)
{
    /* Introducir la descripción del objeto. */
    /* Definir las rutinas de visualización y transformación 3D. */
    /* Mostrar el objeto. */
    glMatrixMode (GL_MODELVIEW);

    /* Introducir los parámetros de traslación tx, ty, tz. */
    /* Introducir los puntos que definen, p1 y p2, el eje de rotación. */
    /* Introducir en grados el ángulo de rotación. */
    /* Introducir los parámetros de cambio de escala: sx, sy, sz y fixedPt. */
    /* Invocar las transformaciones geométricas en orden inverso: */
    glTranslatef (tx, ty, tz); // Transformación final: traslación.
    scale3D (sx, sy, sz, fixedPt); // Segunda transformación: cambio de escala.
    rotate3D (p1, p2, thetaDegrees); // Primera transformación: rotación.

    /* Llamar a la rutinas que muestran los objetos transformados. */
}

```

## 5.18 RESUMEN

---

Las transformaciones geométricas básicas son la traslación, la rotación y el cambio de escala. La traslación mueve un objeto con una trayectoria en línea recta de una posición a otra. La rotación mueve un objeto de una posición a otra a lo largo de una trayectoria circular sobre un eje de rotación específico. Para aplicaciones bidimensionales, la trayectoria de rotación se encuentra en el plano  $xy$  sobre un eje que es paralelo al eje  $z$ . Las transformaciones de cambio de escala cambian las dimensiones de un objeto con respecto a una posición fija.

Podemos expresar transformaciones bidimensionales como operadores de matrices de 3 por 3 y transformaciones tridimensionales como operadores de matrices de 4 por 4, de tal forma que esas secuencias de transformaciones pueden concatenarse dentro de una matriz compuesta. O, en general, podemos representar tanto transformaciones bidimensionales como tridimensionales con matrices de 4 por 4. Representar operaciones de transformaciones geométricas con matrices es una formulación eficiente, en tanto en cuanto nos permite reducir los cálculos aplicando una matriz compuesta a una descripción de un objeto para obtener su posición transformada. Para hacer esto, expresamos posiciones de coordenadas como matrices columna. Elegimos la repre-

sentación de matriz columna para puntos de coordenadas porque ese es el convenio matemático estándar, y muchos paquetes gráficos siguen dicha convención. Nos referimos a una matriz de tres o cuatro elementos (vector) como una representación de coordenadas homogéneas. Para transformaciones geométricas, al coeficiente homogéneo se le asigna el valor 1.

Las transformaciones compuestas se forman como multiplicación de matrices de traslación, rotación, cambio de escala y otras transformaciones. Podemos usar combinaciones de traslación y rotación para aplicaciones de animación, y podemos usar combinaciones de rotación y cambio de escala para cambiar el tamaño de los objetos en cualquier dirección especificada. En general, la multiplicación de matrices no es commutativa. Obtenemos diferentes resultados, por ejemplo, si cambiamos el orden de la secuencia traslación-rotación. Una secuencia de transformación que implica sólo transformaciones y rotaciones es una transformación de sólido-rígido, mientras que los ángulos y las distancias se mantienen invariables. Además, la submatriz superior izquierda de una transformación de sólido-rígido es una matriz ortogonal. Así, la rotación de matrices puede formarse estableciendo la submatriz superior izquierda de 3 por 3 igual a los elementos de los dos vectores unidad ortogonales. Cuando el ángulo es pequeño, podemos reducir el cálculo de las rotaciones usando aproximaciones de primer orden para las funciones de seno y coseno. A lo largo de muchos pasos rotacionales, sin embargo, el error de aproximación puede acumularse y pasar a ser un valor significativo.

Otras transformaciones geométricas incluyen operaciones de reflexión e inclinación. Las reflexiones son transformaciones que giran un objeto 180° sobre un eje de reflexión. Esto produce una imagen de espejo del objeto con respecto a dicho eje. Cuando el eje de reflexión está en el plano *xy*, la reflexión se obtiene como una rotación en un plano que es perpendicular al plano *xy*. Las transformaciones de inclinación distorsionan la forma de un objeto desplazando uno o más valores de coordenadas en una cantidad proporcional a la distancia respecto de la línea de inclinación de referencia.

Las transformaciones entre sistemas de coordenadas cartesianos se llevan a cabo con una secuencia de transformaciones traslación-rotación que hacen que los dos sistemas coincidan. Específicamente, se menciona el origen de coordenadas y vectores de eje para un marco de referencia respecto al marco de coordenadas de referencia original. En un sistema bidimensional, un vector define completamente las direcciones del eje de coordenadas. Pero, en un sistema tridimensional, hay que especificar dos de las tres direcciones de los ejes. La transferencia de las descripciones de objetos desde el sistema de coordenadas original al segundo sistema se calcula como la matriz producto de una traslación que mueve el nuevo origen al antiguo origen de coordenadas y una rotación para alinear los dos juegos de ejes. La rotación necesaria para alinear los dos sistemas puede obtenerse del juego ortonormal de vectores de eje para el nuevo sistema.

Las transformaciones geométricas son transformaciones afines. Esto es, pueden expresarse como una función lineal de posiciones de coordenadas. Traslación, rotación, cambio de escala, reflexión e inclinación son transformaciones afines. Transforman líneas paralelas en líneas paralelas y posiciones de coordenadas finitas en posiciones finitas. Las transformaciones geométricas que no implican cambio de escala o inclinación también mantienen los ángulos y las longitudes.

Podemos usar operaciones de rasterización para desarrollar algunas transformaciones geométricas sencillas en arrays de píxeles. Para aplicaciones bidimensionales, podemos usar operaciones de rasterización para llevar a cabo traslaciones rápidas, reflexiones y rotaciones en múltiplos de 90°. Con un poco más de procesamiento, podemos realizar rotaciones y cambio de escala rasterizadas de tipo general.

La biblioteca básica de OpenGL contiene tres funciones para aplicar transformaciones individuales de traslación, rotación y cambio de escala a posiciones de coordenadas. Cada función genera una matriz que se pre-multiplica con la matriz *modelview*. Así, una secuencia de funciones de transformación geométrica deben especificarse en orden inverso: la última transformación invocada es la primera en aplicarse a las posiciones de coordenadas. Las matrices de transformación se aplican a los objetos definidos con posterioridad. Además de acumular secuencias de transformación en la matriz *modelview*, podemos establecer esta matriz como la matriz identidad o alguna otra. Podemos también formar productos con la matriz *modelview* y cualquier matriz especificada. Todas las matrices se almacenan en pilas, y OpenGL mantiene cuatro pilas para los distintos tipos de transformación que se usan en las aplicaciones gráficas. Podemos usar una función de consul-

ta en OpenGL para determinar el tamaño actual de la pila o la profundidad máxima permitida en la pila para un sistema. Dos rutinas de procesado de pilas están disponibles: una para copiar la cima de la matriz de la pila a la segunda posición, y una para eliminar el contenido de la cima de la pila. Varias operaciones están disponibles en OpenGL para realizar transformaciones de rasterización. Un bloque de píxeles puede trasladarse, rotarse, cambiarse de escala o reflejarse con estas operaciones de rasterización de OpenGL.

La Tabla 5.1 resume las funciones geométricas de OpenGL y las rutinas de matrices vistas en este capítulo. Adicionalmente, la tabla enumera algunas funciones relacionadas.

**TABLA 5.1. RESUMEN DE FUNCIONES PARA TRANSFORMACIONES GEOMÉTRICAS EN OpenGL.**

Función	Descripción
<code>glTranslate*</code>	Especifica los parámetros de traslación.
<code>glRotate*</code>	Especifica los parámetros para la rotación sobre cualquier eje a través del origen.
<code>glScale*</code>	Especifica los parámetros de escala respecto al origen de coordenadas.
<code>glMatrixMode</code>	Especifica la matriz actual para transformaciones de visualización geométrica, transformaciones de proyección, transformaciones de textura o transformaciones de color.
<code>glLoadIdentity</code>	Establece la matriz identidad actual.
<code>glLoadMatrix*(elems);</code>	Establece los elementos de la matriz actual.
<code>glMultMatrix*(elems);</code>	Postmultiplica la matriz actual con la matriz especificada.
<code>glGetIntegerv</code>	Obtiene la profundidad máxima de la pila o el número de matrices en la pila para el modo de matriz seleccionado.
<code>glPushMatrix</code>	Copia la cima de la pila y almacena la copia en la segunda posición de la pila.
<code>glPopMatrix</code>	Borra la cima de la pila y mueve la segunda matriz a la cima de la pila.
<code>glPixelZoom</code>	Especifica los parámetros de escala bidimensionales para operaciones de rasterización.

## REFERENCIAS

Para técnicas adicionales que implican matrices y transformaciones geométricas, véase Glassner (1990), Arvo (1991), Kirk (1992), Heckbert (1994), y Paeth (1995). Exposiciones acerca de coordenadas homogéneas en gráficos por computadora se pueden encontrar en Blinn y Newell (1978) y en Blinn (1993, 1996, y 1998).

Adicionalmente, ejemplos de programas usando funciones de transformaciones geométricas en OpenGL se dan en Woo, Neider, Davis, y Shreiner (1999). Ejemplos de programas para las funciones de transformaciones geométricas en OpenGL están también disponibles en el tutorial del sitio web de Nate Robins: <http://www.cs.utah.edu/narobins/opengl.html>. Y un listado completo de funciones de transformaciones geométricas en OpenGL se ofrecen en Shreiner (2000).

## EJERCICIOS

- 5.1 Escribir un programa de animación que implemente el ejemplo de procedimiento de rotación bidimensional de la Sección 5.1. Un polígono de entrada va a ser rotado repetidamente en pequeños pasos sobre un punto de pivote en el plano *xy*. Los pequeños ángulos van a ser usados para cada paso sucesivo en la rotación, y las aproximacio-

nes para las funciones de seno y coseno van a ser usadas para agilizar los cálculos. Para evitar la acumulación excesiva de errores de redondeo, re establecer los valores de coordenadas originales para el objeto al principio de cada nueva revolución.

- 5.2 Demostrar que la composición de dos rotaciones es aditiva mediante la concatenación de representaciones de matrices para  $\mathbf{R}(\theta_1)$  y  $\mathbf{R}(\theta_2)$  para obtener

$$\mathbf{R}(\theta_1) \cdot \mathbf{R}(\theta_2) = \mathbf{R}(\theta_1 + \theta_2)$$

- 5.3 Modificar la matriz de transformación bidimensional (5.39) para hacer un cambio de escala en una dirección arbitraria, con el fin de incluir coordenadas para cualquier punto fijo de cambio de escala especificado  $(x_f, y_f)$ .
- 5.4 Probar que la multiplicación de transformaciones de matrices para cada una de las siguientes secuencias es conmutativa:
- (a) Dos rotaciones sucesivas.
  - (b) Dos traslaciones sucesivas.
  - (c) Dos cambios de escala sucesivos.
- 5.5 Probar que un cambio de escala uniforme y una rotación forman un par de operadores conmutativo pero que, en general, el cambio de escala y la rotación son operaciones no conmutativas.
- 5.6 Multiplicar las matrices de cambio de escala, rotación y traslación en la Ecuación 5.42 para verificar los elementos de la matriz de transformación compuesta.
- 5.7 Modificar el programa de ejemplo en la Sección 5.4 de tal forma que los parámetros de transformación puedan especificarse como entradas de usuario.
- 5.8 Modificar el programa del ejercicio anterior de tal forma que la secuencia de transformaciones pueda ser aplicada a cualquier polígono, con los vértices especificados como entradas de usuario.
- 5.9 Modificar el programa de ejemplo de la Sección 5.4 de tal forma que el orden de la secuencia de transformaciones geométricas pueda especificarse como entrada de usuario.
- 5.10 Demostrar que la matriz de transformación (5.55) para una reflexión sobre la línea  $y = x$ , es equivalente a una reflexión relativa al eje  $x$  seguida de una rotación de  $90^\circ$  en el sentido contrario a las agujas del reloj.
- 5.11 Demostrar que la matriz de transformación (5.56) para una reflexión sobre la línea  $y = -x$ , es equivalente a una reflexión relativa al eje  $y$  seguida de una rotación de  $90^\circ$  en el sentido contrario a las agujas del reloj.
- 5.12 Demostrar que reflexiones sucesivas sobre el eje  $x$  o el eje  $y$  son equivalentes a una rotación en el plano  $xy$  alrededor del origen de coordenadas.
- 5.13 Determinar la forma de una matriz de transformación bidimensional para una operación de reflexión sobre cualquier línea:  $y = mx + b$ .
- 5.14 Demostrar que dos reflexiones sucesivas sobre cualquier línea en el plano  $xy$  que intersecta con el origen de coordenadas son equivalentes a una rotación en el plano  $xy$  sobre el origen.
- 5.15 Determinar una secuencia de transformaciones básicas que sea equivalente a la matriz de inclinación en la dirección- $x$  (5.57).
- 5.16 Determinar una secuencia de transformaciones básicas que sea equivalente a la matriz de inclinación en la dirección- $y$  (5.61).
- 5.17 Establecer un procedimiento de inclinación para mostrar caracteres en cursiva bidimensionales dado un vector como fuente de definición. Es decir, todos los tamaños de los caracteres en esta fuente se definen con segmentos en línea recta, y los caracteres en cursiva se forman con transformaciones de inclinación. Determinar un valor apropiado para el parámetro de inclinación mediante la comparación de cursivas y texto plano en algunas fuentes disponibles. Definir un vector fuente como entrada a la rutina.
- 5.18 Deducir las siguientes ecuaciones para transformar un punto de coordenadas  $\mathbf{P} = (x, y)$  en un sistema cartesiano bidimensional en los valores de coordenadas  $(x', y')$  en otro sistema cartesiano que se rota en sentido contrario a las agujas del reloj con el ángulo  $\theta$  respecto al primer sistema. Las ecuaciones de transformación pueden obtenerse proyectando el punto  $\mathbf{P}$  sobre cada uno de los cuatro ejes y analizando los triángulos rectángulos resultantes.

$$x' = x \cos \theta + y \sin \theta \quad y' = -x \sin \theta + y \cos \theta$$

- 5.19 Escribir un procedimiento para calcular los elementos de una matriz para la transformación de descripciones de objetos de un sistema de coordenadas cartesiano bidimensional a otro. El segundo sistema de coordenadas va a definirse con un punto origen  $\mathbf{P}_0$  y un vector  $\mathbf{V}$  que da la dirección del eje positivo  $y$  para este sistema.
- 5.20 Establecer procedimientos para implementar la transferencia de un bloque de un área rectangular de un búfer de imagen, usando una función para leer el área dentro de un array y otra función para copiar el array dentro del área de transferencia designado.
- 5.21 Determinar los resultados de desarrollar dos transferencias de bloque sucesivas dentro de un mismo área de un búfer de imagen usando varias operaciones booleanas.
- 5.22 ¿Cuáles son los resultados de desarrollar dos transferencias de bloque sucesivas dentro del mismo área de un búfer de imagen usando operaciones aritméticas binarias?
- 5.23 Implementar una rutina para llevar a cabo transferencias de bloque en un búfer de imagen usando cualquier operación booleana especificada o una operación de sustitución (copia).
- 5.24 Escribir una rutina para implementar rotaciones en incrementos de  $90^\circ$  en transferencias de bloque de un búfer de imagen.
- 5.25 Escriba una rutina para implementar rotaciones para cualquier ángulo especificado en una transferencia de bloque de un búfer de imagen.
- 5.26 Escriba una rutina para implementar el cambio de escala como una transformación de rasterización de un bloque de píxeles.
- 5.27 Demostrar que la matriz de rotación 5.102 es igual a la matriz compuesta  $\mathbf{R}_y(\beta) \cdot \mathbf{R}_x(\alpha)$ .
- 5.28 Evaluando los términos de la Ecuación 5.106, deducir los elementos para la matriz de rotación general dada en la Ecuación 5.107.
- 5.29 Probar que una matriz de rotación cuaternal 5.107 reduce la matriz de representación en la Ecuación 5.74 cuando el eje de rotación es el eje de coordenadas  $z$ .
- 5.30 Probar que la Ecuación 5.109 es equivalente a la transformación de rotación general dada en la Ecuación 5.97.
- 5.31 Usando identidades trigonométricas, deducir los elementos de la matriz de rotación cuaternal 5.108 a partir de la expresión 5.107.
- 5.32 Desarrollar un procedimiento para animar un objeto tridimensional rotándolo incrementalmente alrededor de un eje especificado. Usar aproximaciones apropiadas para las ecuaciones trigonométricas con el fin de acelerar los cálculos, y re establecer el objeto a su posición original después de cada revolución completa sobre el eje.
- 5.33 Deducir la matriz de transformación tridimensional para el cambio de escala de un objeto mediante el factor de escala  $s$  en la dirección definida por los cosenos de dirección  $\alpha$ ,  $\beta$  y  $\gamma$ .
- 5.34 Desarrollar una rutina para reflejar un objeto tridimensional sobre un plano arbitrario seleccionado.
- 5.35 Escribir un procedimiento para inclinar un objeto tridimensional con respecto a cualquiera de los tres ejes de coordenadas, usando valores de entrada para los parámetros de inclinación.
- 5.36 Desarrollar un procedimiento para convertir la definición de un objeto en un sistema de referencia de coordenadas tridimensional en cualquier otro sistema de coordenadas definido con respecto al primer sistema.
- 5.37 Implementar el programa de ejemplo de la Sección 5.17 de tal forma que las rutinas de transformaciones geométricas tridimensionales OpenGL se apliquen al triángulo bidimensional mostrado en la Figura 5.15(a), para producir la transformación mostrada en la parte (b) de dicha figura.
- 5.38 Modificar el programa del ejercicio anterior para que la secuencia de transformación se pueda aplicar a cualquier polígono bidimensional, cuyos vértices sean especificados como entradas de usuario.
- 5.39 Modificar el ejemplo del programa anterior para que el orden de las secuencias de transformaciones geométricas pueda especificarse como entradas de usuario.
- 5.40 Modificar el ejemplo del programa anterior para que los parámetros de la transformación geométrica se especifiquen como entradas de usuario.

# Visualización bidimensional



Una escena que representa un jardín con colibríes pintada por el artista John Derry de Time Arts, Inc., utilizando una tableta gráfica con un lapicero sensible a la presión e inalámbrico. (*Cortesía de Wacom Technology Corporation.*)

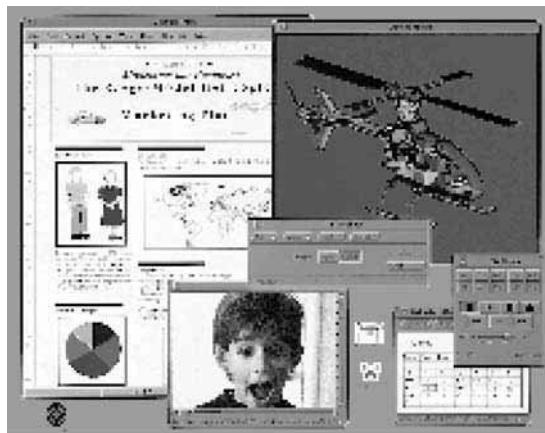
<b>6.1</b>	Pipeline de visualización bidimensional	<b>6.6</b>	Recorte de puntos bidimensionales
<b>6.2</b>	La ventana de recorte	<b>6.7</b>	Recorte de líneas bidimensionales
<b>6.3</b>	Normalización y transformaciones de visor	<b>6.8</b>	Recorte de áreas de relleno de polígonos
<b>6.4</b>	Funciones de visualización bidimensional de OpenGL	<b>6.9</b>	Recorte de curvas
<b>6.5</b>	Algoritmos de recorte	<b>6.10</b>	Recorte de texto
		<b>6.11</b>	Resumen

En el Capítulo 2, presentamos brevemente los conceptos y las funciones de visualización bidimensional. Ahora examinaremos con más detalle los procedimientos para mostrar vistas de una imagen bidimensional en un dispositivo de salida. Habitualmente, un paquete gráfico permite al usuario especificar qué parte de una imagen definida se debe visualizar y dónde esta parte se debe colocar en el dispositivo de visualización. Cualquier sistema de coordenadas cartesianas que sea conveniente, referido al sistema de referencia de coordenadas del mundo, se puede usar para definir la imagen. En el caso de imágenes bidimensionales, una vista se selecciona especificando una región de plano *xy* que contiene la imagen total o cualquier parte de ella. Un usuario puede seleccionar una única zona para visualización, o varias zonas para visualización simultánea o para una secuencia animada panorámica a través de una escena. Las partes dentro de las zonas seleccionadas se mapean entonces sobre zonas especificadas de las coordenadas del dispositivo. Cuando se seleccionan múltiples zonas de vista, estas zonas se pueden colocar en ubicaciones de visualización independientes, o algunas zonas se podrían insertar en otras zonas de visualización más grandes. Las transformaciones de visualización bidimensional desde las coordenadas universales a las coordenadas del dispositivo implican operaciones de translación, rotación y cambio de escala, así como procedimientos de borrado de aquellas partes de la imagen que se encuentran fuera de los límites de una zona seleccionada de la escena.

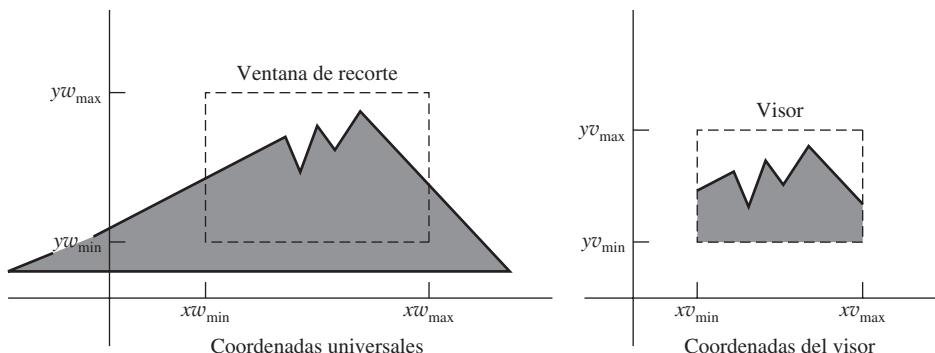
## 6.1 PIPELINE DE VISUALIZACIÓN BIDIMENSIONAL

---

Un parte de una escena bidimensional que se ha seleccionado para su visualización se denomina **ventana de recorte** (*clipping*), porque todas las partes de la escena situadas fuera de la parte seleccionada se «recortan». La única parte de la escena que se muestra en la pantalla es la que se encuentra dentro de la ventana de recorte. A veces la ventana de recorte se denomina *ventana universal* o *ventana de visualización*. Y, alguna vez, los sistemas gráficos se refieren a la ventana de recorte simplemente como «la ventana», pero se utilizan tantas ventanas en los computadores que necesitamos distinguirlas. Por ejemplo, un sistema de gestión de ventanas puede crear y manipular varias zonas en una pantalla de video, cada una de las cuales se denomina «ventana», para la visualización de gráficos y de texto (Figura 6.1). Por lo que, siempre utilizaremos el término *ventana de recorte* para referirnos a una parte seleccionada de una escena que se convierte, en definitiva en patrones de píxeles dentro de una ventana de visualización en un monitor de video. Los paquetes gráficos también nos permiten controlar el emplazamiento dentro de la ventana de visualización utilizando otra «ventana» llamada **visor** (*viewport*). Los objetos que se encuentran dentro de la ventana de recorte se mapean al visor, y es el visor el que se posiciona entonces dentro de la ventana de visualización. La ventana de recorte selecciona *qué* queremos ver; el visor indica *dónde* se debe ver en el dispositivo de salida.



**FIGURA 6.1.** Una pantalla de vídeo mostrando múltiples y simultáneas ventanas de visualización. (Cortesía de Sun Microsystems.)

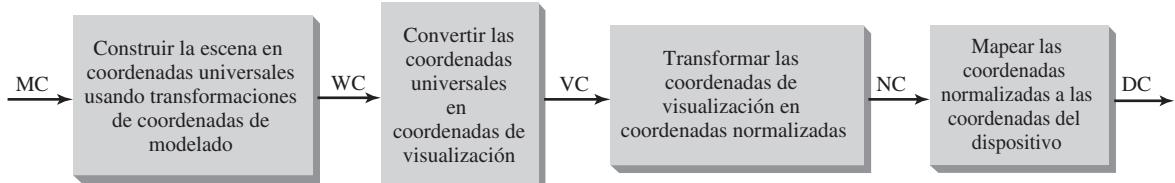


**FIGURA 6.2.** Una ventana de recorte y una vista asociada, especificada como un rectángulo alineado con los ejes de coordenadas.

Cambiando la posición de un visor, podemos ver objetos en posiciones diferentes en la zona de visualización de un dispositivo de salida. Se pueden utilizar múltiples visores para visualizar partes distintas de una escena en posiciones distintas de la pantalla. También, variando el tamaño de los visores, podemos cambiar el tamaño y las proporciones de los objetos visualizados. Conseguimos efectos de zoom mapeando sucesivamente ventanas de recorte de diferente tamaño sobre un visor de tamaño fijo. A medida que las ventanas de recorte se hacen más pequeñas, ampliamos alguna parte de la escena para ver detalles que no se muestran en ventanas de recorte mayores. De forma similar, se obtiene una mayor visión de conjunto reduciendo a partir de una parte de una escena con ventanas de recorte sucesivamente mayores. Los efectos panorámicos se logran moviendo una ventana de recorte de tamaño fijo a través de varios objetos de una escena.

Habitualmente, las ventanas de recorte y los visores son rectángulos en posiciones estándar, con las aristas del rectángulo paralelas a los ejes de coordenadas. En algunas aplicaciones se utilizan otras geometrías de ventana o de visor, tales como polígonos generales y círculos, pero el procesamiento de estas formas requiere un mayor tiempo. En primer lugar consideraremos sólo los visores y las ventanas de recorte rectangulares, como se muestra en la Figura 6.2.

El mapeo de la descripción de una escena bidimensional en coordenadas universales a coordenadas de dispositivo se denomina **transformación de visualización bidimensional**. A veces esta transformación se denomina simplemente *transformación de ventana a visor* o *transformación de ventana*. Pero, por lo general, la visualización implica más que la transformación desde las coordenadas de la ventana de recorte a las coordenadas del visor. Estableciendo una analogía con la visualización tridimensional, podemos describir los pasos de visualización bidimensional como se indica en la Figura 6.3. Una vez que se ha construido la escena en



**FIGURA 6.3.** Pipeline de transformación de visualización bidimensional.

coordenadas universales, podríamos establecer un **sistema independiente bidimensional de referencia de coordenadas de visualización** para especificar la ventana de recorte. Pero la ventana de recorte se define a menudo sólo en coordenadas universales, para que las coordenadas de visualización en aplicaciones bidimensionales sean las mismas en coordenadas universales. (En el caso de una escena tridimensional, sin embargo, necesitamos un sistema de visualización independiente para especificar los parámetros de posición, dirección y orientación de la visualización).

Para hacer que el proceso de visualización sea independiente de los requisitos de cualquier dispositivo de salida, los sistemas gráficos convierten las descripciones de los objetos a coordenadas normalizadas y aplican las subrutinas de recorte. Algunos sistemas utilizan coordenadas normalizadas que varían de 0 a 1, y otros entre -1 y 1. Dependiendo de la biblioteca gráfica que se utilice, el visor se define en coordenadas normalizadas o en coordenadas de pantalla después del proceso de normalización. En el último paso de la transformación de visualización, el contenido del visor se transfiere a las posiciones dentro de la ventana de visualización.

El recorte se realiza habitualmente en coordenadas normalizadas. Esto nos permite reducir los cálculos concatenando en primer lugar las múltiples matrices de transformación. Los procedimientos de recorte tienen una importancia fundamental en los gráficos por computador. Ellos se utilizan no sólo en las transformaciones de visualización, sino también en los sistemas de gestión de ventanas, en los paquetes de dibujo y pintura para borrar partes de una imagen, y en muchas otras aplicaciones.

## 6.2 LA VENTANA DE RECORTE

---

Para lograr un efecto de visualización particular en un programa de aplicación, podríamos diseñar nuestra propia ventana de recorte con cualquier forma, tamaño y orientación que elijamos. Por ejemplo, podríamos querer utilizar un patrón de estrellas, una elipse, o una figura con límites definidos mediante *splines* como ventana de recorte. Pero recortar una escena utilizando un polígono cóncavo o una ventana de recorte con límites no lineales requiere más procesamiento que recortar con un rectángulo. Necesitamos realizar más cálculos para determinar dónde un objeto intersecta con un círculo que para encontrar dónde intersecta con una línea recta. Las aristas de ventana más simples para recortar son las líneas rectas que son paralelas a los ejes de coordenadas. Por tanto, los paquetes gráficos habitualmente sólo permiten ventanas de recorte rectangulares alineadas con los ejes *x* e *y*.

Si queremos alguna otra forma para las ventanas de recorte, entonces debemos implementar algoritmos y transformaciones de coordenadas particulares. O podríamos simplemente editar la imagen para producir una cierta forma para el marco de visualización de la escena. Por ejemplo, podríamos adornar los bordes de una imagen con cualquier patrón superponiendo polígonos llenos con el color de fondo. De este modo, podríamos generar cualquier efecto en los bordes o incluso incluir agujeros interiores en la imagen.

Las ventanas de recorte rectangulares en posición estándar se definen fácilmente, proporcionando las coordenadas de dos esquinas opuestas del rectángulo. Si quisieramos obtener una vista rotada de una escena, podríamos definir una ventana de recorte en un sistema de coordenadas de visualización rotado o, de forma equivalente, podríamos rotar la escena en coordenadas universales. Algunos sistemas proporcionan opciones para seleccionar un marco de visualización bidimensional rotado pero, habitualmente, la ventana de recorte se debe definir en coordenadas universales.

## Ventana de recorte en coordenadas de visualización

Una forma general de realizar la transformación de visualización bidimensional consiste en establecer un *sistema de coordenadas de visualización*, dentro del sistema de coordenadas universales. El sistema de referencia de visualización proporciona una referencia para especificar una ventana de recorte rectangular con cualquier orientación y posición, como se muestra en la Figura 6.4. Para obtener una vista de la escena en el sistema de coordenadas universales como la determinada por la ventana de recorte de la Figura 6.4, sólo necesitamos transformar la descripción de la escena en coordenadas de visualización. Aunque muchos paquetes gráficos no proporcionan funciones para especificar una ventana de recorte en un sistema de coordenadas bidimensional de visualización, esta es la manera estándar de definir una región de recorte de una escena tridimensional.

Elegimos un origen para un sistema de coordenadas bidimensional de visualización en alguna posición  $\mathbf{P}_0 = (x_0, y_0)$  del universo, y podemos establecer la orientación empleando un vector  $\mathbf{V}$  en coordenadas universales que defina la dirección de  $y_{\text{vista}}$ . El vector  $\mathbf{V}$  se denomina **vector de orientación** bidimensional. Un método alternativo para especificar la orientación del sistema de referencia de visualización consiste en proporcionar un ángulo de rotación relativo al eje  $x$  o al eje  $y$  del sistema de referencia universal. A partir de este ángulo de rotación, podemos entonces obtener el vector de orientación. Una vez que hemos establecido los parámetros que definen el sistema de coordenadas de visualización, utilizamos los procedimientos de la Sección 5.8 para transformar la descripción de la escena al sistema de visualización. Esto implica una secuencia de transformaciones equivalente a superponer el sistema de referencia de visualización sobre el sistema universal.

El primer paso de la secuencia de transformaciones es trasladar el origen de visualización al origen universal. A continuación, rotamos el sistema de visualización para alinearlo con el sistema universal. Dado el vector de orientación  $\mathbf{V}$ , podemos calcular las componentes de los vectores unitarios  $\mathbf{v} = (v_x, v_y)$  y  $\mathbf{u} = (u_x, u_y)$  de los ejes  $y_{\text{view}}$  y  $x_{\text{view}}$ , respectivamente. Estos vectores unitarios se utilizan para formar la primera y la segunda fila de la matriz de rotación  $\mathbf{R}$  que alinea los ejes de visualización  $x_{\text{vista}}$  e  $y_{\text{vista}}$  con los ejes universales  $x_w$  e  $y_w$ .

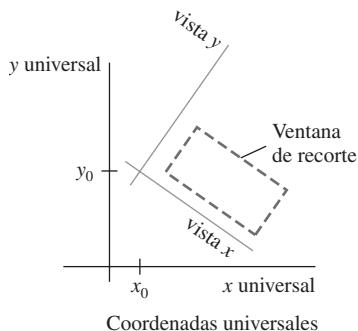
Las posiciones de los objetos en coordenadas universales se convierten entonces en coordenadas de visualización con la matriz compuesta de transformación bidimensional

$$\mathbf{M}_{WC,VC} = \mathbf{R} \cdot \mathbf{T} \quad (6.1)$$

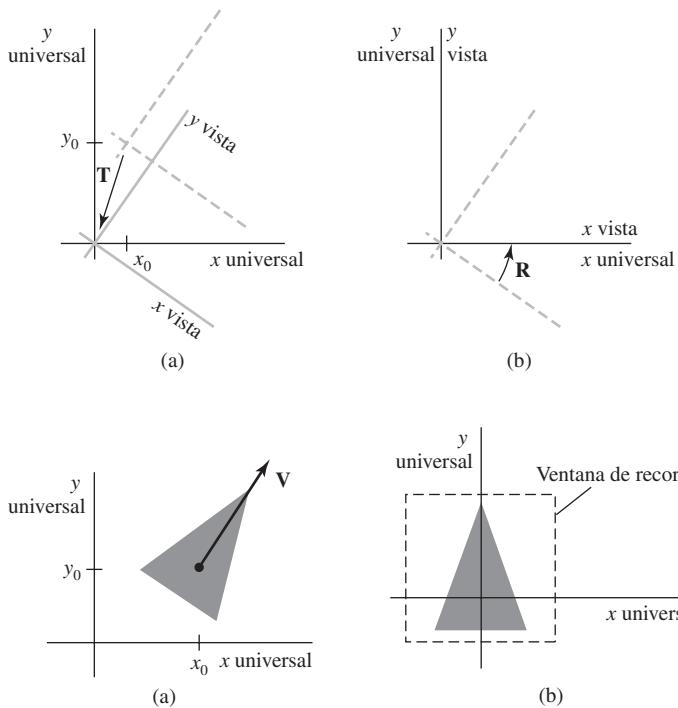
donde  $\mathbf{T}$  es la matriz de traslación que lleva el origen de visualización  $\mathbf{P}_0$  hasta el origen universal, y  $\mathbf{R}$  es la matriz de rotación que rota el sistema de visualización hasta que coincide con el sistema de coordenadas universales. La Figura 6.5 muestra los pasos de esta transformación de coordenadas.

## Ventana de recorte en coordenadas universales

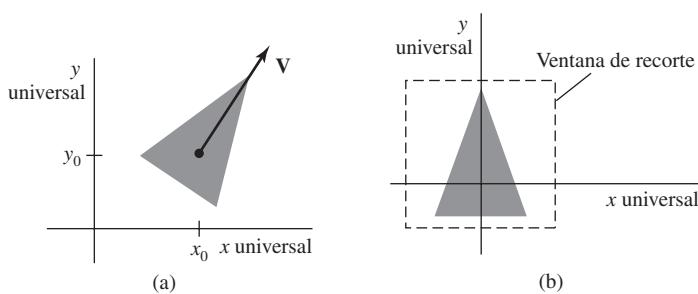
En una biblioteca de programación de gráficos se proporciona habitualmente una subrutina para definir una ventana de recorte estándar en coordenadas universales. Simplemente especificamos dos posiciones en coor-



**FIGURA 6.4.** Una ventana de recorte rotada definida en coordenadas de visualización.



**FIGURA 6.5.** Un sistema de coordenadas de visualización se mueve para que coincida con el sistema de referencia universal (a) aplicando una matriz de traslación  $T$  para mover el origen de visualización hasta el origen universal, después (b) aplicando una matriz de rotación  $R$  para alinear los ejes de los dos sistemas.



**FIGURA 6.6.** Un triángulo (a), con un punto de referencia seleccionado y un vector de orientación, se traslada y se rota hasta la posición (b) dentro de una ventana de recorte.

nadas universales, que se asignan entonces a las dos esquinas opuestas de un rectángulo estándar. Una vez que se ha establecido la ventana de recorte, la descripción de la escena se procesa a través de las subrutinas de visualización hacia el dispositivo de salida.

Si queremos obtener una vista rotada de una escena bidimensional, como se estudió en la sección anterior, realizaremos exactamente los mismos pasos que se describieron allí, pero sin considerar un sistema de referencia de visualización. Por tanto, simplemente rotamos (y posiblemente trasladamos) objetos a la posición deseada y establecemos la ventana de recorte, todo en coordenadas universales. A modo de ejemplo, podríamos visualizar la vista rotada de un triángulo de la Figura 6.6(a) rotándola hasta la posición que queramos y estableciendo un rectángulo de recorte estándar. Análogamente a la transformación de coordenadas descrita en la sección anterior, podríamos también trasladar el triángulo al origen universal y definir una ventana de recorte alrededor del triángulo. En ese caso, definimos un vector de orientación y elegimos un punto de referencia tal como el centroide del triángulo (Apéndice A). Despues trasladamos el punto de referencia hasta el origen universal y rotamos el vector de orientación hasta el eje  $y_{\text{universal}}$  utilizando la matriz 6.1. Con el triángulo en la posición deseada, podemos utilizar una ventana de recorte estándar en coordenadas universales para capturar la vista del triángulo rotado. La posición transformada del triángulo y la ventana de recorte seleccionada se muestran en la Figura 6.6(b).

## 6.3 TRANSFORMACIONES DE VISOR Y NORMALIZACIÓN

En algunos paquetes gráficos, la normalización y las transformaciones de ventura a visor se combinan en una única operación. En este caso, las coordenadas del visor se proporcionan a menudo dentro del rango que varía desde 0 a 1 para que se posicione dentro de un cuadrado unidad. Despues del recorte, el cuadrado unidad que contiene el visor se mapea al dispositivo de salida de visualización. En otros sistemas, las subrutinas de normalización y de recorte se aplican antes de la transformación del visor. En estos sistemas, los límites del visor se especifican en coordenadas de pantalla relativas a la posición de la ventana de visualización.

## Mapeo de la ventana de recorte en un visor normalizado

Para ilustrar los procedimientos generales de las transformaciones de visores y de normalización, en primer lugar consideramos un visor definido con valores de coordenadas normalizadas comprendidos entre 0 y 1. Las descripciones de los objetos se transfieren a este espacio normalizado utilizando una transformación que mantiene el mismo emplazamiento relativo de un punto en el visor que cuando estaba en la ventana de recorte. Si una posición de coordenadas está en el centro de la ventana de recorte, por ejemplo, ésta se mapearía al centro del visor. La Figura 6.7 muestra este mapeo ventana a visor. La posición  $(xw, yw)$  de la ventana de recorte se mapea a la posición  $(xv, yv)$  del visor asociado.

Para transformar el punto en coordenadas universales en la misma posición relativa dentro del visor, necesitamos que:

$$\begin{aligned}\frac{xv - xv_{\min}}{xv_{\max} - xv_{\min}} &= \frac{xw - xw_{\min}}{xw_{\max} - xw_{\min}} \\ \frac{yv - yv_{\min}}{yv_{\max} - yv_{\min}} &= \frac{yw - yw_{\min}}{yw_{\max} - yw_{\min}}\end{aligned}\quad (6.2)$$

Resolviendo estas expresiones para la posición de vista  $(xv, yv)$ , tenemos,

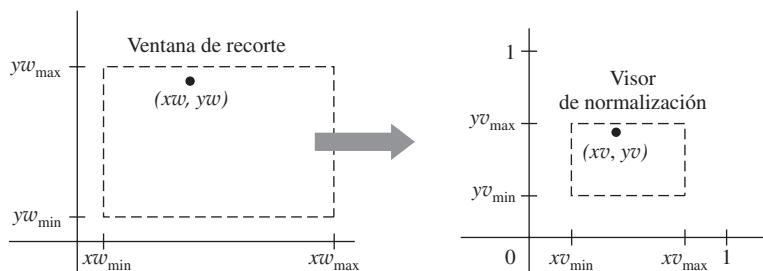
$$\begin{aligned}xv &= s_x xw + t_x \\ yv &= s_y yw + t_y\end{aligned}\quad (6.3)$$

donde los factores de escala son:

$$\begin{aligned}s_x &= \frac{xv_{\max} - xv_{\min}}{xw_{\max} - xw_{\min}} \\ s_y &= \frac{yv_{\max} - yv_{\min}}{yw_{\max} - yw_{\min}}\end{aligned}\quad (6.4)$$

y los factores de traslación son:

$$\begin{aligned}t_x &= \frac{xw_{\max} xv_{\min} - xw_{\min} xv_{\max}}{xw_{\max} - xw_{\min}} \\ t_y &= \frac{yw_{\max} yv_{\min} - yw_{\min} yv_{\max}}{yw_{\max} - yw_{\min}}\end{aligned}\quad (6.5)$$



**FIGURA 6.7.** Un punto  $(xw, yw)$  de una ventana de recorte en coordenadas universales se mapea a las coordenadas de visor  $(xv, yv)$ , dentro de un cuadrado unidad, para que las posiciones relativas de los dos puntos en sus respectivos rectángulos sean las mismas.

Ya que simplemente mapeamos las posiciones en coordenadas universales a un visor que está posicionado cerca del origen universal, podemos deducir las Ecuaciones 6.3 utilizando cualquier secuencia de transformaciones que convierta el rectángulo de la ventana de recorte en el rectángulo del visor. Por ejemplo, podríamos obtener la transformación de las coordenadas universales a las coordenadas del visor con la secuencia:

- (1) Cambie de escala la ventana de recorte al tamaño del visor utilizando un punto fijo ( $xw_{\min}, yw_{\min}$ ).
- (2) Traslade ( $xw_{\min}, yw_{\min}$ ) a ( $xv_{\min}, yv_{\min}$ ).

La transformación de cambio de escala del paso (1) se puede representar con la matriz bidimensional:

$$\mathbf{S} = \begin{bmatrix} s_x & 0 & xw_{\min}(1-s_x) \\ 0 & s_y & yw_{\min}(1-s_y) \\ 0 & 0 & 1 \end{bmatrix} \quad (6.6)$$

donde  $s_x$  y  $s_y$  son las mismas que en las Ecuaciones 6.4. La matriz bidimensional que representa la traslación de la esquina inferior izquierda de la ventana de recorte, hasta la esquina inferior izquierda del visor es:

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & xv_{\min} - sw_{\min} \\ 0 & 1 & yv_{\min} - yw_{\min} \\ 0 & 0 & 1 \end{bmatrix} \quad (6.7)$$

Y la matriz compuesta que representa la transformación al visor normalizado es

$$\mathbf{M}_{\text{ventana, visornorm}} = \mathbf{T} \cdot \mathbf{S} = \begin{bmatrix} s_x & 0 & t_x \\ 0 & s_y & t_y \\ 0 & 0 & 1 \end{bmatrix} \quad (6.8)$$

la cual nos proporciona el mismo resultado que en las Ecuaciones 6.3. Cualquier otro punto de referencia de la ventana de recorte, tal como la esquina superior derecha o el centro de la ventana, se podría utilizar en las operaciones de cambio de escala y traslación. O, podríamos en primer lugar trasladar cualquier posición de la ventana de recorte hasta la posición correspondiente del visor y después realizar un cambio de escala relativo a la posición del visor.

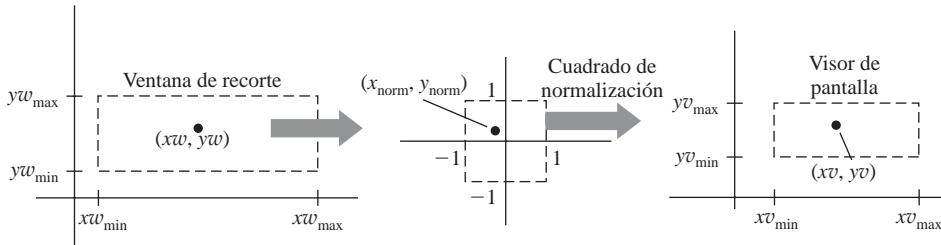
La transformación ventana a visor mantiene el emplazamiento relativo de las descripciones de los objetos. Un objeto situado dentro de la ventana de recorte se mapea a una posición correspondiente dentro del visor. De forma similar, un objeto situado fuera de la ventana de recorte está fuera del visor.

Por otra parte, las proporciones relativas de los objetos se mantienen sólo si la relación de aspecto del visor es la misma que la relación de aspecto de la ventana de recorte. En otras palabras, mantenemos las mismas proporciones de los objetos si los factores de escala  $s_x$  y  $s_y$  son iguales. De otro modo, los objetos universales se alargarán o contraerán en la dirección del eje  $x$  o del eje  $y$  (o en ambas direcciones) cuando se visualizan en el dispositivo de salida.

Las subrutinas de recorte se pueden aplicar utilizando los límites de la ventana de recorte o los límites del visor. Después del recorte, las coordenadas normalizadas se transforman en coordenadas del dispositivo. El cuadrado unidad se puede mapear al dispositivo de salida utilizando los mismos procedimientos que en la transformación ventana a visor, transfiriendo la zona interior del cuadrado unidad a toda la zona de visualización del dispositivo de salida.

### Mapeo de la ventana de recorte a un cuadrado normalizado

Otro enfoque de la visualización bidimensional consiste en transformar la ventana de recorte en un cuadrado normalizado, recortar en coordenadas normalizadas y después transferir la descripción de la escena a un visor



**FIGURA 6.8.** Un punto  $(x_w, y_w)$  de la ventana de recorte se mapea a una posición definida mediante coordenadas normalizadas  $(x_{\text{norm}}, y_{\text{norm}})$ , después a una posición definida mediante coordenadas de pantalla  $(x_v, y_v)$  en un visor. Los objetos se recortan con el cuadrado de normalización antes de la transformación a coordenadas de visor.

especificado en coordenadas de pantalla. Esta transformación se ilustra en la Figura 6.8 con coordenadas normalizadas dentro del rango que varía desde  $-1$  hasta  $1$ . Los algoritmos de recorte en esta secuencia de transformación se estandarizan ahora para que los objetos situados fuera de los límites se detecten y sean eliminados de la descripción de la escena. En el paso final de la transformación de visualización, los objetos contenidos en el visor se posicionan dentro de la ventana de visualización.

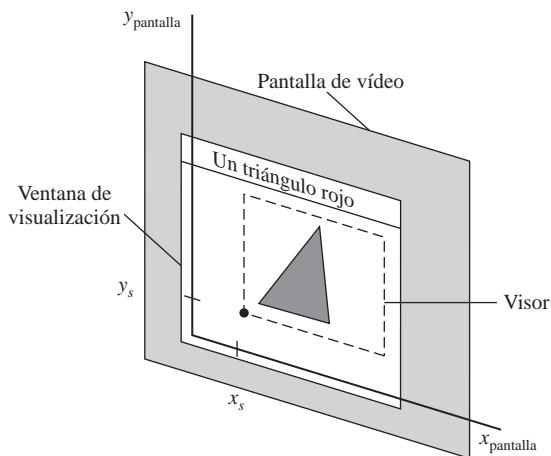
Transferimos los contenidos de la ventana de recorte al cuadrado de normalización utilizando los mismos procedimientos de la transformación ventana a vista. La matriz de transformación de normalización se obtiene a partir de la Ecuación 6.8 sustituyendo  $-1$  por  $x_{\text{vmin}}$  e  $y_{\text{vmin}}$  y sustituyendo  $+1$  por  $x_{\text{vmax}}$  e  $y_{\text{vmax}}$ . Haciendo estas sustituciones en las expresiones con  $t_x$ ,  $t_y$ ,  $s_x$  y  $s_y$ , tenemos,

$$\mathbf{M}_{\text{ventana, cuadradonorm}} = \begin{bmatrix} \frac{2}{x_{\text{wmax}} - x_{\text{wmin}}} & 0 & -\frac{x_{\text{wmax}} + x_{\text{wmin}}}{x_{\text{wmax}} - x_{\text{wmin}}} \\ 0 & \frac{2}{y_{\text{wmax}} - y_{\text{wmin}}} & -\frac{y_{\text{wmax}} + y_{\text{wmin}}}{y_{\text{wmax}} - y_{\text{wmin}}} \\ 0 & 0 & 1 \end{bmatrix} \quad (6.9)$$

De forma similar, después de que los algoritmos de recorte se hayan aplicado, el cuadrado normalizado con la arista de longitud igual a  $2$  se transforma en un visor especificado. Esta vez, obtenemos la matriz de transformación a partir de la Ecuación 6.8 sustituyendo  $-1$  por  $x_{\text{wmin}}$  e  $y_{\text{wmin}}$  y sustituyendo  $+1$  por  $x_{\text{wmax}}$  e  $y_{\text{wmax}}$ :

$$\mathbf{M}_{\text{cuadradonorm, visor}} = \begin{bmatrix} \frac{x_{\text{vmax}} - x_{\text{vmin}}}{2} & 0 & \frac{x_{\text{vmax}} + x_{\text{vmin}}}{2} \\ 0 & \frac{y_{\text{vmax}} - y_{\text{vmin}}}{2} & \frac{y_{\text{vmax}} + y_{\text{vmin}}}{2} \\ 0 & 0 & 1 \end{bmatrix} \quad (6.10)$$

El último paso del proceso de visualización es posicionar el área de la vista en la ventana de visualización. Habitualmente, la esquina inferior izquierda de la vista se sitúa en las coordenadas especificadas relativas a la esquina inferior izquierda de la ventana de visualización. La Figura 6.9 muestra el posicionamiento de una vista dentro de una ventana de visualización.



**FIGURA 6.9.** Una vista en las coordenadas  $(x_s, y_s)$  dentro de la ventana de visualización.

Como anteriormente, mantenemos las proporciones iniciales de los objetos eligiendo la relación de aspecto de la vista igual que la de la ventana de recorte. De otro modo, los objetos se伸展 o se contraerán en la dirección del eje  $x$  o del eje  $y$ . También, la relación de aspecto de la ventana de visualización puede afectar a las proporciones de los objetos. Si el visor se mapea a toda la ventana de visualización y el tamaño de la ventana de visualización se cambia, los objetos se pueden distorsionar a menos que la relación de aspecto del visor también se ajuste.

## Visualización de cadenas de caracteres

Las cadenas de caracteres se pueden manipular de dos formas cuando se mapean a un visor mediante la pipeline de visualización. El mapeo más simple mantiene un tamaño de carácter constante. Este método se podría emplear con patrones de caracteres de mapas de bits. Pero las fuentes de contorno se podrían transformar del mismo modo que otras primitivas; sólo será necesario transformar las posiciones de definición de los segmentos de línea de las formas de los caracteres de contorno. Los algoritmos de determinación de los patrones de los píxeles de los caracteres transformados, se aplican después cuando se procesan las otras primitivas de la escena.

## Efectos de división de pantalla y múltiples dispositivos de salida

Seleccionando diferentes ventanas de recorte y sus visores asociados en una escena, podemos proporcionar visualizaciones simultáneas de dos o más objetos, múltiples partes de la imagen, o vistas diferentes de una única escena. Podemos posicionar estas vistas en diferentes partes de una única ventana de visualización o en múltiples ventanas de visualización en la pantalla. En una aplicación de diseño, por ejemplo, podemos visualizar una vista de malla de alambre de un objeto en una vista, mientras también visualizamos una vista totalmente sombreada del objeto en otro visor. Podríamos enumerar otra información o menús en un tercer visor.

También es posible que se puedan utilizar dos o más dispositivos de salida de forma concurrente en un sistema concreto, y que podamos establecer un par de ventana de recorte/visor en cada dispositivo de salida. Un mapeo a un dispositivo de salida seleccionado a veces se denomina **transformación de estación de trabajo**. En este caso, los visores se podrían especificar en las coordenadas de un dispositivo de visualización particular. O cada visor se podría especificar dentro de un cuadrado unidad, que se mapea después al dispositivo de salida elegido. Algunos sistemas gráficos proporcionan un par de funciones de estación de trabajo para este propósito. Una función se utiliza para designar una ventana de recorte para un dispositivo de salida seleccionado, identificado por un *número de estación de trabajo*, y la otra función se utiliza para establecer el visor asociado de dicho dispositivo.

## 6.4 FUNCIONES DE VISUALIZACIÓN BIDIMENSIONAL DE OpenGL

---

Realmente, la biblioteca básica de OpenGL no posee funciones específicas para visualización bidimensional, ya que está diseñada principalmente para aplicaciones tridimensionales. Pero podemos adaptar las subrutinas de visualización tridimensional a una escena bidimensional, y la biblioteca de núcleo contiene una función de visor. Además, OpenGL Utility (GLU) proporciona una función bidimensional para especificar la ventana de recorte, y tenemos funciones de GLUT para manipular ventanas de visualización. Por tanto, podemos utilizar estas subrutinas bidimensionales, junto con la función de visor de OpenGL, para todas las operaciones de visualización que necesitemos.

### Modo de proyección de OpenGL

Antes de que seleccionemos una ventana de recorte y un visor en OpenGL, necesitamos establecer el modo apropiado para construir la matriz de transformación de coordenadas universales a coordenadas de pantalla. Con OpenGL, no podemos establecer un sistema de coordenadas de visualización bidimensional independiente como el de la Figura 6.4, y debemos establecer los parámetros de la ventana de recorte como una parte de la transformación de proyección. Por tanto, en primer lugar hay que seleccionar el modo de proyección. Hacemos esto con la misma función que utilizamos para establecer el modo de vista de modelo para las transformaciones geométricas. Los comandos posteriores de definición de una ventana de recorte y un visor se aplicarán entonces a la matriz de proyección.

```
glMatrixMode (GL_PROJECTION);
```

Esta función designa la matriz de proyección como la matriz actual, que se establece inicialmente como una matriz identidad. Sin embargo, si vamos a volver a ejecutar esta línea en otras vistas de la escena, también podemos establecer la inicialización con

```
glLoadIdentity ( );
```

Esto asegura que cada vez que entremos en el modo de proyección, la matriz se reinicializará con la matriz identidad, de modo que los nuevos parámetros de visualización no se combinen con los anteriores.

### Función de ventana de recorte de GLU

Para definir una ventana de recorte bidimensional, podemos utilizar la función de OpenGL Utility:

```
gluOrtho2D (xwmin, xwmax, ywmin, ywmax);
```

Las coordenadas de los límites de la ventana de recorte se proporcionan como números de doble precisión. Esta función especifica una proyección ortogonal para mapear la escena a la pantalla. En una escena tridimensional, esto significa que los objetos se proyectarían según líneas paralelas que son perpendiculares a la pantalla de visualización bidimensional *xy*. Pero en una aplicación bidimensional, los objetos ya están definidos en el plano *xy*. Por tanto, la proyección ortogonal no tiene efecto sobre nuestra escena bidimensional salvo convertir las posiciones de los objetos en coordenadas normalizadas. No obstante, debemos especificar la proyección ortogonal porque nuestra escena bidimensional se procesa a través de la *pipeline* de visualización de OpenGL que es totalmente tridimensional. De hecho, podríamos especificar la ventana de recorte utilizando la versión tridimensional de la biblioteca de núcleo de OpenGL de la función *gluOrtho2D* (Sección 7.10).

Las coordenadas normalizadas en el rango que varía de -1 hasta 1 se utilizan en las subrutinas de recorte OpenGL. La función *gluOrtho2D* establece una versión tridimensional de la matriz de transformación 6.9 para el mapeo de objetos dentro de la ventana de recorte a coordenadas normalizadas. Los objetos situados fuera del cuadrado normalizado (y fuera de la ventana de recorte) se eliminan de la escena que se va a mostrar.

Si no especificamos una ventana de recorte en un programa de aplicación, las coordenadas predeterminadas son  $(xw_{\min}, yw_{\min}) = (-1.0, -1.0)$  y  $(xw_{\max}, yw_{\max}) = (1.0, 1.0)$ . Por tanto, la ventana de recorte pre-

determinada es el cuadrado normalizado centrado en el origen de coordenadas con una longitud de lado de 2.0.

## Función de visor de OpenGL

Especificamos los parámetros del visor con la función OpenGL

```
glViewport (xvmin, yvmin, vpWidth, vpHeight);
```

donde todos los valores de los argumentos se proporcionan en coordenadas de pantalla enteras relativas a la ventana de visualización. Los argumentos `xvmin` e `yvmin` especifican la posición de la esquina inferior izquierda del visor respecto a la esquina inferior izquierda de la ventana de visualización. El ancho y la altura en píxeles del visor se establecen con los argumentos `vpWidth` y `vpHeight`. Si no se invoca la función `glViewport` en un programa, el tamaño y la posición predeterminados del visor son los mismos que los de la ventana de visualización.

Después de que se han aplicado las subrutinas de recorte, las posiciones dentro del cuadrado normalizado se transforman en el rectángulo de visor utilizando la matriz 6.10. Las coordenadas de la esquina superior derecha del visor se calculan para esta matriz de transformación en función del ancho y de la altura del visor:

$$xv_{\max} = xv_{\min} + vpWidth, \quad yv_{\max} = yv_{\min} + vpHeight \quad (6.11)$$

En la transformación final, los colores de los píxeles de las primitivas dentro del visor se cargan en el búfer de refresco en las posiciones de pantalla especificadas.

Se pueden crear múltiples visores en OpenGL para una gran variedad de aplicaciones (Sección 6.3). Podemos obtener los parámetros del visor activo actualmente utilizando la función de consulta:

```
glGetIntegerv (GL_VIEWPORT, vpArray);
```

donde `vpArray` es una matriz de un único índice y cuatro elementos. Esta función `Get` devuelve los parámetros del visor actual en `vpArray` en el orden `xvmin`, `yvmin`, `vpWidth` y `vpHeight`. En una aplicación interactiva, por ejemplo, podemos utilizar esta función para obtener los parámetros del visor que contiene el cursor de pantalla.

## Creación de una ventana de visualización con GLUT

En la Sección 2.9, presentamos brevemente algunas de las funciones de la biblioteca GLUT. Ya que la biblioteca GLUT actúa como interfaz con cualquier sistema de gestión de ventanas, usamos las subrutinas de GLUT para la creación y la manipulación de ventanas de visualización con el fin de que nuestros programas de ejemplo sean independientes de cualquier máquina específica. Para acceder a estas subrutinas, necesitamos en primer lugar inicializar GLUT con la siguiente función.

```
glutInit (&argc, argv);
```

Los argumentos de esta función de inicialización son los mismos que los del procedimiento `main`, y podemos utilizar `glutInit` para procesar argumentos de la línea de comandos.

Disponemos de tres funciones en GLUT para definir la ventana de visualización y elegir sus dimensiones y su posición:

```
glutInitWindowPosition (xTopLeft, yTopLeft);
glutInitWindowSize (dwWidth, dwHeight);
glutCreateWindow ("Title of Display Window");
```

La primera de estas funciones proporciona la posición en coordenadas de pantalla enteras de la esquina superior izquierda de la ventana de visualización, relativa a la esquina superior izquierda de la pantalla. Si ninguna coordenada es negativa, el sistema de gestión de ventanas determina la posición de la ventana de visualización.

lización en la pantalla. Con la segunda función, seleccionamos el ancho y la altura de la ventana de visualización en píxeles enteros positivos. Si no utilizamos estas dos funciones para especificar el tamaño y la posición, el tamaño predeterminado es 300 por 300 y la posición predeterminada es  $(-1, -1)$ , que cede el posicionamiento de la ventana de visualización al sistema gestor de ventanas. En cualquier caso, el tamaño y la posición de la ventana de visualización especificados con subrutinas de GLUT se podrían ignorar, dependiendo del estado o de los otros requisitos actualmente activos en el sistema gestor de ventanas. Por tanto, el sistema de ventanas podría dimensionar y posicionar la ventana de visualización de una forma diferente. La tercera función crea la ventana de visualización, con la posición y tamaño especificados, y asigna el título, aunque el uso del título también depende del sistema de ventanas. A estas alturas, la ventana de visualización está definida pero no se muestra en la pantalla hasta que se han completado todas las operaciones de configuración de GLUT.

## Establecimiento del modo y del color de la ventana de visualización con GLUT

Con la siguiente función de GLUT se seleccionan varios parámetros de la ventana de visualización

```
glutInitDisplayMode (mode);
```

Utilizamos esta función para elegir un modo de color (RGB o indexado) y las diferentes combinaciones de los búferes. Los parámetros seleccionados se combinan con la operación lógica `or`. El modo predeterminado es búfer simple y modo de color RGB (o RGBA), que es el mismo que se obtendría con la siguiente instrucción:

```
glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
```

La especificación de color GLUT\_RGB es equivalente a GLUT\_RGBA. Un color de fondo para la ventana de visualización se elige con la subrutina OpenGL

```
glClearColor (red, green, blue, alpha);
```

En el modo de color indexado, establecemos el color de la ventana de visualización con

```
glClearColor (index);
```

donde al argumento `index` se le asigna un valor entero que se corresponde con una posición de la tabla de colores.

## Identificador de la ventana de visualización con GLUT

Se pueden crear múltiples ventanas de visualización para una aplicación, y a cada una se le asigna un entero positivo, el **identificador de la ventana de visualización**, que comienza por el valor 1 para la primera ventana que se cree. A la vez que iniciamos una ventana de visualización, podemos registrar su identificador con la instrucción

```
windowID = glutCreateWindow ("A Display Window");
```

Una vez que hemos guardado el identificador entero de la ventana de visualización en la variable `windowID`, podemos utilizar el número del identificador para cambiar los parámetros de visualización o para borrar la ventana de visualización.

## Borrado de una ventana de visualización con GLUT

La biblioteca GLUT también incluye una función para borrar una ventana de visualización que hayamos creado. Si conocemos el identificador de la ventana de visualización, podemos eliminarla con la siguiente instrucción

```
glutDestroyWindow (windowID);
```

## Ventana de visualización actual con GLUT

Cuando especificamos cualquier operación de ventana de visualización, ésta se aplica a la **ventana de visualización actual**, que es la última ventana de visualización que hemos creado o la que seleccionamos con el siguiente comando.

```
glutSetWindow (windowID);
```

En cualquier momento, podemos preguntar al sistema cuál es la ventana de visualización actual:

```
currentWindowID = glutGetWindow ( );
```

Esta función devuelve el valor 0 si no hay ventanas de visualización o si la ventana de visualización actual se destruyó.

## Reposición y cambio de tamaño de una ventana de visualización con GLUT

Podemos cambiar la posición en pantalla de la ventana de visualización actual con:

```
glutPositionWindow (xNewTopLeft, yNewTopLeft);
```

donde las coordenadas especifican la nueva posición de la esquina superior izquierda de la ventana de visualización respecto a la esquina superior izquierda de la pantalla. De forma similar, la siguiente función cambia el tamaño de la ventana de visualización actual:

```
glutReshapeWindow (dwNewWidth, dwNewHeight);
```

Y con el siguiente comando, podemos ampliar la ventana de visualización actual para que ocupe toda la pantalla.

```
glutFullScreen ( );
```

El tamaño exacto de la ventana de visualización después de la ejecución de esta subrutina depende del sistema de gestión de ventanas. Una llamada posterior a `glutPositionWindow` o `glutReshapeWindow` cancelará la petición de expansión a tamaño de pantalla completa.

Cada vez que se cambia el tamaño de la ventana de visualización, su relación de aspecto puede cambiar y los objetos se pueden distorsionar. Como indicamos en la Sección 3.24, podemos reaccionar frente a un cambio de las dimensiones de la ventana de visualización utilizando la siguiente instrucción:

```
glutReshapeFunc (winReshapeFcn);
```

Esta subrutina de GLUT se activa cuando se cambia el tamaño de una ventana de visualización y se pasan el nuevo ancho y la nueva altura a su argumento: la función `winReshapeFcn`, en este ejemplo. Por tanto, `winReshapeFcn` es la «función de atención a evento» para el «evento cambio de forma». Podemos entonces utilizar esta función de atención a evento para cambiar los parámetros del visor para que la relación de aspecto original de la escena se mantenga. Además, podríamos también cambiar los límites de la ventana de recorte, cambiar el color de la ventana de visualización, modificar otros parámetros de visualización y realizar cualesquier otras tareas.

## Gestión de múltiples ventanas de visualización con GLUT

La biblioteca GLUT también tiene un gran número de subrutinas para manipular una ventana de visualización de formas diversas. Estas subrutinas son particularmente útiles cuando tenemos múltiples ventanas de visualización en la pantalla y queremos redistribuirlas o localizar una ventana de visualización particular.

Utilizamos la siguiente subrutina para convertir la ventana de visualización actual en un ícono en forma de imagen pequeña o un símbolo que representa la ventana.

```
glutIconifyWindow ( );
```

Este ícono tendrá una etiqueta con el mismo nombre que asignamos a la ventana, pero podemos cambiar el nombre del ícono con:

```
glutSetIconTitle ("Icon Name");
```

Podemos cambiar el nombre de la ventana de visualización con un comando similar:

```
glutSetWindowTitle ("New Window Name");
```

Cuando están abiertas múltiples ventanas de visualización en la pantalla, algunas de ellas se pueden solapar con otras ventanas de visualización. Podemos seleccionar cualquier ventana de visualización para pasárla delante de todas las otras ventanas designado en primer lugar ésta como ventana actual, y entonces ejecutar el comando «traer al frente ventana»:

```
glutSetWindow (windowID);
glutPopWindow ( );
```

De un modo similar, podemos «empujar» la ventana de visualización actual hacia atrás, para que quede detrás de las restantes ventanas de visualización. Esta secuencia de operaciones es:

```
glutSetWindow (windowID);
glutPushWindow ( );
```

También podemos quitar de la pantalla la ventana actual con:

```
glutHideWindow ( );
```

Y podemos volver a mostrar una ventana de visualización «oculta», o una que se haya convertido en un ícono, designando ésta como la ventana de visualización actual e invocando la función:

```
glutShowWindow ( );
```

## Subventanas de GLUT

Dentro de una ventana de visualización seleccionada, podemos establecer cualquier número de ventanas de visualización de segundo nivel, llamadas *subventanas*. Esto proporciona un medio para particionar ventanas de visualización en zonas de visualización diferentes. Creamos una subventana con la siguiente función.

```
glutCreateSubwindow (windowID, xBottomLeft, yBottomLeft, width, height);
```

El argumento *windowID* identifica la ventana de visualización en la que queremos establecer la subventana. Con el resto de los argumentos, especificamos su tamaño y la posición de la esquina inferior izquierda de la subventana relativa a la esquina inferior izquierda de la ventana de visualización.

A las subventanas se les asigna como identificador un número entero positivo del mismo modo que se numeran las ventanas de visualización de primer nivel. Podemos colocar una subventana dentro de otra subventana. También, a cada subventana se le puede asignar un modo de visualización individual y otros parámetros. Podemos incluso cambiar la forma, reposicionar, empujar hacia el fondo, traer a primer plano, ocultar y mostrar subventanas, del mismo modo que lo hacemos con las ventanas de visualización de primer nivel. Pero no podemos convertir una subventana creada con GLUT en un ícono.

## Selección de la forma del cursor de pantalla en una ventana de visualización

Podemos utilizar la siguiente subrutina de GLUT para solicitar una forma para el cursor de pantalla que hay que utilizar con la ventana actual.

```
glutSetCursor (shape);
```

Las formas posibles de cursor que podemos seleccionar son una punta de flecha con una dirección que elijamos, una flecha bidireccional, una flecha rotante, una cruz, un reloj de pulsera, una interrogación, o inclu-

so una calavera con huesos que se cruzan. Por ejemplo, podemos asignar la constante simbólica GLUT\_CURSOR\_UP\_DOWN al argumento `shape` para obtener una flecha bidireccional de arriba hacia abajo. Una flecha rotante se selecciona con GLUT\_CURSOR\_CYCLE, una forma de reloj de pulsera se selecciona con GLUT\_CURSOR\_WAIT, y una calavera con huesos que se cruzan se obtiene con la constante GLUT\_CURSOR\_DESTROY. Una forma de cursor se puede asignar a una ventana de visualización para indicar una clase particular de aplicación, tal como una animación. Sin embargo, las formas exactas que podemos utilizar dependen del sistema.

## Visualización de objetos gráficos en una ventana de visualización de GLUT

Después de que hayamos creado una ventana de visualización y seleccionado su posición, tamaño, color y otras características, indicamos qué hay que mostrar en dicha ventana. Si se ha creado más de una ventana de visualización, en primer lugar, designamos la que queramos como ventana de visualización actual. A continuación, invocamos la siguiente función para asignar algo a aquella ventana.

```
glutDisplayFunc (pictureDescription);
```

El argumento es una subrutina que describe lo que hay que mostrar en la ventana actual. Esta subrutina, llamada `pictureDescription`, en este ejemplo, se denomina *función de atención a evento*, ya que es la subrutina que se debe ejecutar cada vez que GLUT determina que los contenidos de la ventana de visualización se deberían renovar. La subrutina `pictureDescription` contiene habitualmente las primitivas y los atributos de OpenGL que definen una imagen, aunque ésta podría especificar otras estructuras tales como la visualización de un menú.

Si hemos establecido múltiples ventanas de visualización, entonces repetimos este proceso en cada ventana de visualización o subventana. También, podemos necesitar invocar `glutDisplayFunc` después del comando `glutPopWindow` si se ha dañado la ventana de visualización durante el proceso de redibujado de las ventanas. En este caso, la siguiente función se utiliza para indicar que los contenidos de la ventana de visualización actual se deberían renovar.

```
glutPostRedisplay ( );
```

Esta subrutina se utiliza también cuando hay que mostrar en una ventana de visualización un objeto adicional tal como un menú contextual.

## Ejecución del programa de aplicación

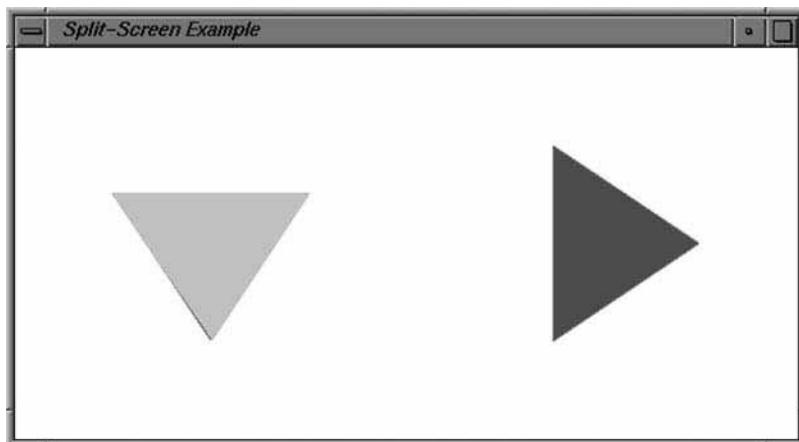
Cuando se ha completado la organización del programa y se han creado e inicializado las ventanas de visualización, necesitamos ejecutar el comando final de GLUT que señala la ejecución del programa:

```
glutMainLoop ( );
```

En este momento, las ventanas de visualización y sus contenidos gráficos se envían a la pantalla. El programa también entra en el **bucle de procesamiento de GLUT** que continuamente comprueba si se han producido eventos «nuevos», tales como una entrada interactiva procedente de un ratón o de una tableta gráfica.

## Otras funciones de GLUT

La biblioteca GLUT proporciona una gran variedad de subrutinas para gestionar procesos que dependen del sistema y para añadir características a la biblioteca básica de OpenGL. Por ejemplo, esta biblioteca contiene funciones para generar caracteres de mapas de bit y de contorno (Sección 3.21), y proporciona funciones para cargar valores en una tabla de color (Sección 4.3). Además, hay disponibles algunas funciones de GLUT, estudiadas en el Capítulo 8, que permiten visualizar objetos tridimensionales, con representación sólida o en su modelo alámbrico. Entre estos objetos se incluye una esfera, un toro y los cinco poliedros regulares (cubo, tetraedro, octaedro, dodecaedro e icosaedro).



**FIGURA 6.10.** Efecto de división de pantalla generado dentro de una ventana de visualización mediante el procedimiento `displayFunc`.

A veces es conveniente designar una función que se ha de ejecutar cuando no hay otros eventos para que el sistema los procese. Podemos hacer esto con:

```
glutIdleFunc (function);
```

El argumento de esta subrutina de GLUT podría hacer referencia a una función de segundo plano o a un procedimiento para actualizar los parámetros de una animación cuando no tienen lugar otros procesos.

También tenemos funciones de GLUT, estudiadas en el Capítulo 11, para obtener y procesar entrada interactiva y para crear y gestionar menús. GLUT proporciona subrutinas individuales para dispositivos de entrada tales como un ratón, un teclado, una tableta gráfica y un *spaceball*.

Finalmente, podemos utilizar la siguiente función para preguntar al sistema por alguno de los parámetros de estado actuales.

```
glutGet (stateParam);
```

Esta función devuelve un valor entero que se corresponde con la constante simbólica que hayamos seleccionado en su argumento. A modo de ejemplo, podemos obtener la coordenada *x* de la esquina superior izquierda de la ventana actual de visualización, relativa a la esquina superior izquierda de la pantalla, con la constante `GLUT_WINDOW_X`. Podemos obtener el ancho de la ventana actual de visualización o el ancho de la pantalla con `GLUT_WINDOW_WIDTH` o `GLUT_SCREEN_WIDTH`.

### Ejemplo de programa de visualización bidimensional en OpenGL

Como demostración del uso de la función de visor de OpenGL usamos un efecto de división de pantalla para mostrar dos vistas de un triángulo en el plano *xy* con su centroide en el origen de coordenadas universales. En primer lugar, se define una vista en la mitad izquierda de la ventana de visualización, y el triángulo original se muestra allí en color gris claro (color azul en el listado del programa de ejemplo). Utilizando la misma ventana de recorte, definimos otra vista en la mitad derecha de la ventana de visualización, y se cambia el color de relleno a gris más oscuro (rojo en el listado). El triángulo después se gira por su centroide y se muestra en el segundo visor. La Figura 6.10 muestra los dos triángulos mostrados con este programa de ejemplo.

```
#include <GL/glut.h>

class wcPt2D {
public:
    GLfloat x, y;
```

```

}

void init (void)
{
    /* Establece el color de la ventana visualización en blanco. */
    glClearColor (1.0, 1.0, 1.0, 0.0);

    /* Establece los parámetros de la ventana de recorte en
     * coordenadas universales. */
    glMatrixMode (GL_PROJECTION);
    gluOrtho2D (-100.0, 100.0, -100.0, 100.0);

    /* Establece el modo de construcción de la matriz de
     * transformación geométrica. */
    glMatrixMode (GL_MODELVIEW);
}

void triangle (wcPt2D *verts)
{
    GLint k;

    glBegin (GL_TRIANGLES);
    for (k = 0; k < 3; k++)
        glVertex2f (verts [k].x, verts [k].y);
    glEnd ( );
}

void displayFcn (void)
{
    /* Define la posición inicial del triángulo. */
    wcPt2D verts [3] = { {-50.0, -25.0}, {50.0, -25.0}, {0.0, 50.0} };

    glClear (GL_COLOR_BUFFER_BIT); // Borra la ventana de visualización.

    glColor3f (0.0, 0.0, 1.0); // Establece el color de relleno en azul.
    glViewport (0, 0, 300, 300); // Establece el visor izquierdo.
    triangle (verts); // Muestra el triángulo.

    /* Gira el triángulo y lo visualiza en la mitad derecha de la
     * ventana de visualización. */

    glColor3f (1.0, 0.0, 0.0); // Establece el color de relleno en rojo.
    glViewport (300, 0, 300, 300); // Establece el visor derecho.
    glRotatef (90.0, 0.0, 0.0, 1.0); // Gira alrededor del eje z.
    triangle (verts); // Muestra el triángulo rojo girado.

    glFlush ( );
}

```

```

void main (int argc, char ** argv)
{
    glutInit (&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    glutInitWindowPosition (50, 50);
    glutInitWindowSize (600, 300);
    glutCreateWindow («Split-Screen Example»);

    init ();
    glutDisplayFunc (displayFcn);

    glutMainLoop ( );
}

```

## 6.5 ALGORITMOS DE RECORTE

---

Generalmente, cualquier procedimiento que elimina aquellas porciones de una imagen que están dentro o fuera de una región del espacio especificada se denomina **algoritmo de recorte** o simplemente **recorte**. Habitualmente, una región de recorte es un rectángulo en posición estándar, aunque podríamos utilizar cualquier forma en una aplicación de recorte.

La aplicación de recorte más común está en la *pipeline* de visualización, donde el recorte se aplica para extraer una porción designada de la escena (bidimensional o tridimensional) para su visualización en un dispositivo de salida. Los métodos de recorte también se utilizan para suavizar los límites de los objetos, para construir objetos utilizando métodos de modelado de sólidos, gestionar entornos multiventana y para permitir que partes de una imagen se muevan, se copien o se borren en programas de dibujo y pintura.

Los algoritmos de recorte se aplican en procedimientos de visualización bidimensional para identificar aquellas partes de una imagen que están dentro de la ventana de recorte. Todo lo que se encuentra fuera de la ventana de recorte, se elimina de la descripción de la escena que se transfiere al dispositivo de salida para su visualización. Una implementación eficiente de recorte en la *pipeline* de visualización consiste en aplicar los algoritmos a los límites normalizados de la ventana de recorte. Esto reduce los cálculos, porque todas las matrices de transformación geométrica y de visualización se pueden concatenar y aplicar a una descripción de una escena antes de que el recorte se lleve a cabo. La escena recortada se puede después transferir a las coordenadas de pantalla para el procesamiento final.

En las secciones siguientes, tratamos los algoritmos bidimensionales para:

- Recorte de puntos
- Recorte de líneas (segmentos de línea recta)
- Recorte de áreas de relleno (polígonos)
- Recorte de curvas
- Recorte de texto

El recorte de puntos, líneas y polígonos es un componente estándar de los paquetes gráficos. Pero se pueden aplicar métodos similares a otros objetos, particularmente cónicas, tales como círculos, elipses y esferas, además de a curvas de tipo *spline* y a superficies. Habitualmente, sin embargo, los objetos con límites no lineales se aproximan mediante segmentos de línea recta o superficies de polígonos para reducir los cálculos.

A menos que se indique otra cosa, asumimos que la región de recorte es una ventana rectangular en posición estándar, con las aristas límite en las coordenadas  $xw_{\min}, xw_{\max}, yw_{\min}$  y  $yw_{\max}$ . Estas aristas límite habi-

tualmente se corresponden con un cuadrado normalizado, en el que los valores de  $x$  e  $y$  se encuentran dentro del rango que varía desde 0 a 1 o desde -1 a 1.

## 6.6 RECORTE DE PUNTOS BIDIMENSIONALES

---

En un rectángulo de recorte ubicado en la posición estándar, mantenemos un punto bidimensional  $\mathbf{P} = (x, y)$  para su visualización si se satisfacen las siguientes desigualdades:

$$\begin{aligned} x_{w_{\min}} \leq x \leq x_{w_{\max}} \\ y_{w_{\min}} \leq y \leq y_{w_{\max}} \end{aligned} \quad (6.12)$$

Si no se satisface una de estas cuatro ecuaciones, el punto se recorta (no se guarda para su visualización).

Aunque el recorte de puntos se aplica menos a menudo que el recorte de líneas o de polígonos, resulta útil en diversas situaciones, sobre todo cuando las imágenes se modelan como sistemas de partículas. Por ejemplo, se puede aplicar recorte de puntos a escenas que incluyan nubes, espuma de mar, humo o explosiones que estén modeladas mediante «partículas», tales como las coordenadas de los centros de pequeños círculos o esferas.

## 6.7 RECORTE DE LÍNEAS BIDIMENSIONALES

---

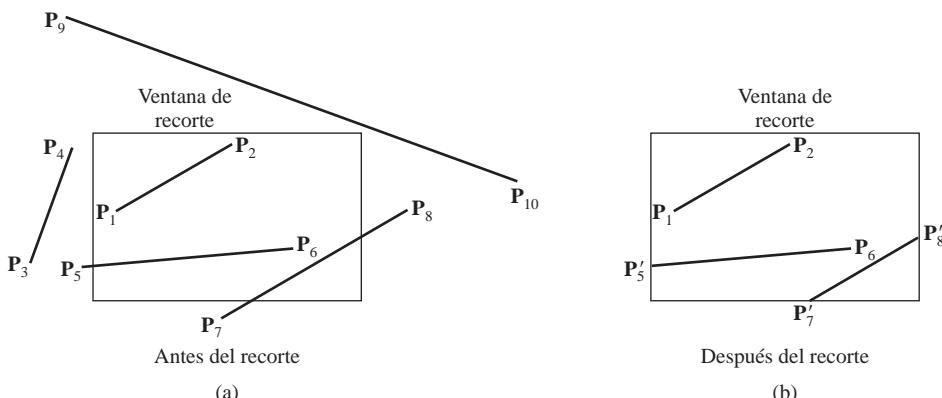
La Figura 6.11 muestra posibles posiciones de segmentos de línea recta en relación con una ventana de recorte estándar. Un algoritmo de recorte de líneas procesa cada línea de una escena mediante una serie de pruebas y cálculos de intersecciones para determinar si se debe guardar la línea completa o una parte de ésta. La parte más costosa de un procedimiento de recorte de líneas es el cálculo de las intersecciones de una línea con las aristas de la ventana. Por tanto, un objetivo importante en cualquier algoritmo de recorte de líneas consiste en minimizar los cálculos de intersecciones. Para ello, podemos realizar en primer lugar pruebas para determinar si un segmento de línea está completamente dentro de la ventana de recorte o está completamente fuera. Es sencillo determinar si una línea está completamente dentro de una ventana, pero es más difícil identificar todas las líneas que están completamente fuera de la ventana. Si somos incapaces de identificar si una línea está completamente dentro o completamente fuera de un rectángulo de recorte, debemos entonces realizar los cálculos de intersección para determinar si una parte de la línea cruza el interior de la ventana.

Comprobamos un segmento de línea para determinar si está completamente dentro o fuera del borde de una ventana de recorte seleccionada, aplicando las pruebas de recorte de puntos de la sección anterior. Cuando ambos puntos extremos de un segmento de línea están dentro de los cuatro límites de recorte, como la línea que va de  $\mathbf{P}_1$  a  $\mathbf{P}_2$  en la Figura 6.11, la línea está completamente dentro de la ventana de recorte y la guardamos. Cuando ambos puntos extremos de un segmento de línea se encuentran fuera de cualquiera de los cuatro límites ( $\mathbf{P}_3\mathbf{P}_4$  en la Figura 6.11), dicha línea se encuentra completamente fuera de la ventana y se elimina de la descripción de la escena. Pero ambas pruebas fallan, el segmento de línea intersecta con al menos un límite de recorte y puede o no cruzar el interior de la ventana de recorte.

Un modo de formular la ecuación de un segmento de línea recta consiste en utilizar la siguiente representación paramétrica, donde las coordenadas  $(x_0, y_0)$  y  $(x_{\text{fin}}, y_{\text{fin}})$  designan los dos puntos extremos de la línea.

$$\begin{aligned} x &= x_0 + u (x_{\text{fin}} - x_0) \\ y &= y_0 + u (y_{\text{fin}} - y_0) \quad 0 \leq u \leq 1 \end{aligned} \quad (6.13)$$

Podemos utilizar esta representación para determinar dónde un segmento de línea corta cada arista de la ventana de recorte, asignando el valor de la coordenada de cada arista a  $x$  o  $y$  y resolviendo para el paráme-



**FIGURA 6.11.** Recorte de segmentos de línea recta utilizando una ventana de recorte rectangular estándar.

tro  $u$ . A modo de ejemplo, el límite izquierda de la ventana está en la posición  $xw_{min}$ , por lo que sustituimos  $x$  por este valor y resolvemos para  $u$ , y calculamos el valor de intersección y correspondiente. Si este valor de  $u$  se encuentra fuera del rango que varía desde 0 a 1, el segmento de línea no intersecta con dicha arista de la ventana. Pero si el valor de  $u$  se encuentra dentro del rango que varía entre 0 y 1, parte de la línea se encuentra dentro de dicho borde. Podemos a continuación procesar esta porción interior del segmento de línea con respecto a los demás límites de recorte hasta que hayamos recortado la línea entera o encontramos una parte que esté dentro de la ventana.

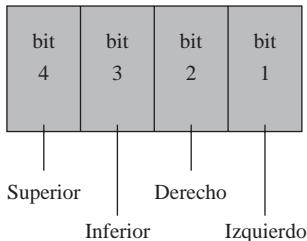
El procesamiento de segmentos de línea de una escena utilizando la sencilla técnica descrita en el párrafo anterior es directo, pero no muy eficiente. Es posible reformular la prueba inicial y los cálculos de intersección para reducir el tiempo de procesamiento de un conjunto de segmentos de línea. Se han desarrollado algoritmos de recorte de líneas más rápidos. Algunos de estos algoritmos se han diseñado explícitamente para imágenes bidimensionales y algunos se adaptan fácilmente a conjuntos de segmentos de línea tridimensionales.

### Recorte de líneas de Cohen-Sutherland

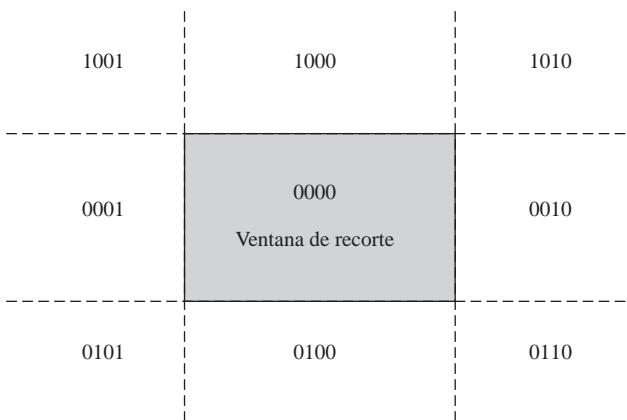
Éste es uno de los algoritmos más antiguos que se ha desarrollado para el recorte de líneas rápido, y variaciones de este método se utilizan ampliamente. El tiempo de procesamiento se reduce en el método de Cohen-Sutherland realizando más pruebas antes de proceder con los cálculos de las intersecciones. Inicialmente, se asigna a cada punto extremo de las líneas de una imagen un valor binario de cuatro dígitos llamado **código de región**. Cada bit se utiliza para indicar si está dentro o fuera de uno de los límites de la ventana de recorte. Podemos hacer referencia a las aristas de la ventana en cualquier orden. La Figura 6.12 muestra una posible ordenación en la que los bits están numerados de 1 a 4 de derecha a izquierda. Por tanto, para esta ordenación, el bit situado más a la derecha (bit 1) hace referencia al borde izquierdo de la ventana de recorte, y el situado más a la izquierda (bit 4) hace referencia al borde superior de la ventana. Un valor de 1 (o *verdadero*) en cualquier bit indica que el punto extremo está fuera de la ventana. De forma similar, un valor de 0 (o *falso*) en cualquier bit indica que el punto extremo no está fuera (está dentro o sobre) del límite correspondiente de la ventana. A veces, un código de región se denomina **código de «fuera»** porque un valor de 1 en cualquier bit indica que el punto del espacio está fuera del correspondiente borde de recorte.

Cada arista de la ventana de recorte divide el espacio bidimensional en un semiespacio interior y un semiespacio exterior. En total, los cuatro límites de la ventana crean nueve regiones. La Figura 6.13 enumera el valor del código binario en cada una de estas regiones. Por tanto, a un punto extremo que esté situado debajo y a la izquierda de la ventana de recorte se le asigna un código de región 0101, y el valor del código de región de cualquier punto interior a la ventana de recorte es 0000.

Los valores de los bits de un código de región se determinan comparando los valores de las coordenadas ( $x, y$ ) de un punto extremo con los límites de recorte. El bit 1 se pone a 1 si  $x < xw_{min}$ . Los valores de los otros



**FIGURA 6.12.** Una posible ordenación de los límites de la ventana de recorte correspondiente a las posiciones de los bits en el código de región de los puntos extremos del método Cohen-Sutherland.



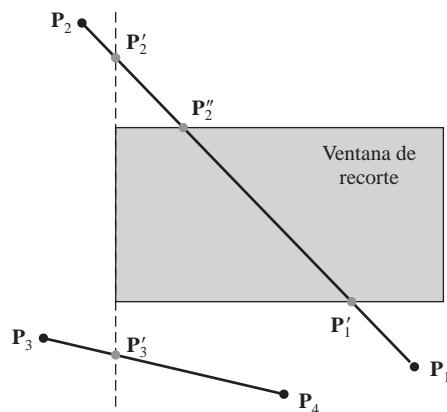
**FIGURA 6.13.** Los nueve códigos de región para la identificación de la posición de un punto extremo de una línea, relativos a los límites de la ventana de recorte.

tres bits se determinan de forma semejante. En lugar de utilizar pruebas de desigualdad, podemos determinar más eficientemente los valores de un código de región utilizando las operaciones de procesamiento de bits y siguiendo dos pasos: (1) Calcular las diferencias entre las coordenadas de los puntos extremos y los límites de recorte. (2) Utilizar el bit de signo resultante de cada cálculo de diferencia para cambiar el valor correspondiente de cada código de región. En el caso del esquema de ordenación mostrado en la Figura 6.12, el bit 1 es el bit de signo de  $x - x_{w_{\min}}$ ; el bit 2 es el bit de signo de  $x_{w_{\max}} - x$ ; el bit 3 es el bit de signo de  $y - y_{w_{\min}}$ ; y el bit 4 es el bit de signo de  $y_{w_{\max}} - y$ .

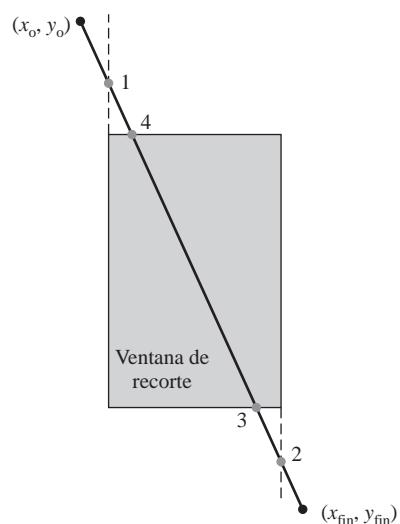
Una vez que hemos establecido los códigos de región de todos los puntos extremos de todas las líneas, podemos determinar rápidamente qué líneas se encuentran completamente dentro de la ventana de recorte y cuáles se encuentran claramente fuera. Cualesquier líneas que se encuentran completamente contenidas dentro de las aristas de la ventana tienen un código de región 0000 en ambos puntos extremos y estos segmentos de línea los guardamos. Cualquier línea que tenga un código de región de valor 1 en el mismo bit en cada punto extremo está completamente fuera del rectángulo de recorte, por lo que eliminamos dicho segmento de línea. A modo de ejemplo, una línea que tenga un código de región 1001 en un punto extremo y un código 0101 en el otro punto extremo está completamente a la izquierda de la ventana de recorte, como lo indica el valor 1 en el primer bit de cada código de región.

Podemos realizar las pruebas de dentro-fuera para los segmentos de línea utilizando operadores lógicos. Cuando la operación *or* entre los dos códigos de región de los puntos extremos de un segmento de línea es falsa (0000), la línea se encuentra dentro de la ventana de recorte. Por tanto, guardamos la línea y procedemos a comprobar la línea siguiente de la descripción de la escena. Cuando la operación *and* entre los dos códigos de región de los puntos extremos de una línea es verdadera (no 0000), la línea está completamente fuera de la ventana de recorte, y podemos eliminarla de la descripción de la escena.

En el caso de las líneas que no se pueden identificar como que están completamente dentro o completamente fuera de una ventana de recorte mediante las pruebas del código de región, se comprueba a continuación si intersectan con los límites de la ventana. Como se muestra en la Figura 6.14, los segmentos de línea pueden intersectar con los límites de recorte sin entrar dentro del interior de la ventana. Por tanto, para recortar un segmento de línea podrían ser necesarios varios cálculos de intersecciones, dependiendo del orden en



**FIGURA 6.14.** Las líneas que se extienden desde una región de la ventana de recorte a otra pueden atravesar la ventana de recorte o pueden intersectar con uno o más límites de recorte sin entrar en el interior de la ventana.



**FIGURA 6.15.** Los cuatro puntos de intersección (etiquetados de 1 a 4) de un segmento de línea que se recorta de acuerdo con los límites de la ventana en el orden izquierda, derecha, inferior, superior.

que procesemos los límites de recorte. Cuando procesamos cada arista de la ventana de recorte, se recorta una parte de la línea, y la parte que permanece de la línea se comprueba frente a los restantes límites de la ventana. Continuaremos eliminando partes hasta que la línea esté totalmente recortada o la parte que permanece de la línea se encuentre dentro de la ventana de recorte. En el siguiente estudio, asumimos que las aristas de la ventana se procesan en el orden: izquierda, derecha, inferior, superior. Para determinar si una línea cruza un límite de recorte seleccionado, podemos comprobar los valores correspondientes de los bits de los códigos de región de los dos puntos extremos. Si uno de estos bits es 1 y el otro es 0, el segmento de línea cruza dicho límite.

La Figura 6.14 muestra dos segmentos de línea que se pueden identificar inmediatamente como completamente dentro o completamente fuera de la ventana de recorte. Los códigos de región de la línea desde  $P_1$  a  $P_2$  son 0100 y 1001. Por tanto,  $P_1$  está dentro del límite izquierdo de recorte y  $P_2$  está fuera de dicho límite. A continuación, calculamos la intersección  $P'_2$ , y recortamos la parte de la línea desde  $P_2$  a  $P'_2$ . La parte que permanece de la línea se encuentra dentro de la línea límite derecha, y por ello a continuación comprobamos el límite inferior. El punto extremo  $P_1$  se encuentra por debajo de la arista inferior de recorte y  $P'_1$  se encuentra por encima de ésta, por lo que determinamos la intersección con esta arista ( $P'_1$ ). Eliminamos la parte de la línea desde  $P_1$  a  $P'_1$  y procedemos con la arista superior de la ventana. Allí determinamos la intersección  $P''_2$ . El último paso consiste en recortar la parte situada por encima del límite superior y guardar el segmento interior desde  $P'_1$  hasta  $P''_2$ . En el caso de la segunda línea, obtenemos que el punto  $P_3$  se encuentra fuera

del límite izquierdo y  $\mathbf{P}_4$  se encuentra dentro. Por tanto, calculamos la intersección  $\mathbf{P}'_3$  y eliminamos la parte de la línea que va desde  $\mathbf{P}_3$  a  $\mathbf{P}'_3$ . Comprobando los códigos de región de los puntos extremos  $\mathbf{P}'_3$  y  $\mathbf{P}_4$ , observamos que la parte que permanece de la línea se encuentra por debajo de la ventana de recorte y se puede eliminar también.

Cuando se recorta un segmento de línea utilizando esta técnica se puede calcular una intersección con los cuatro límites de recorte, dependiendo de cómo se procesen los puntos extremos de la línea y qué ordenación utilicemos en los límites. La Figura 6.15 muestra las cuatro intersecciones que se podrían calcular para un segmento de línea que se procesa frente a las aristas de la ventana de recorte en el orden izquierda, derecha, inferior, superior. Por tanto, se han desarrollado variaciones de esta técnica básica en un esfuerzo por reducir los cálculos de intersecciones.

Para determinar una intersección de un segmento de línea con la frontera, podemos utilizar la forma pendiente-punto de corte de la ecuación de la línea. Para una línea con coordenadas en sus puntos extremos  $(x_0, y_0)$  y  $(x_{\text{fin}}, y_{\text{fin}})$ , la coordenada  $y$  del punto de intersección con un borde de recorte vertical se puede obtener mediante el cálculo,

$$y = y_0 + m(x - x_0) \quad (6.14)$$

donde el valor de  $x$  se establece en  $xw_{\min}$  o  $xw_{\max}$ , y la pendiente de esta línea se calcula como  $m = (y_{\text{fin}} - y_0)/(x_{\text{fin}} - x_0)$ . De forma similar, si buscamos la intersección con el borde horizontal, la coordenada  $x$  se puede calcular como:

$$x = x_0 + \frac{y - y_0}{m} \quad (6.15)$$

(estableciendo  $y$  en  $yw_{\min}$  o en  $yw_{\max}$ ).

En los siguientes procedimientos se proporciona una implementación del algoritmo de recorte de líneas de Cohen-Sutherland bidimensional. La ampliación de este algoritmo a tres dimensiones es directa. Estudiaremos los métodos de visualización tridimensional en el capítulo siguiente.

```
class wcPt2D {
public:
    GLfloat x, y;
};

inline GLint round (const GLfloat a) { return GLint (a + 0.5); }

/* Define un código de cuatro bits para cada una de las regiones
 * exteriores de una ventana rectangular de recorte.
 */

const GLint winLeftBitCode = 0x1;
const GLint winRightBitCode = 0x2;
const GLint winBottomBitCode = 0x4;
const GLint winTopBitCode = 0x8;

/* Un código de región de máscara de bit se asigna también a cada extremo
 * de un segmento de línea de entrada, de acuerdo con su posición respecto de
 * los cuatro bordes de una ventana de recorte rectangular de entrada.
 *
 * Un extremo con un valor de código de región de 0000 se encuentra dentro
 * de la ventana de recorte, en otro caso, está fuera al menos respecto
```

```

* de uno de los límites de recorte. Si la operación 'or' entre los dos
* códigos de los puntos extremos da como resultado un valor falso, la línea
* completa definida por dichos dos puntos se guarda (se acepta).
* Si la operación 'and' entre los códigos de los puntos extremos da como
* resultado verdadero, quiere decir que la línea está completamente fuera de
* la ventana de recorte y se elimina (se rechaza) para posteriores
* procesamientos.
*/

```

```

inline GLint inside (GLint code) { return GLint (!code); }
inline GLint reject (GLint code1, GLint code2)
    { return GLint (code1 & code2); }
inline GLint accept (GLint code1, GLint code2)
    { return GLint (!(code1 | code2)); }

GLubyte encode (wcPt2D pt, wcPt2D winMin, wcPt2D winMax)

{
    GLubyte code = 0x00;

    if (pt.x < winMin.x)
        code = code | winLeftBitCode;
    if (pt.x > winMax.x)
        code = code | winRightBitCode;
    if (pt.y < winMin.y)
        code = code | winBottomBitCode;
    if (pt.y > winMax.y)
        code = code | winTopBitCode;
    return (code);
}

void swapPts (wcPt2D * p1, wcPt2D * p2)
{
    wcPt2D tmp;
    tmp = *p1; *p1 = *p2; *p2 = tmp;
}

void swapCodes (GLubyte * c1, GLubyte * c2)
{
    GLubyte tmp;
    tmp = *c1; *c1 = *c2; *c2 = tmp;
}

void lineClipCohSuth (wcPt2D winMin, wcPt2D winMax, wcPt2D p1, wcPt2D p2)
{
    GLubyte code1, code2;
    GLint done = false, plotLine = false;
    GLfloat m;
}

```

```

while (!done) {
    code1 = encode (p1, winMin, winMax);
    code2 = encode (p2, winMin, winMax);
    if (accept (code1, code2)) {
        done = true;
        plotLine = true;
    }
    else
        if (reject (code1, code2))
            done = true;
        else {
            /* Etiqueta el punto extremo que está fuera de la ventana
             * de visualización como p1. */
            if (inside (code1)) {
                swapPts (&p1, &p2);
                swapCodes (&code1, &code2);
            }
            /* Usa la pendiente m para hallar la intersección línea-límite
             * de recorte.*/
            if (p2.x != p1.x)
                m = (p2.y - p1.y) / (p2.x - p1.x);
            if (code1 & winLeftBitCode) {
                p1.y += (winMin.x - p1.x) * m;
                p1.x = winMin.x;
            }
            else
                if (code1 & winRightBitCode) {
                    p1.y += (winMax.x - p1.x) * m;
                    p1.x = winMax.x;
                }
            else
                if (code1 & winBottomBitCode) {
                    /* Es necesario actualizar p1.x sólo para líneas no verticales. */
                    if (p2.x != p1.x)
                        p1.x += (winMin.y - p1.y) / m;
                    p1.y = winMin.y;
                }
            else
                if (code1 & winTopBitCode) {
                    if (p2.x != p1.x)
                        p1.x += (winMax.y - p1.y) / m;
                    p1.y = winMax.y;
                }
            }
        }
    if (plotLine)
        lineBres (round (p1.x), round (p1.y), round (p2.x), round (p2.y));
}

```

## Recorte de líneas de Liang-Barsky

Se han desarrollado algoritmos de recorte de líneas más rápidos que realizan una comprobación mayor de las líneas antes de proceder con los cálculos de intersección. Uno de los más antiguos esfuerzos en esta dirección es un algoritmo desarrollado por Cyrus y Beck, que se basa en el análisis de las ecuaciones paramétricas de las líneas. Posteriormente, Liang y Barsky de forma independiente idearon una forma incluso más rápida del algoritmo de recorte de líneas paramétrico.

Para un segmento de línea cuyos puntos extremos son  $(x_0, y_0)$  y  $(x_{\text{fin}}, y_{\text{fin}})$ , podemos describir la línea en forma paramétrica:

$$\begin{aligned} x &= x_0 + u\Delta x \\ y &= y_0 + u\Delta y \end{aligned} \quad 0 \leq u \leq 1 \quad (6.16)$$

donde  $\Delta x = x_{\text{fin}} - x_0$  y  $\Delta y = y_{\text{fin}} - y_0$ . En el algoritmo de Liang-Barsky, las ecuaciones paramétricas de la línea se combinan con las condiciones de recorte de puntos 6.12 para obtener las desigualdades:

$$\begin{aligned} xw_{\min} &\leq x_0 + u\Delta x \leq xw_{\max} \\ yw_{\min} &\leq y_0 + u\Delta y \leq yw_{\max} \end{aligned} \quad (6.17)$$

que se pueden expresar como:

$$u p_k \leq q_k, \quad k = 1, 2, 3, 4 \quad (6.18)$$

donde los parámetros  $p$  y  $q$  se definen como:

$$\begin{aligned} p_1 &= -\Delta x, & q_1 &= x_0 - xw_{\min} \\ p_2 &= \Delta x, & q_2 &= xw_{\max} - x_0 \\ p_3 &= -\Delta y, & q_3 &= y_0 - yw_{\min} \\ p_4 &= \Delta y, & q_4 &= yw_{\max} - y_0 \end{aligned} \quad (6.19)$$

Cualquier línea que sea paralela a una de las aristas de la ventana de recorte tiene  $p_k = 0$  para el valor de  $k$  correspondiente a dicho límite, donde  $k = 1, 2, 3$  y  $4$  se corresponde con los límites izquierdo, derecho, inferior y superior, respectivamente. Si, para ese valor de  $k$ , también observamos que  $q_k < 0$ , entonces la línea está completamente fuera del límite y se puede eliminar de las consideraciones posteriores. Si  $q_k \geq 0$ , la línea se encuentra dentro del límite de recorte paralelo.

Cuando  $p_k < 0$ , el alargamiento infinito de la línea está orientado desde fuera hacia dentro del alargamiento infinito de ese borde particular de la ventana de recorte. Si  $p_k > 0$ , la línea procede de dentro hacia fuera. Para un valor distinto de cero de  $p_k$ , podemos calcular el valor de  $u$  que se corresponde con el punto donde la línea alargada hasta el infinito intersecta con la ampliación del borde  $k$  de la ventana del siguiente modo:

$$u = \frac{q_k}{p_k} \quad (6.20)$$

Para cada línea, podemos calcular los valores de los parámetros  $u_1$  y  $u_2$  que definen aquella parte de la línea que se encuentra dentro del rectángulo de recorte. El valor de  $u_1$  se determina mediante la búsqueda en las aristas del rectángulo para las que la línea está orientada de afuera hacia dentro ( $p < 0$ ). Para estas aristas, calculamos  $r_k = q_k/p_k$ . El valor de  $u_1$  se obtiene como el mayor del conjunto que contiene 0 y los valores de  $r$ . Inversamente, el valor de  $u_2$  se determina examinando los límites para los que la línea está orientada de dentro hacia afuera ( $p > 0$ ). Para cada una de estas aristas se calcula un valor de  $r_k$  y el valor de  $u_2$  es el mínimo del conjunto que contiene 1 y los valores de  $r$  calculados. Si  $u_1 > u_2$ , la línea está completamente fuera de la ventana de recorte y se puede rechazar. De lo contrario, los puntos extremos de la línea recortada se calculan a partir de los dos valores del parámetro  $u$ .

Este algoritmo está implementado en las siguientes secciones de código. Los parámetros de las intersecciones se inicializan con los valores  $u_1 = 0$  y  $u_2 = 1$ . Para cada límite de recorte, se calculan los valores apropiados de  $p$  y  $q$  y se utilizan en la función clipTest para determinar si la línea se puede rechazar o si se han de ajustar los parámetros de intersección. Cuando  $p < 0$ , el parámetro  $r$  se utiliza para actualizar  $u_1$ ; cuando  $p > 0$ , el parámetro  $r$  se utiliza para actualizar  $u_2$ . Si al actualizar  $u_1$  o  $u_2$  resulta que  $u_1 > u_2$ , rechazamos la línea. De lo contrario, actualizamos el parámetro  $u$  apropiado sólo si el nuevo valor acorta la línea. Cuando  $p = 0$  y  $q < 0$ , podemos eliminar la línea ya que es paralela a ese límite de recorte y se encuentra situada fuera del mismo. Si la línea no se rechaza después de comprobar los cuatro valores de  $p$  y  $q$ , se determinan los puntos extremos de la línea recortada a partir de los valores de  $u_1$  y  $u_2$ .

```

class wcPt2D
{
private:
    GLfloat x, y;

public:
    /* Constructor predeterminado: inicializa la posición como (0.0, 0.0). */
    wcPt3D ( ) {
        x = y = 0.0;
    }

    setCoords (GLfloat xCoord, GLfloat yCoord) {
        x = xCoord;
        y = yCoord;
    }

    GLfloat getx ( ) const {
        return x;
    }

    GLfloat gety ( ) const {
        return y;
    }
};

inline GLint round (const GLfloat a) { return GLint (a + 0.5); }

GLint clipTest (GLfloat p, GLfloat q, GLfloat * u1, GLfloat * u2)
{
    GLfloat r;
    GLint returnValue = true;

    if (p < 0.0) {
        r = q / p;
        if (r > *u2)
            returnValue = false;
    }
    else
        if (r > *u1)
            *u1 = r;
}

```

```

else
    if (p > 0.0) {
        r = q / p;
        if (r < *u1)
            returnValue = false;

    else if (r < *u2)
        *u2 = r;
    }

else
    /* Luego p = 0 y la línea es paralela al límite de recorte. */
    if (q < 0.0)
        /* La línea está fuera del límite de recorte. */
        returnValue = false;
    return (returnValue);
}

void lineClipLiangBarsk (wcPt2D winMin, wcPt2D winMax, wcPt2D p1, wcPt2D p2)
{
    GLfloat u1 = 0.0, u2 = 1.0, dx = p2.getx () - p1.getx (), dy;

    if (clipTest (-dx, p1.getx () - winMin.getx (), &u1, &u2))
        if (clipTest (dx, winMax.getx () - p1.getx (), &u1, &u2)) {
            dy = p2.gety () - p1.gety ();
            if (clipTest (-dy, p1.gety () - winMin.gety (), &u1, &u2))
                if (clipTest (dy, winMax.gety () - p1.gety (), &u1, &u2)) {
                    if (u2 < 1.0) {
                        p2.setCoords (p1.getx () + u2 * dx, p1.gety ()
                                      + u2 * dy);
                    }
                    if (u1 > 0.0) {
                        p1.setCoords (p1.getx () + u1 * dx, p1.gety ()
                                      + u1 * dy);
                    }
                    lineBres (round (p1.getx ()), round (p1.gety ()),
                               round (p2.getx ()), round (p2.gety ()));
                }
        }
}
}

```

Por lo general, el algoritmo de recorte de líneas de Liang-Barsky es más eficiente que el de Cohen-Sutherland. Cada actualización de los parámetros  $u_1$  y  $u_2$  requiere sólo una división; y las intersecciones de la línea con la ventana se calculan sólo una vez, cuando se han calculado los valores finales de  $u_1$  y  $u_2$ . Sin embargo, el algoritmo de Cohen y Sutherland puede calcular repetidamente las intersecciones a lo largo de la trayectoria de la línea, aun cuando la línea se encuentre totalmente fuera de la ventana de recorte. Y cada cálculo de intersección de Cohen-Sutherland requiere tanto una división como una multiplicación. El algoritmo bidimensional de Liang-Barsky se puede ampliar para recortar líneas tridimensionales (véase el Capítulo 7).

## Recorte de líneas de Nicholl-Lee-Nicholl

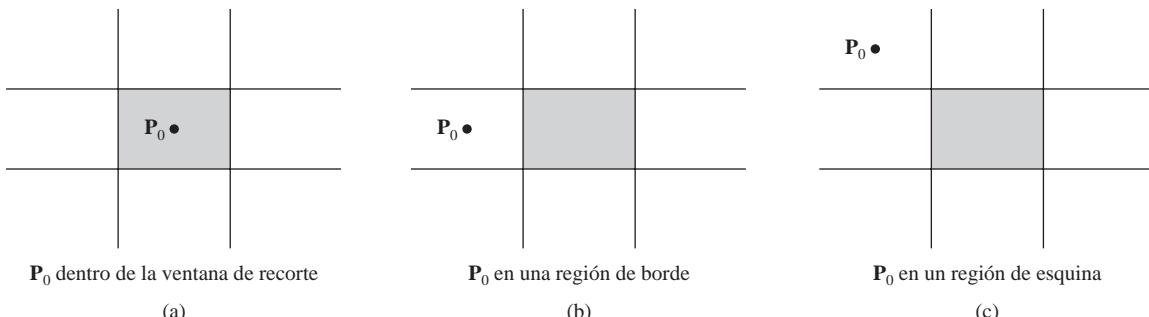
Mediante la creación de más regiones alrededor de la ventana de recorte, el algoritmo de Nicholl-Lee-Nicholl (NLN) evita los múltiples cálculos de las intersecciones de la línea. En el método de Cohen-Sutherland, por ejemplo, se podían calcular múltiples intersecciones a lo largo de la trayectoria de un segmento de línea antes de localizar una intersección en el rectángulo de recorte o de rechazar completamente la línea. Estos cálculos adicionales de intersecciones se eliminan en el algoritmo NLN mediante una mayor comprobación de regiones antes de calcular las intersecciones. Comparado tanto con el algoritmo de Cohen-Sutherland como con el de Liang-Barsky, el algoritmo de Nicholl-Lee-Nicholl realiza menos comparaciones y divisiones. La desventaja del algoritmo NLN es que sólo se puede aplicar al recorte bidimensional, mientras que tanto el algoritmo de Liang-Barsky como el de Cohen-Sutherland se pueden ampliar fácilmente a escenas tridimensionales.

La comprobación inicial para determinar si un segmento de línea está completamente dentro de la ventana de recorte o fuera se puede realizar con comprobaciones de códigos de región, como en los dos algoritmos previos. Si no es posible una aceptación o un rechazo triviales, el algoritmo NLN procede a establecer regiones de recorte adicionales.

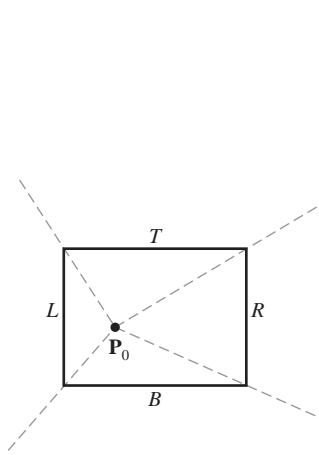
Para una línea con puntos extremos  $P_0$  y  $P_{fin}$ , primero determinamos la posición del punto  $P_0$  respecto a las nueve posibles regiones relativas a la ventana de recorte. Sólo se necesita considerar las tres regiones que se muestran en la Figura 6.16. Si  $P_0$  se encuentra en cualquiera de las otras seis regiones, podemos moverlo a una de las tres regiones de la Figura 6.16 mediante una transformación de simetría. Por ejemplo, se puede transformar la región situada directamente encima de la ventana de recorte en la región situada a la izquierda de la ventana utilizando una reflexión respecto de la línea  $y = -x$ , o podríamos utilizar una rotación de 90° en sentido contrario al movimiento de las agujas del reloj.

Asumiendo que  $P_0$  y  $P_{fin}$  no están ambos dentro de la ventana de recorte, a continuación determinamos la posición de  $P_{fin}$  respecto de  $P_0$ . Para ello, creamos algunas regiones nuevas en el plano, dependiendo de la posición de  $P_0$ . Los límites de las nuevas regiones son los segmentos de línea semi-infinitos que comienzan en  $P_0$  y pasan a través de las esquinas de la ventana de recorte. Si  $P_0$  se encuentra dentro de la ventana de recorte, establecemos las cuatro regiones que se muestran en la Figura 6.17. Después, dependiendo de cuál de estas cuatro regiones (L, T, R, o B) contenga  $P_{fin}$ , calculamos la intersección de la línea con el límite correspondiente de la ventana.

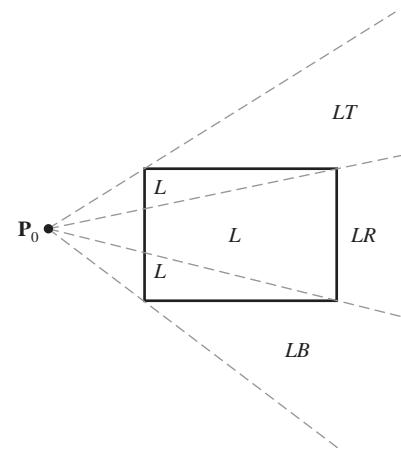
Si  $P_0$  se encuentra en la región situada a la izquierda de la ventana, establecemos las cuatro regiones etiquetadas como L, LT, LR y LB en la Figura 6.18. Estas cuatro regiones de nuevo determinan una única arista de la ventana de recorte para el segmento de línea, relativa a la posición de  $P_{fin}$ . Por ejemplo, si  $P_{fin}$  está en una de las tres regiones etiquetadas como L, recortamos la línea del límite izquierdo de la ventana y guardamos el segmento de línea desde esta intersección a  $P_{fin}$ . Si  $P_{fin}$  está en la región LT, salvamos el segmento desde el límite izquierdo de la ventana hasta el límite superior. Un procesamiento similar se lleva a cabo en las regiones LR y LB. Pero si  $P_{fin}$  no está en ninguna de las cuatro regiones L, LT, LR o LB, se recorta la línea completa.



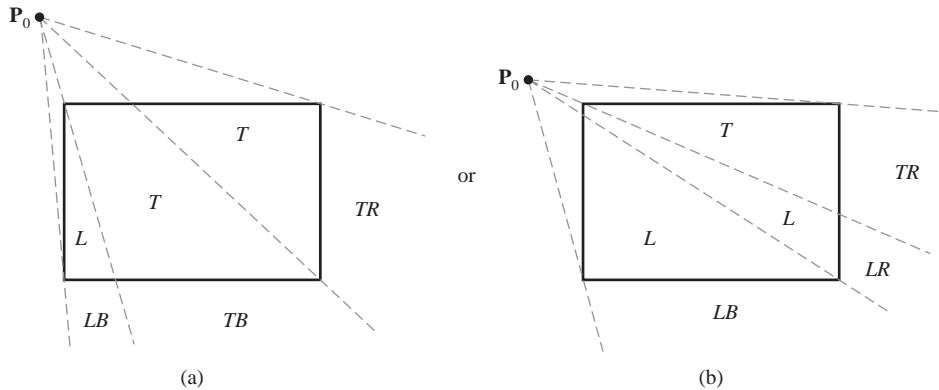
**FIGURA 6.16.** Tres posibles posiciones de un punto extremo de una línea  $P_0$  en el algoritmo de recorte de líneas NLN.



**FIGURA 6.17.** Las cuatro regiones utilizadas en el algoritmo NLN cuando  $P_0$  está dentro de la ventana de recorte y  $P_{\text{fin}}$  está fuera.



**FIGURA 6.18.** Las cuatro regiones de recorte utilizadas en el algoritmo NLN cuando  $P_0$  está directamente a la izquierda de la ventana de recorte.



**FIGURA 6.19.** Los dos posibles conjuntos de regiones de recorte utilizados en el algoritmo NLN cuando  $P_0$  está encima y a la izquierda de la ventana de recorte.

En el tercer caso, cuando  $P_0$  está a la izquierda y encima de la ventana de recorte, utilizamos las regiones de la Figura 6.19. En este caso, tenemos las dos posibilidades mostradas, dependiendo de la posición de  $P_0$  con respecto a la esquina superior izquierda de la ventana de recorte. Cuando  $P_0$  está cerca del límite izquierdo de recorte de la ventana, utilizamos las regiones del apartado (a) de esta figura. De lo contrario, cuando  $P_0$  está cerca del límite superior de recorte de la ventana, utilizamos las regiones del apartado (b). Si  $P_{\text{fin}}$  está en una de las regiones T, L, TR, TB, LR, o LB, esto determina un único borde de la ventana de recorte para los cálculos de intersección. De lo contrario, la línea entera se rechaza.

Para determinar la región en la que se encuentra  $P_{\text{fin}}$ , comparamos la pendiente del segmento de línea con las pendientes de los límites de las regiones del algoritmo NLN. Por ejemplo, si  $P_0$  está a la izquierda de la ventana de recorte (Figura 6.18), entonces  $P_{\text{fin}}$  está en la región LT si,

$$\text{pendiente } \overline{P_0 P_{TR}} < \text{pendiente } \overline{P_0 P_{\text{fin}}} < \text{pendiente } \overline{P_0 P_{TL}} \quad (6.21)$$

$$\frac{y_T - y_0}{x_R - x_0} < \frac{y_{\text{fin}} - y_0}{x_{\text{fin}} - x_0} < \frac{y_L - y_0}{x_L - x_0} \quad (6.22)$$

Y recortamos la línea entera si,

$$(y_T - y_0)(x_{\text{fin}} - x_0) < (x_L - x_0)(y_{\text{fin}} - y_0) \quad (6.23)$$

Los cálculos de diferencias de coordenadas y los cálculos de productos utilizados en los tests de las pendientes se almacenan y se utilizan también en los cálculos de las intersecciones. A partir de las ecuaciones paramétricas:

$$x = x_0 + (x_{\text{fin}} - x_0)u$$

$$y = y_0 + (y_{\text{fin}} - y_0)u$$

calculamos la coordenada  $x$  de la intersección con el límite izquierdo de la ventana como  $x = x_L$ , por lo que  $u = (x_L - x_0)/(x_{\text{end}} - x_0)$  y la coordenada  $y$  de la intersección es:

$$y = y_0 + \frac{y_{\text{fin}} - y_0}{x_{\text{fin}} - x_0}(x_L - x_0) \quad (6.24)$$

Y para una intersección con el límite superior tenemos  $y = y_T$  y  $u = (y_T - y_0)/(y_{\text{fin}} - y_0)$ , por lo que la coordenada  $x$  vale:

$$x = x_0 + \frac{x_{\text{fin}} - x_0}{y_{\text{fin}} - y_0}(y_T - y_0) \quad (6.25)$$

## Recorte de líneas con ventanas de recorte poligonales no rectangulares

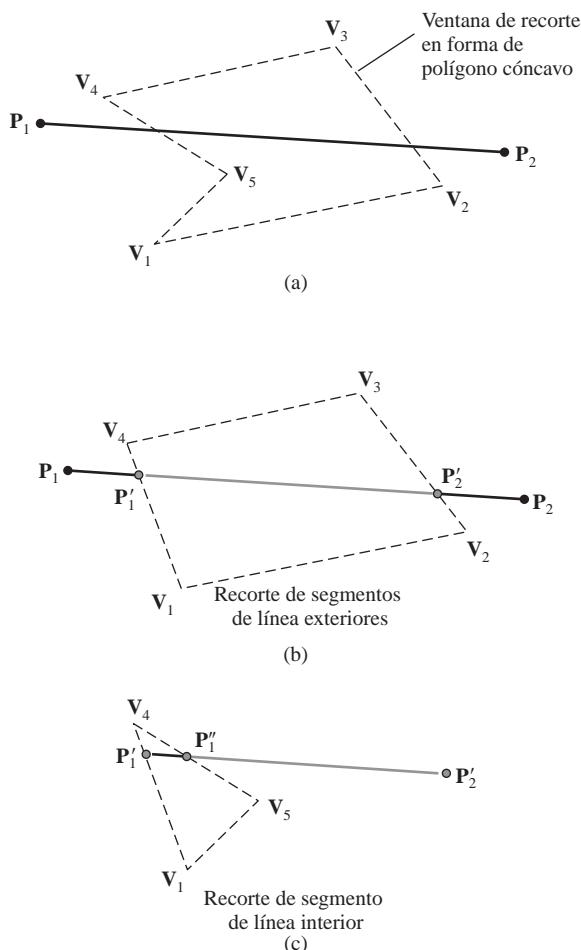
En algunas aplicaciones, se puede desear recortar líneas con polígonos de forma arbitraria. Los métodos basados en las ecuaciones paramétricas de la línea, tales como el algoritmo de Cyrus-Beck o el de Liang-Barsky, se pueden ampliar fácilmente para recortar líneas con ventanas poligonales convexas. Hacemos esto modificando el algoritmo para que incluya las ecuaciones paramétricas de los límites de la región de recorte. La visualización preliminar de los segmentos de línea se puede realizar procesando las líneas frente a las extensiones de coordenadas del polígono de recorte.

En el caso de las regiones de recorte poligonales cóncavas, todavía podríamos aplicar estos procedimientos de recorte paramétricos si primero dividimos el polígono cóncavo en un conjunto de polígonos convexos utilizando uno de los métodos descritos en la Sección 3.15. Otra técnica consiste simplemente en añadir una o más aristas adicionales al área de recorte cóncava para que se transforme en un polígono convexo. Entonces se puede aplicar una serie de operaciones de recorte utilizando las componentes del polígono convexo modificado, como se muestra en la Figura 6.20.

El segmento de línea  $\overline{\mathbf{P}_1 \mathbf{P}_2}$  del apartado (a) de esta figura hay que recortarlo con la ventana cóncava de vértices  $\mathbf{V}_1, \mathbf{V}_2, \mathbf{V}_3, \mathbf{V}_4$  y  $\mathbf{V}_5$ . En este caso, se obtienen dos regiones de recorte convexas, añadiendo un segmento de línea de  $\mathbf{V}_4$  a  $\mathbf{V}_1$ . Después se recorta la línea en dos pasos: (1) se recorta la línea  $\overline{\mathbf{P}_1 \mathbf{P}_2}$  con el polígono convexo de vértices  $\mathbf{V}_1, \mathbf{V}_2, \mathbf{V}_3$  y  $\mathbf{V}_4$  para producir el segmento recortado  $\overline{\mathbf{P}'_1 \mathbf{P}'_2}$  (véase la Figura 6.20(b)). (2) Se recorta el segmento de línea interno  $\overline{\mathbf{P}'_1 \mathbf{P}'_2}$  utilizando el polígono convexo de vértices  $\mathbf{V}_1, \mathbf{V}_5$  y  $\mathbf{V}_4$  (Figura 6.20(c)) para producir el segmento de línea recortado final  $\overline{\mathbf{P}''_1 \mathbf{P}''_2}$ .

## Recorte de líneas utilizando ventanas de recorte con límites no lineales

Los círculos u otras regiones de recorte con límites curvados también son posibles, pero requieren más procesamiento, ya que los cálculos de intersección implican ecuaciones no lineales. En el primer paso, las líneas



**FIGURA 6.20.** Una ventana de recorte con forma de polígono cóncavo (a) definida por los vértices  $V_1, V_2, V_3, V_4$  y  $V_5$  se modifica para obtener el polígono convexo (b) definido por los vértices  $V_1, V_2, V_3$  y  $V_4$ . Los segmentos externos de la línea  $\overline{P_1P_2}$  se descartan usando esta ventana de recorte convexa. El segmento de línea resultante,  $\overline{P'_1P'_2}$  se procesa después de acuerdo con el triángulo ( $V_1, V_5, V_4$ ) (c) para recortar el segmento de línea interno  $\overline{P'_1P''_1}$  y generar la línea recortada final  $\overline{P''_1P'_2}$ .

se recortarían con el rectángulo limitador (extensiones de coordenadas) de la región de recorte curvada. Se eliminan las líneas que se encuentran fuera de las extensiones de las coordenadas. Para identificar las líneas que están dentro de un círculo, por ejemplo, podríamos calcular la distancia de los puntos extremos de la línea al centro del círculo. Si el cuadrado de esta distancia para ambos puntos extremos de una línea es menor o igual que el radio al cuadrado, podemos guardar la línea completa. Las líneas restantes se procesan después mediante cálculos de intersección, que deben ser la solución simultánea de las ecuaciones de la línea y del círculo.

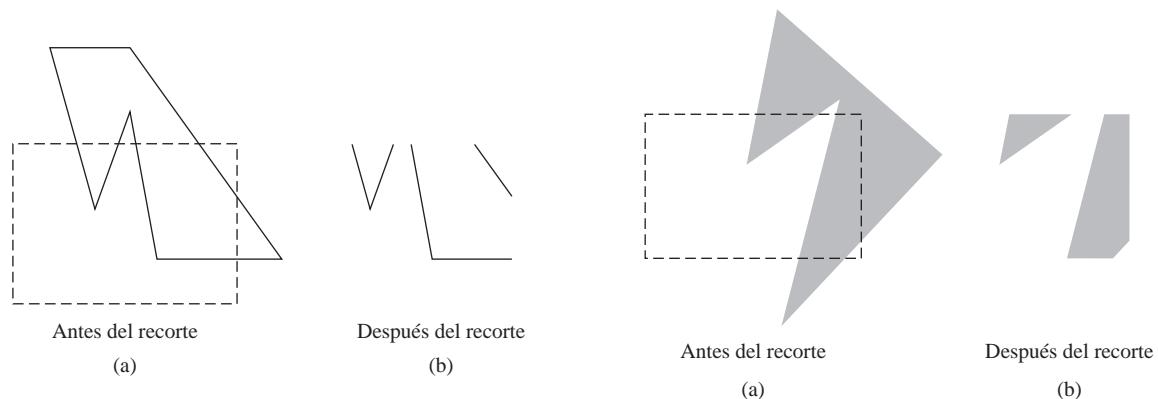
## 6.8 RECORTE DE ÁREAS DE RELLENO POLIGONALES

Los paquetes gráficos habitualmente sólo permiten áreas de relleno que sean polígonos y, a menudo, sólo polígonos convexos. Para recortar un área de relleno poligonal, no podemos aplicar directamente un método de

recorte de líneas a las aristas individuales del polígono porque esta técnica no produciría, por lo general, una polilínea cerrada. En lugar de eso, un recortador de líneas produciría a menudo un conjunto disjunto de líneas con información incompleta, acerca de cómo podríamos formar un límite cerrado alrededor del área de relleno recortada. La Figura 6.21 muestra una posible salida de un procedimiento de recorte de líneas aplicado a las aristas de un área de relleno poligonal. Lo que necesitamos es un procedimiento que produzca una o más polilíneas cerradas como límites del área de relleno recortada, con el fin de que los polígonos se puedan convertir por exploración para llenar los interiores con el color o patrón asignado, como en la Figura 6.22.

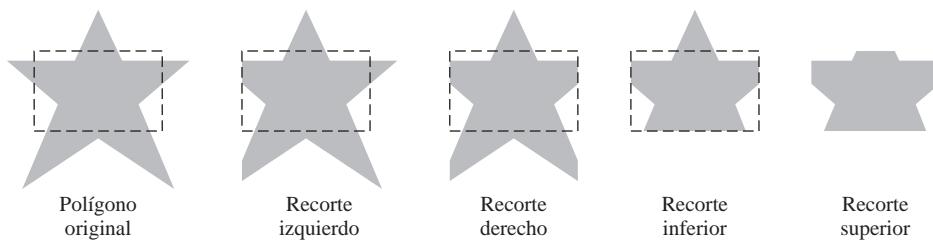
Podemos procesar un área de relleno poligonal de acuerdo con los bordes de una ventana de recorte, utilizando la misma técnica general del recorte de líneas. Un segmento de línea se define mediante sus dos puntos extremos, y estos puntos extremos se procesan mediante un procedimiento de recorte de líneas, construyendo un nuevo conjunto de puntos extremos recortados en cada borde de la ventana de recorte. De forma similar, necesitamos mantener un área de relleno como una entidad a medida que ella se procesa a través de las etapas de recorte. Por tanto, podemos recortar un área de relleno poligonal determinando la nueva forma del polígono a medida que se procesa cada arista de la ventana de recorte, como se muestra en la Figura 6.23. Por supuesto, el relleno interior del polígono no se aplicará hasta que se haya determinado el borde de recorte final.

Al igual que comprobamos un segmento de línea para determinar si se puede guardar completamente o recortar completamente, podemos hacer lo mismo con un área de relleno poligonal comprobando las extensiones de sus coordenadas. Si los valores máximo y mínimo de las coordenadas del área de relleno están dentro de los cuatro límites de recorte, el área de relleno se almacena para un posterior procesamiento. Si estas extensiones de coordenadas están todas fuera de cualquiera de los límites de la ventana de recorte, eliminamos el polígono de la descripción de la escena (Figura 6.24).

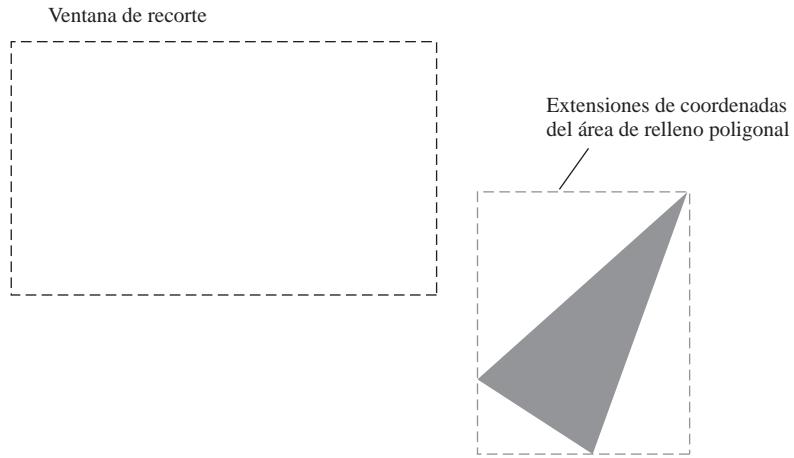


**FIGURA 6.21.** Un algoritmo de recorte de líneas aplicado a los segmentos de línea del límite poligonal de (a) genera el conjunto de líneas sin conexión mostrado en (b).

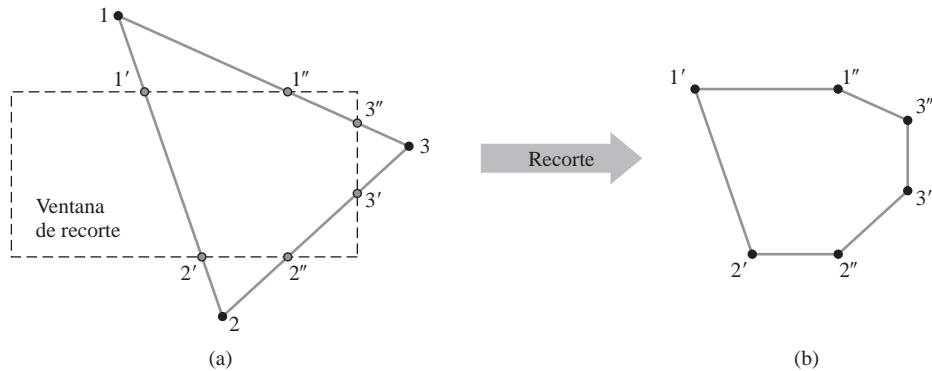
**FIGURA 6.22.** Visualización de un área de relleno poligonal correctamente recortada.



**FIGURA 6.23.** Procesamiento de un área de relleno poligonal con los sucesivos límites de la ventana de recorte.



**FIGURA 6.24.** Un área de relleno poligonal con las extensiones de coordenadas fuera del límite derecho de recorte.



**FIGURA 6.25.** Un área de relleno poligonal convexa (a), definida por la lista de vértices  $\{1, 2, 3\}$ , se recorta para producir la forma del área de relleno mostrada en (b), que se define por la lista de vértices de salida  $\{1', 2', 2'', 3', 3'', 1''\}$ .

Cuando no podemos identificar si un área de relleno está totalmente dentro o totalmente fuera de la ventana de recorte, necesitamos entonces localizar las intersecciones del polígono con los límites de recorte. Una manera de implementar el recorte de polígonos convexas consiste en crear una nueva lista de vértices en cada límite de recorte, y entonces pasar esta nueva lista de vértices al siguiente recortador de límites. La salida de la etapa final de recorte es la lista de vértices del polígono recortado (Figura 6.25). Para el recorte de polígonos cóncavos, es necesario modificar esta técnica básica para que se puedan generar múltiples listas de vértices.

### Recorte de polígonos de Sutherland-Hodgman

Un método eficiente de recorte de áreas de relleno poligonales convexas, desarrollado por Sutherland y Hodgman, consiste en enviar los vértices del polígono a través de cada etapa de recorte para que un único vértice recortado se pueda pasar inmediatamente a la etapa siguiente. Esto elimina la necesidad de producir como salida un conjunto de vértices en cada etapa de recorte, lo que permite que las subrutinas de recorte de bordes se implementen en paralelo. La salida final es una lista de vértices que describe las aristas del área de relleno poligonal recortada.

Ya que el algoritmo de Sutherland-Hodgman produce sólo una lista de vértices de salida, no puede generar correctamente los dos polígonos de salida de la Figura 6.22(b), que son el resultado del recorte del polí-

gono cóncavo mostrado en el apartado (a) de la figura. Sin embargo, se puede añadir un procesamiento adicional al algoritmo de Sutherland-Hodgman para obtener múltiples listas de vértices como salida, con el fin de poder realizar el recorte de polígonos cóncavos generales. El algoritmo básico de Sutherland-Hodgman es capaz de procesar polígonos cóncavos cuando el área de relleno recortada se puede describir con una única lista de vértices.

La estrategia general de este algoritmo es enviar el par de puntos extremos de cada sucesivo segmento de línea del polígono a través de la serie de recortadores (izquierdo, derecho, inferior y superior). Tan pronto como un recortador completa el procesamiento de un par de vértices, los valores de las coordenadas recortadas, si existen, para dicha arista se envían al siguiente recortador. Después, el primer recortador procesa el siguiente par de puntos extremos. De este modo, los recortadores individuales de bordes pueden funcionar en paralelo.

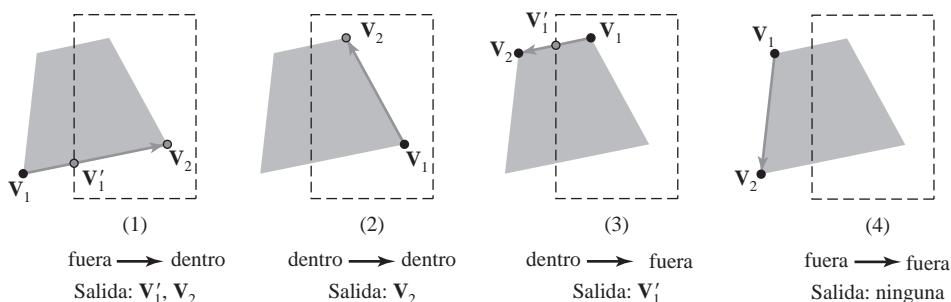
Existen cuatro posibles casos que hay que considerar cuando se procesa una arista de un polígono de acuerdo con uno de los límites de recorte. Una posibilidad es que el primer punto extremo de la arista esté fuera del límite de recorte y el segundo extremo esté dentro. O, ambos extremos podrían estar dentro del límite de recorte. Otra posibilidad es que el primer extremo esté dentro del límite de recorte y el segundo fuera. Y, finalmente, ambos puntos extremos podrían estar fuera del límite de recorte.

Para facilitar el paso de los vértices de una etapa de recorte a la siguiente, la salida de cada recortador se puede formular como se muestra en la Figura 6.26. A medida que cada sucesivo par de puntos extremos se pasa a uno de los cuatro recortadores, se genera una salida para el siguiente recortador de acuerdo con los resultados de las siguientes pruebas.

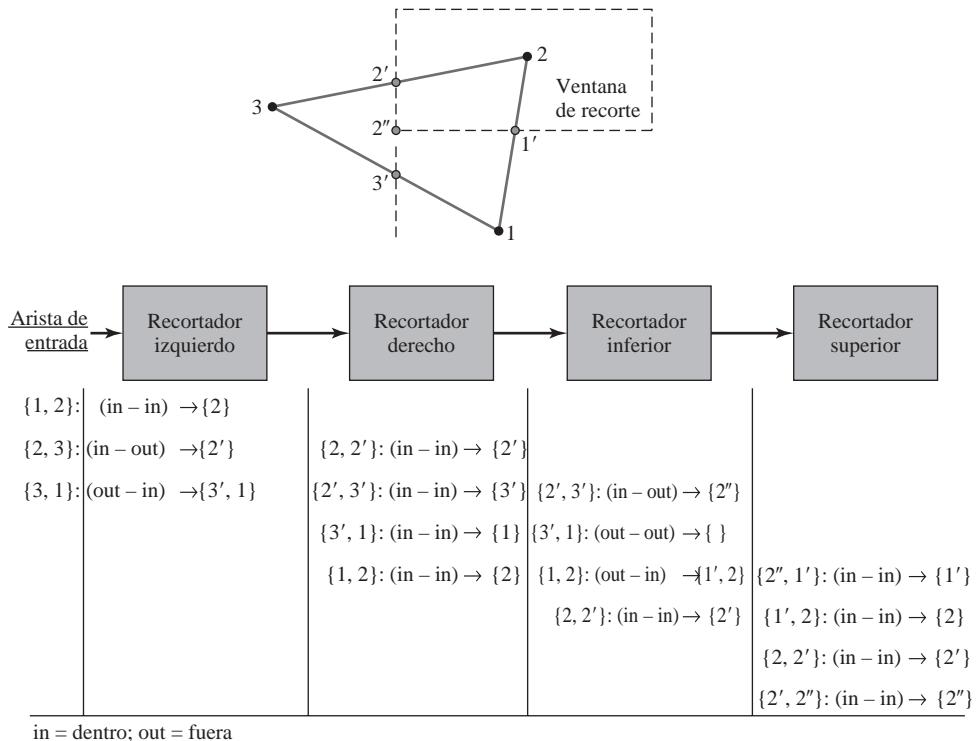
- (1) El primer vértice de entrada está fuera del borde de la ventana de recorte y el segundo está dentro, tanto el punto de intersección de la arista del polígono con el borde de la ventana como el segundo vértice se envían al siguiente recortador.
- (2) Si ambos vértices de entrada se encuentran dentro de este borde de la ventana de recorte, sólo el segundo vértice se envía al siguiente recortador.
- (3) Si el primer vértice está dentro de este borde de la ventana de recorte y el segundo vértice está fuera, sólo la intersección de la arista del polígono con el borde de la ventana de recorte se envía al siguiente recortador.
- (4) Si ambos vértices de entrada se encuentran fuera de este borde de la ventana de recorte, no se envían vértices al siguiente recortador.

El último recortador de esta serie genera una lista de vértices que describe el área final de relleno recortada.

La Figura 6.27 proporciona un ejemplo del algoritmo de recorte de polígonos de Sutherland-Hodgman para un área de relleno definida por el conjunto de vértices  $\{1, 2, 3\}$ . Tan pronto como un recortador recibe un par de puntos extremos, determina la salida apropiada utilizando las pruebas mostradas en la Figura 6.26.



**FIGURA 6.26.** Las cuatro posibles salidas generadas por el recortador izquierdo, dependiendo de la posición de un par de puntos extremos respecto del borde izquierdo de la ventana de recorte.



**FIGURA 6.27.** Procesamiento de un conjunto de vértices de un polígono,  $\{1, 2, 3\}$ , mediante los recortadores de bordes utilizando el algoritmo de Sutherland-Hodgman. El conjunto final de vértices recortados es  $\{1', 2, 2', 2''\}$ .

Estas salidas se pasan sucesivamente desde el recortador izquierdo al derecho, al inferior y al superior. La salida del recortador superior es el conjunto de vértices que define el área de relleno recortada. En este ejemplo, la lista de vértices de salida es  $\{1', 2, 2', 2''\}$ .

En el siguiente conjunto de procedimientos se muestra una implementación secuencial del algoritmo de recorte de polígonos de Sutherland-Hodgman. Un conjunto de vértices de entrada se convierte en una lista de vértices de salida mediante las subrutinas de recorte izquierda, derecha, inferior y superior.

```

typedef enum { Left, Right, Bottom, Top } Boundary;
const GLint nClip = 4;

GLint inside (wcPt2D p, Boundary b, wcPt2D wMin, wcPt2D wMax)
{
    switch (b) {
    case Left: if (p.x < wMin.x) return (false); break;
    case Right: if (p.x > wMax.x) return (false); break;
    case Bottom: if (p.y < wMin.y) return (false); break;
    case Top: if (p.y > wMax.y) return (false); break;
    }
    return (true);
}

GLint cross (wcPt2D p1, wcPt2D p2, Boundary winEdge, wcPt2D wMin, wcPt2D wMax)
{
    if (inside (p1, winEdge, wMin, wMax) == inside (p2, winEdge, wMin, wMax))

```

```

        return (false);
    else return (true);
}

wcPt2D intersect (wcPt2D p1, wcPt2D p2, Boundary winEdge, wcPt2D wMin, wcPt2D wMax)
{
    wcPt2D iPt;
    GLfloat m;

    if (p1.x != p2.x) m = (p1.y - p2.y) / (p1.x - p2.x);
    switch (winEdge) {
        case Left:
            iPt.x = wMin.x;
            iPt.y = p2.y + (wMin.x - p2.x) * m;
            break;
        case Right:
            iPt.x = wMax.x;
            iPt.y = p2.y + (wMax.x - p2.x) * m;
            break;
        case Bottom:
            iPt.y = wMin.y;
            if (p1.x != p2.x) iPt.x = p2.x + (wMin.y - p2.y) / m;
            else iPt.x = p2.x;
            break;
        case Top:
            iPt.y = wMax.y;
            if (p1.x != p2.x) iPt.x = p2.x + (wMax.y - p2.y) / m;
            else iPt.x = p2.x;
            break;
    }
    return (iPt);
}

void clipPoint (wcPt2D p, Boundary winEdge, wcPt2D wMin, wcPt2D wMax,
                wcPt2D * pOut, int * cnt, wcPt2D * first[], wcPt2D * s)
{
    wcPt2D iPt;

    /* Si no existe ningún punto anterior para este límite de recorte,
     * guardar este punto. */
    if (!first[winEdge])
        first[winEdge] = &p;
    else
        /* Existe un punto previo. Si p y un punto anterior cruzan este
         * límite de recorte, hallar la intersección. Recortar de acuerdo
         * con el siguiente límite, si lo hay. Si no hay más límites de recorte,
         * añadir la intersección a la lista de salida. */
        if (cross (p, s[winEdge], winEdge, wMin, wMax)) {
            iPt = intersect (p, s[winEdge], winEdge, wMin, wMax);
            if (winEdge < Top)

```

```

        clipPoint (iPt, b+1, wMin, wMax, pOut, cnt, first, s);
    else {
        pOut[*cnt] = iPt; (*cnt)++;
    }
}

/* Guardar p como el punto más reciente para este límite de recorte. */
s[winEdge] = p;

/* Para todos, si el punto está dentro, pasar al siguiente límite
 * de recorte, si lo hay. */
if (inside (p, winEdge, wMin, wMax))
    if (winEdge < Top)
        clipPoint (p, winEdge + 1, wMin, wMax, pOut, cnt, first, s);
    else {
        pOut[*cnt] = p; (*cnt)++;
    }
}

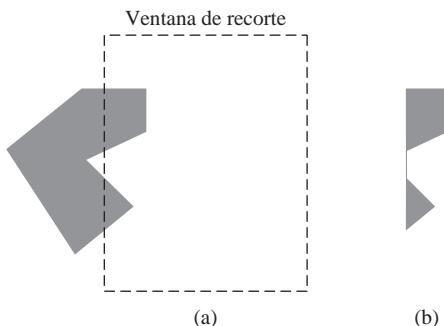
void closeClip (wcPt2D wMin, wcPt2D wMax, wcPt2D * pOut,
                GLint * cnt, wcPt2D * first [ ], wcPt2D * s)
{
    wcPt2D pt;
    Boundary winEdge;

    for (winEdge = Left; winEdge <= Top; winEdge++) {
        if (cross (s[winEdge], *first[winEdge], winEdge, wMin, wMax)) {
            pt = intersect (s[winEdge], *first[winEdge], winEdge, wMin, wMax);
            if (winEdge < Top)
                clipPoint (pt, winEdge + 1, wMin, wMax, pOut, cnt, first, s);
            else {
                pOut[*cnt] = pt; (*cnt)++;
            }
        }
    }
}

GLint polygonClipSuthHodg (wcPt2D wMin, wcPt2D wMax, GLint n, wcPt2D * pIn,
                           wcPt2D * pOut)
{
    /* El parámetro "first" guarda un puntero a primer punto porcesado por un
     * límite de recorte; "s" almacena el punto más recientemente procesado
     * por el límite de recorte.*/
    wcPt2D * first[nClip] = { 0, 0, 0, 0 }, s[nClip];
    GLint k, cnt = 0;

    for (k = 0; k < n; k++)
        clipPoint (pIn[k], Left, wMin, wMax, pOut, &cnt, first, s);
    closeClip (wMin, wMax, pOut, &cnt, first, s);
    return (cnt);
}

```



**FIGURA 6.28.** El recorte del polígono cóncavo de (a) utilizando el algoritmo de Sutherland-Hodgman produce las dos áreas conectadas de (b).

Cuando se recorta un polígono cóncavo con el algoritmo de Sutherland-Hodgman, se pueden visualizar líneas extrañas. En la Figura 6.28 se muestra un ejemplo de este efecto. Esto ocurre cuando el polígono recortado tiene dos o más partes independientes. Pero ya que sólo hay una lista de vértices de salida, el último vértice de la lista se une siempre al primer vértice.

Hay varias cosas que podemos hacer para visualizar correctamente polígonos cóncavos recortados. Podemos dividir un polígono cóncavo en dos o más polígonos convexos (Sección 3.15) y procesar cada polígono convexo de forma independiente utilizando el algoritmo de Sutherland-Hodgman. Otra posibilidad consiste en modificar el método de Sutherland-Hodgman para que se compruebe si en la lista de vértices final existen puntos de intersección múltiples, a lo largo de cualquier límite de la ventana de recorte. Si encontramos más de dos vértices a lo largo de cualquier límite de recorte, podemos dividir la lista de vértices en dos o más listas, que identifiquen correctamente las partes independientes del área de relleno recortada. Esto puede requerir un análisis amplio para determinar si algunos puntos a lo largo del límite de recorte se deben emparejar o si representan un único vértice que se ha recortado. Una tercera posibilidad es utilizar un recortador de polígonos más general que se haya diseñado para procesar polígonos cóncavos correctamente.

## Recorte de polígonos de Weiler-Atherton

Este algoritmo proporciona una técnica general de recorte de polígonos que se puede utilizar para recortar un área de relleno que sea un polígono convexo o un polígono cóncavo. Además, este método se desarrolló como medio de identificación de superficies visibles de una escena tridimensional. Por tanto, podríamos también utilizar esta técnica para recortar cualquier área de relleno poligonal según una ventana de recorte con cualquier forma poligonal.

En lugar de simplemente recortar las aristas del área de relleno como en el método de Sutherland-Hodgman, el algoritmo de Weiler-Atherton recorre el perímetro del polígono de relleno en busca de los límites que encierran una región de relleno recortada. De este modo, como se observa en la Figura 6.28(b), se pueden identificar múltiples regiones de relleno y mostrarlas como polígonos independientes y no conectados. Para encontrar las aristas de un área de relleno recortada, seguimos un camino (en sentido antihorario o en sentido horario) alrededor del área de relleno que se desvía a lo largo de un límite de ventana de recorte, cada vez que una arista de un polígono cruza hacia el exterior de dicho límite. La dirección de un desvío en una arista de una ventana de recorte es la misma que la dirección de procesamiento de las aristas del polígono.

Habitualmente, podemos determinar si la dirección de procesamiento es en sentido antihorario o en sentido horario, a partir de la ordenación de la lista de vértices que define un área de un polígono de relleno. En la mayoría de los casos, la lista de vértices se especifica en sentido antihorario como un medio para definir la cara frontal del polígono. Por tanto, el producto vectorial de los vectores de dos aristas sucesivas que forman un ángulo convexo determina la dirección del vector normal, que está en la dirección de la cara posterior hacia la cara frontal del polígono. Si no conocemos la ordenación de los vértices, podríamos calcular el vector normal, o podríamos usar un método cualquiera de los estudiados en la Sección 3.15 para localizar el interior del área de relleno a partir de cualquier posición de referencia. Despues, si procesamos secuencialmente las aris-

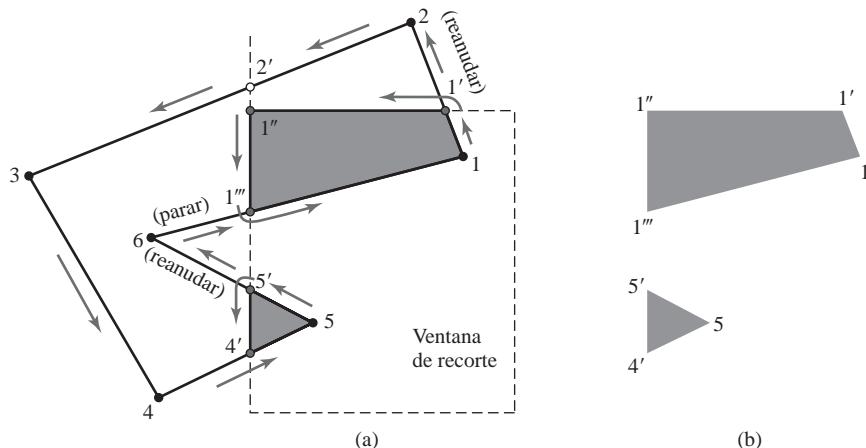
tas para que el interior del polígono esté siempre a nuestra izquierda, obtenemos una transversal en sentido contrario al movimiento de las agujas del reloj. De otra manera, con el interior a nuestra derecha, tenemos una transversal en el sentido del movimiento de las agujas del reloj.

Para una transversal en el sentido contrario al movimiento de las agujas del reloj de los vértices del área de relleno poligonal, aplicamos los siguientes procedimientos de Weiler-Atherton:

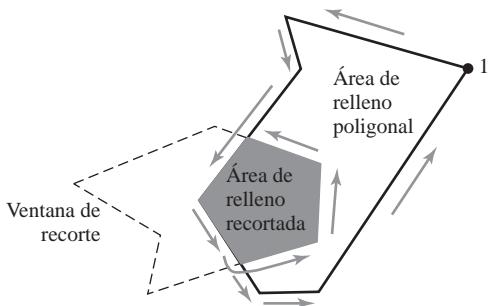
- (1) Procese las aristas del área de relleno poligonal en sentido contrario a las agujas del reloj hasta que encuentre un par de vértices situados dentro y fuera de uno de los límites de recorte; es decir, el primer vértice de la arista del polígono está dentro de la región de recorte y el segundo vértice está fuera de la región de recorte.
- (2) Siga los límites de la ventana en sentido contrario a las agujas del reloj, desde el punto de intersección de salida hacia otro punto de intersección con el polígono. Si éste es un punto previamente procesado, continúe con el paso siguiente. Si éste es un punto de intersección nuevo, continúe procesando las aristas del polígono en sentido contrario al movimiento de las agujas del reloj hasta que se encuentre un vértice procesado previamente.
- (3) Forme la lista de vértices de esta parte del área de relleno recortada.
- (4) Vuelva al punto de intersección de salida y continue procesando las aristas del polígono en orden contrario al movimiento de las agujas del reloj.

La Figura 6.29 muestra el recorte de Weiler-Atherton de un polígono cóncavo con una ventana de recorte rectangular y estándar para una transversal en sentido contrario al movimiento de las agujas del reloj de las aristas del polígono. Para una transversal de las aristas en el sentido del movimiento de las agujas del reloj, usaríamos una transversal de la ventana de recorte en el sentido del movimiento de las agujas del reloj.

Comenzando por el vértice etiquetado como 1 en la Figura 6.26(a), el siguiente vértice del polígono a procesar en sentido antihorario es el etiquetado como 2. Por tanto, esta arista sale de la ventana de recorte por la frontera superior. Calculamos la intersección (punto 1') y hacemos un giro a la izquierda para procesar las aristas de la ventana en dirección contraria al movimiento de las agujas del reloj. Continuando a lo largo del límite superior de la ventana de recorte, no intersectamos con una arista del polígono antes de alcanzar el límite izquierdo de la ventana, por lo que etiquetamos este punto como vértice 1'' y seguimos el límite izquierdo hacia la intersección 1''. Despues seguimos esta arista del polígono en sentido contrario a las agujas del reloj, que nos devuelve al vértice 1. Esto completa el circuito de los límites de la ventana e identifica la lista de



**FIGURA 6.29.** Un polígono cóncavo (a), definido mediante la lista de vértices  $\{1, 2, 3, 4, 5, 6\}$ , se recorta utilizando el algoritmo de Weiler-Atherton para generar las dos listas  $\{1, 1', 1'', 1'''\}$  y  $\{4', 5, 5'\}$ , que representan las áreas de relleno poligonales e independientes mostradas en (b).



**FIGURA 6.30.** Recorte de un área de relleno poligonal con una ventana de recorte poligonal cóncava utilizando el algoritmo de Weiler-Atherton.

vértices  $\{1, 1', 1'', 1'''\}$  como una región recortada del área original de relleno. Después, se continúa el procesamiento de las aristas del polígono en el punto  $1'$ . La arista definida por los puntos  $2$  y  $3$  cruza hacia el exterior del límite izquierdo, pero los puntos  $2$  y  $2'$  están encima de la arista superior de la ventana de recorte y los puntos  $2'$  y  $3$  están a la izquierda de la región de recorte. También la arista de puntos extremos  $3$  y  $4$  se encuentra fuera del límite izquierdo de la ventana de recorte. Pero la siguiente arista (desde el extremo  $4$  hasta el extremo  $5$ ) entra de nuevo en la región de recorte y obtenemos el punto de intersección  $4'$ . La arista de extremos  $5$  y  $6$  sale de la ventana por el punto de intersección  $5'$ , por lo que nos desviamos hacia el límite izquierdo de recorte para obtener la lista de vértices cerrada  $\{4', 5, 5'\}$ . Continuamos el procesamiento de aristas por el punto  $5'$ , que nos devuelve al punto  $1'''$  previamente procesado. En este punto, todos los vértices y las aristas del polígono se han procesado, por lo que el área de relleno está completamente recortada.

### Recorte de polígonos utilizando ventanas de recorte poligonales no rectangulares

El algoritmo de Liang-Barsky y otros métodos de recorte de líneas paramétricos están particularmente bien adecuados para el procesamiento de áreas de relleno de polígonos con ventanas de recorte poligonales y convexas. En esta técnica, utilizamos una representación paramétrica de las aristas tanto del área de relleno como de la ventana de recorte, y ambos polígonos se representan mediante listas de vértices. En primer lugar, comparamos las posiciones de las rectángulos frontera del área de relleno y del polígono de recorte. Si no podemos identificar el área de relleno como completamente fuera del polígono de recorte, podemos utilizar las pruebas dentro-fuera para procesar las ecuaciones paramétricas de las aristas. Después de finalizar todas las pruebas de las regiones, resolvemos sistemas de dos ecuaciones paramétricas de líneas para determinar las intersecciones con la ventana.

También podemos procesar cualquier área de relleno de un polígono con cualquier ventana de recorte de forma poligonal (convexa o cóncava), como en la Figura 6.30, utilizando la técnica arista-transversal del algoritmo de Weiler-Aherton. En este caso, necesitamos mantener una lista de vértices de la ventana de recorte así como del área de relleno, organizadas con un orden en el sentido contrario al movimiento de las agujas del reloj (o a favor). Necesitamos aplicar pruebas dentro-fuera para determinar si un vértice del área de relleno está dentro o fuera de los límites de una ventana de recorte particular. Como en los ejemplos anteriores, seguimos los límites de la ventana cada vez que una arista del área de relleno sale de uno de los límites de recorte. Este método de recorte también se puede utilizar cuando el área de relleno o la ventana de recorte contienen agujeros que quedan definidos mediante límites poligonales. Además, podemos utilizar esta técnica básica en aplicaciones de construcción de geometría de sólidos para identificar la unión, la intersección o la diferencia de dos polígonos. De hecho, la localización de la región recortada de un área de relleno es equivalente a la determinación de la intersección de dos áreas planas.

### Recorte de polígonos utilizando ventanas de recorte con límites no lineales

Un método para procesar una ventana de recorte con límites curvados consiste en aproximar los límites con partes de líneas rectas y utilizar uno de los algoritmos de recorte con una ventana de recorte de forma poligo-

nal. Alternativamente, podríamos utilizar los mismos procedimientos generales que estudiamos para los segmentos de línea. En primer lugar, podemos comparar las extensiones de coordenadas del área de relleno con las extensiones de coordenadas de la ventana de recorte. Dependiendo de la forma de la ventana de recorte, podemos también ser capaces de realizar algunas otras pruebas de región basadas en consideraciones de simetría. En el caso de las áreas de relleno que no se puedan identificar como completamente dentro o completamente fuera de la ventana de recorte, finalmente necesitamos calcular las intersecciones de la ventana con el área de relleno.

## 6.9 RECORTE DE CURVAS

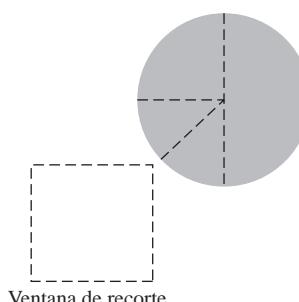
---

Las áreas con límites curvos se pueden recortar con métodos similares a los estudiados en los apartados anteriores. Si los objetos se aproximan mediante secciones de línea recta, se puede emplear un método de recorte de polígonos. De otro modo, los procedimientos de recorte implican ecuaciones no lineales, y esto requiere más procesamiento que para los objetos con límites lineales.

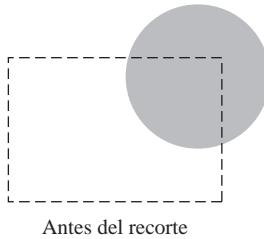
En primer lugar podemos comprobar las extensiones de coordenadas de un objeto respecto a los límites de recorte, para determinar si es posible aceptar trivialmente o rechazar el objeto entero. Si no, podríamos comprobar las simetrías del objeto que podríamos explotar en las pruebas iniciales de aceptación o rechazo. Por ejemplo, los círculos presentan simetrías entre cuadrantes y octantes, por lo que podríamos comprobar las extensiones de las coordenadas de estas regiones individuales del círculo. No podemos rechazar el área de relleno circular completa de la Figura 6.31 sólo comprobando sus extensiones de coordenadas en conjunto. Pero la mitad del círculo está fuera del límite derecho de recorte (o fuera del límite superior), el cuadrante superior izquierdo está encima del límite superior de recorte, y los dos octantes restantes se pueden eliminar de forma similar.

Un cálculo de intersección implica la sustitución de una posición en el límite de recorte ( $xw_{\min}$ ,  $xw_{\max}$ ,  $yw_{\min}$ , o  $yw_{\max}$ ) en la ecuación no lineal de los límites del objeto y la resolución para la otra coordenada. Una vez que todas las intersecciones se han evaluado, los puntos que definen el objeto se pueden almacenar para su uso posterior en procedimientos de relleno por línea de barrido. La Figura 6.32 muestra el recorte de un círculo con una ventana rectangular. En este ejemplo, el radio del círculo y los puntos extremos del arco recortado se pueden utilizar para llenar la región recortada, invocando el algoritmo del círculo para localizar los puntos a lo largo del arco entre los puntos extremos de intersección.

Se pueden aplicar procedimientos similares cuando se recorta un objeto curvado con una región de recorte de forma poligonal general. En el primer paso, podríamos comparar el rectángulo delimitador del objeto con el rectángulo delimitador de la región de recorte. Si no se almacena o elimina el objeto entero, a continuación resolvemos el sistema de ecuaciones de líneas y las curvas para determinar los puntos de intersección de recorte.



**FIGURA 6.31.** Un área de relleno circular, mostrando el cuadrante y los octantes que están fuera de los límites de la ventana de recorte.



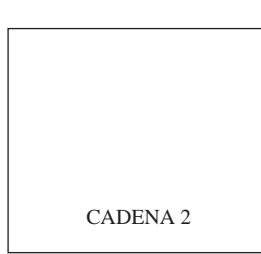
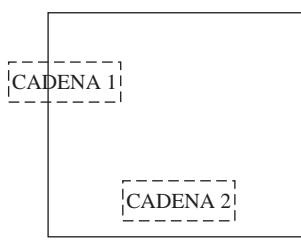
**FIGURA 6.32.** Recorte de un área de relleno circular.

## 6.10 RECORTE DE TEXTOS

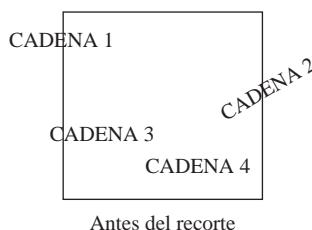
Existen varias técnicas que se pueden utilizar para proporcionar recorte de textos en un paquete gráfico. En una aplicación concreta, la elección del método de recorte depende de cómo se generen los caracteres y de qué requisitos tengamos para la visualización de cadenas de caracteres.

El método más simple para procesar las cadenas de caracteres relativo a los límites de la ventana de recorte consiste en utilizar la estrategia de *recorte de cadenas todo o ninguno* mostrada en la Figura 6.33. Si toda la cadena se encuentra dentro de la ventana de recorte, visualizamos la cadena completa. En caso contrario, se elimina la cadena completa. Este procedimiento se implementa examinando las extensiones de coordenadas de la cadena de caracteres. Si los límites de las coordenadas de este rectángulo delimitador no están completamente dentro de la ventana de recorte, se rechaza la cadena.

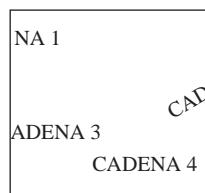
Una alternativa consiste en utilizar la estrategia de *recorte de caracteres todo o ninguno*. Aquí eliminamos sólo aquellos caracteres que no están completamente dentro de la ventana de recorte (Figura 6.34). En este



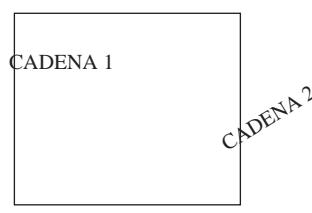
**FIGURA 6.33.** Recorte de texto utilizando las extensiones de coordenadas para una cadena completa.



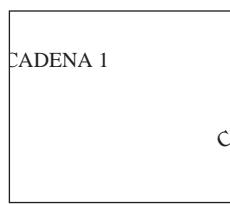
Antes del recorte



Después del recorte

**FIGURA 6.34.** Recorte de texto utilizando el rectángulo delimitador para caracteres individuales de una cadena.

Antes del recorte



Después del recorte

**FIGURA 6.35.** Recorte de texto realizado sobre las componentes de los caracteres individuales.

caso, las extensiones de coordenadas de los caracteres individuales se comparan con los límites de la ventana. Cualquier carácter que no esté completamente dentro del límite de la ventana de recorte se elimina.

Una tercera técnica para recortar textos consiste en recortar los componentes de los caracteres individuales. Esto proporciona la visualización más precisa de cadenas de caracteres recortadas, pero requiere el mayor procesamiento. Ahora tratamos los caracteres del mismo modo que tratabamos las líneas o los polígonos. Si un carácter individual se superpone con la ventana de recorte, sólo recortamos las partes del carácter que se encuentran fuera de la ventana (Figura 6.35). Las fuentes de caracteres de contorno definidas con segmentos de línea se procesan de este modo utilizando un algoritmo de recorte de polígonos. Los caracteres definidos como mapas de bits se recortan comparando la posición relativa de los píxeles individuales en los patrones de rejilla de caracteres con los límites de la región de recorte.

## 6.11 RESUMEN

---

La pipeline de transformación de visualización bidimensional es una serie de operaciones que da lugar a la visualización de una imagen en coordenadas universales que se han definido en el plano  $xy$ . Después de construir la escena, se puede mapear a un sistema de referencia de coordenadas de visualización y, a continuación, a un sistema de coordenadas normalizado donde se pueden aplicar subrutinas de recorte. Finalmente, la escena se transfiere a coordenadas de dispositivo para su visualización. Las coordenadas normalizadas se pueden especificar dentro del rango que varía de 0 a 1 o dentro del rango que varía de -1 a 1, y se utilizan para hacer los paquetes gráficos independientes de los requisitos de los dispositivos de salida.

Seleccionamos parte de una escena para su visualización en un dispositivo de salida utilizando una ventana de recorte, que se puede describir en el sistema de coordenadas universales o en el sistema de coordenadas de visualización relativo a las coordenadas universales. Los contenidos de la ventana de recorte se transfieren a un visor para su visualización en un dispositivo de salida. En algunos sistemas, un visor se especifica en coordenadas normalizadas. Otros sistemas especifican el visor en coordenadas de dispositivo. Habitualmente, la ventana de recorte y el visor son rectángulos cuyas aristas son paralelas a los ejes de coordenadas. Un objeto se mapea al visor para que tenga la misma posición relativa en el visor que la que tiene en la ventana de recorte. Para mantener las proporciones relativas de un objeto, el visor debe tener la misma relación de aspecto que la ventana de recorte correspondiente. Podemos establecer un número cualquiera de ventanas de recorte y de visores para una escena.

Los algoritmos de recorte se implementan habitualmente en coordenadas normalizadas, para que todas las transformaciones geométricas y las operaciones de visualización, que son independientes de las coordenadas del dispositivo, se puedan concatenar en una matriz de transformación. Con el visor especificado en coordenadas de dispositivo, podemos recortar una escena bidimensional con un cuadrado simétrico y normalizado con coordenadas normalizadas que varían entre -1 y 1, antes de transferir los contenidos del cuadrado normalizado y simétrico al visor.

Todos los paquetes gráficos incluyen subrutinas para recortar segmentos de línea recta y áreas de relleno poligonales. Los paquetes que contienen funciones para especificar puntos únicos o cadenas de texto, también incluyen subrutinas de recorte para estas primitivas gráficas. Ya que los cálculos de recorte consumen un tiempo, el desarrollo de algoritmos de recorte mejorados continúa siendo un área de gran preocupación en los gráficos por computadora. Cohen y Sutherland desarrollaron un algoritmo de recorte de líneas que utiliza un código de región para identificar la posición de un extremo de una línea respecto a los límites de la ventana de recorte. Los códigos de región de los puntos extremos se utilizan para identificar rápidamente aquellas líneas que se encuentran completamente dentro de la ventana de recorte y algunas líneas que están completamente fuera. Para el resto de las líneas, se deben calcular las intersecciones con los límites de la ventana. Liang y Barsky desarrollaron un algoritmo de recorte de líneas más rápido que representa los segmentos de línea con ecuaciones paramétricas, similar al algoritmo de Cyrus-Beck. Esta técnica permite que se realicen más comprobaciones antes de proceder con los cálculos de intersección. El algoritmo de Nicholl-Lee-Nicholl reduce más aún los cálculos de intersección utilizando una comprobación de más regiones en el plano  $xy$ . Los métodos de recorte de líneas paramétricos se amplían fácilmente a ventanas de recorte convexas y a escenas tridimensionales. Sin embargo, la técnica de Nicholl-Lee-Nicholl sólo es aplicable a segmentos de línea bidimensionales.

También se han desarrollado algoritmos de recorte de segmentos de línea recta con ventanas de recorte poligonales cóncavas. Una técnica consiste en dividir una ventana de recorte cóncava en un conjunto de polígonos convexos y aplicar los métodos de recorte de líneas paramétricos. Otra técnica consiste en añadir aristas a la ventana cóncava para convertirla en convexa. Después se puede realizar una serie de operaciones de recorte interiores y exteriores para obtener el segmento de línea recortado.

Aunque las ventanas de recorte con límites curvados se utilizan raramente, podemos aplicar métodos similares a los de recorte de líneas. Sin embargo, los cálculos de intersección ahora implican ecuaciones no lineales.

Un área de relleno de un polígono se define con una lista de vértices, y los procedimientos de recorte de polígonos deben guardar información a cerca de cómo las aristas recortadas, se deben conectar a medida que el polígono pasa por varios estados de procesamiento. En el algoritmo de Sutherland-Hodgman, los pares de vértices del área de relleno son procesados por cada recortador de contorno por turnos y la información de recorte para aquella arista se pasa inmediatamente al siguiente recortador, lo cual permite a las cuatro subrutinas de recorte (izquierda, derecha, inferior y superior) funcionar en paralelo. Este algoritmo proporciona un método eficiente para recortar áreas de relleno de polígonos convexos. Sin embargo, cuando un polígono cóncavo contiene partes disjuntas, el algoritmo de Sutherland-Hodgman produce extraños segmentos de línea de conexión. También se pueden utilizar para recortar áreas de relleno de polígonos convexos, ampliaciones de recortadores de líneas paramétricos, tales como el método de Liang-Barsky. Tanto las áreas de relleno convexas como las áreas de relleno cóncavas se pueden recortar correctamente con el algoritmo de Weiler-Atherton, que utiliza una técnica de contorno-transversal.

Las áreas de relleno se pueden recortar con ventanas de recorte convexas utilizando una ampliación de la técnica de representación de líneas paramétricas. El método de Weiler-Atherton puede recortar cualquier área de relleno de un polígono utilizando cualquier ventana de recorte de forma poligonal. Las áreas de relleno se pueden recortar mediante ventanas con límites no lineales utilizando una aproximación poligonal de la ventana o procesando el área de relleno con los límites curvados de la ventana.

El método de recorte de textos más rápido es la estrategia todos-o-ninguno, que recorta completamente una cadena de texto si cualquier parte de la cadena está fuera de cualquier límite de la ventana de recorte. Podríamos recortar una cadena de texto eliminando sólo aquellos caracteres de la cadena que no están completamente dentro de la ventana de recorte. El método más preciso de recorte de texto consiste en aplicar recorte de puntos, líneas, polígonos o curvas a los caracteres individuales de una cadena, dependiendo de si los caracteres están definidos con fuentes de cuadrículas de puntos o de contorno.

Aunque OpenGL está diseñado para aplicaciones tridimensionales, se proporciona una función GLU bidimensional para especificar una ventana de recorte estándar y rectangular en coordenadas universales. En OpenGL, las coordenadas de la ventana de recorte son parámetros para la transformación de proyección. Por tanto, en primer lugar necesitamos invocar el modo de la matriz de proyección. A continuación, podemos especificar la vista, utilizando una función de la biblioteca básica de OpenGL y una ventana de visualización, utilizando funciones de GLUT. Hay disponible una gran variedad de funciones de GLUT para modificar varios parámetros de la ventana de visualización. La Tabla 6.1 resume las funciones de visualización bidimensional de OpenGL. Adicionalmente, la tabla enumera algunas funciones relacionadas con la visualización.

**TABLA 6.1. RESUMEN DE LAS FUNCIONES OpenGL DE VISUALIZACIÓN BIDIMENSIONAL.**

Función	Descripción
gluOrtho2D	Especifica las coordenadas de la ventana de recorte en sus argumentos de una proyección ortogonal bidimensional.
glViewport	Especifica los parámetros del sistema de coordenadas de pantalla de una vista.
glGetIntegerv	Utiliza los argumentos GL_VIEWPORT y vpArray para obtener los parámetros de la vista activa actual.
glutInit	Inicializa la biblioteca GLUT.
glutInitWindowPosition	Especifica las coordenadas de la esquina superior izquierda de la ventana de visualización.
glutInitWindowSize	Especifica la anchura y la altura de la ventana de visualización.
glutCreateWindow	Crea una ventana de visualización (a la que se asigna un identificador entero) y especifica un título para la misma.

(Continúa)

**TABLA 6.1.** RESUMEN DE LAS FUNCIONES DE VISUALIZACIÓN BIDIMENSIONAL DE OpenGL. (*Cont.*)

<i>Función</i>	<i>Descripción</i>
glutInitDisplayMode	Selecciona parámetros tales como los búferes y el modo de color de una ventana de visualización.
glClearColor	Especifica un color RGB de fondo en una ventana de visualización.
glClearIndex	Especifica un color de fondo de una ventana de visualización utilizando el modo de color indexado.
glutDestroyWindow	Especifica un número identificador de una ventana de visualización que hay que borrar.
glutSetWindow	Especifica el número identificador de una ventana de visualización que debe ser establecida como la ventana de visualización actual.
glutPositionWindow	Restablece la localización en la pantalla de la ventana de visualización actual.
glutReshapeWindow	Restablece la anchura y la altura de la ventana de visualización actual.
glutFullScreen	Cambia la ventana de visualización actual al tamaño de la pantalla de vídeo.
glutReshapeFunc	Especifica la función que se debe invocar cuando se cambia el tamaño de la ventana de visualización.
glutIconifyWindow	Convierte la ventana de visualización actual en un ícono.
glutSetIconTitle	Especifica una etiqueta para un ícono de la ventana de visualización.
glutSetWindowTitle	Especifica un nuevo título para la ventana de visualización actual.
glutPopWindow	Mueve la ventana de visualización actual hacia la «cima»; es decir, la coloca encima de las demás ventanas.
glutPushWindow	Mueve la ventana de visualización actual hacia el «fondo»; es decir, la coloca detrás de las demás ventanas.
glutShowWindow	Muestra en pantalla la ventana de visualización actual.
GlutCreateSubWindow	Crea una ventana de segundo nivel dentro de una ventana de visualización.
glutSetCursor	Selecciona una forma para el cursor de pantalla.
glutDisplayFunc	Invoca una función para crear una imagen dentro de la ventana de visualización actual.
glutPostRedisplay	Redibuja los contenidos de la ventana actual.
glutMainLoop	Ejecuta el programa de gráficos por computadora.
glutIdleFunc	Especifica una función para su ejecución cuando el sistema no tiene algo que hacer.
glutGet	Pregunta al sistema acerca de un parámetro de estado especificado.

## REFERENCIAS

---

Se estudian algoritmos de recorte de líneas en Sproull y Sutherland (1968), Cyrus y Beck (1978), Liang y Barsky (1984), y Nicholl, Lee y Nicholl (1987). Se proporcionan métodos de mejora de la velocidad del algoritmo de recorte de líneas de Cohen-Sutherland en Duvanenko (1990).

En Sutherland y Hodgman (1974) y en Liang y Barsky (1983) se presentan métodos básicos de recorte de polígonos. En Weiler y Atherton (1977) y en Weiler (1980) se proporcionan técnicas generales para recortar polígonos de forma arbitraria entre sí.

En Woo, Neider, Davis y Shreiner (1999) se estudian funciones de visualización de OpenGL. Las subrutinas de GLUT de ventanas de visualización se estudian en Kilgard (1996) y se puede obtener información adicional sobre GLUT en el sitio web: <http://reality.sgi.com/opengl/glut3/glut3.html>.

## EJERCICIOS

---

- 6.1 Escriba un procedimiento para calcular los elementos de la matriz 6.1 de transformación bidimensional de coordenadas universales a coordenadas de visualización, proporcionando las coordenadas del origen de visualización  $P_0$  y el vector de dirección de vista  $V$ .
- 6.2 Obtenga la matriz 6.8 de transferencia de los contenidos de una ventana de recorte a la vista cambiando de escala en primer lugar la ventana al tamaño de la vista y después trasladando la ventana a la que se le ha aplicado el cambio de escala a la posición de la vista. Utilice el centro de la ventana de recorte como punto de referencia para las operaciones de cambio de escala y de traslación.
- 6.3 Escriba un procedimiento para calcular los elementos de la matriz 6.9 de transformación de una ventana de recorte en el cuadrado normalizado y simétrico.
- 6.4 Escriba un conjunto de procedimientos para implementar la tubería de visualización bidimensional sin las operaciones de recorte. Su programa debería permitir que se construya una escena con transformaciones de coordenadas de modelado, un sistema de visualización especificado y una transformación al cuadrado normalizado y simétrico. De forma opcional, se podría implementar una tabla de visualización para almacenar diferentes conjuntos de parámetros de transformación de visualización.
- 6.5 Escriba un programa completo para implementar el algoritmo de recorte de líneas de Cohen-Sutherland.
- 6.6 Estudie cuidadosamente el fundamento lógico de los tests y los métodos para el cálculo de los parámetros de las intersecciones  $u_1$  y  $u_2$  del algoritmo de recorte de líneas de Liang-Barsky.
- 6.7 Compare el número de operaciones aritméticas realizados en los algoritmos de recorte de líneas de Cohen-Sutherland y Liang-Barsky para varias orientaciones diferentes de línea relativas a la ventana de recorte.
- 6.8 Escriba un programa completo para implementar el algoritmo de recorte de líneas de Liang-Barsky.
- 6.9 Idee transformaciones de simetría para mapear los cálculos de intersecciones para las tres regiones de la Figura 6.16 a las otras seis regiones del plano  $xy$ .
- 6.10 Establezca un algoritmo detallado para la técnica de Nicholl-Lee-Nicholl de recorte de líneas para cualquier par de puntos extremos de línea de entrada.
- 6.11 Compare el número de operaciones aritméticas realizadas en el algoritmo NLN con los algoritmos de Cohen-Sutherland y de Liang-Barsky de recorte de líneas, para varias orientaciones diferentes de línea respecto de la ventana de recorte.
- 6.12 Adapte el algoritmo de recorte de líneas de Liang-Barsky al recorte de polígonos.
- 6.13 Establezca un algoritmo detallado para el recorte de polígonos de Weiler-Atherton, asumiendo que la ventana de recorte es un rectángulo en posición estándar.
- 6.14 Idee un algoritmo para el recorte de polígonos de Weiler-Atherton, en el que la ventana de recorte pueda ser cualquier polígono convexo.

- 6.15 Idee un algoritmo para el recorte de polígonos de Weiler-Atherton, en el que la ventana de recorte pueda ser cualquier polígono especificado (convexo o cóncavo).
- 6.16 Escriba una subrutina para recortar una elipse en posición estándar con una ventana rectangular.
- 6.17 Asumiendo que todos los caracteres de una cadena de texto tienen la misma anchura, desarrolle un algoritmo de recorte de texto que recorte una cadena según la estrategia de recorte de caracteres todos-o-ninguno.
- 6.18 Desarrolle un algoritmo de recorte de texto que recorte caracteres individuales, asumiendo que los caracteres se definen en una cuadrícula de píxeles de un tamaño especificado.

## CAPÍTULO 7

# Visualización tridimensional



El Templo de Luxor, una escena de un vídeo de E&S Digital Theater donde se realiza una visualización tridimensional en tiempo real con técnicas de animación infográfica. (*Cortesía de Evans & Sutherland.*)

<b>7.1</b>	Panorámica de los conceptos de visualización tridimensional	<b>7.7</b>	Proyecciones paralelas oblicuas
<b>7.2</b>	<i>Pipeline</i> de visualización tridimensional	<b>7.8</b>	Proyecciones en perspectiva
<b>7.3</b>	Parámetros de coordenadas de visualización tridimensional	<b>7.9</b>	Transformación del visor y coordenadas de pantalla tridimensionales
<b>7.4</b>	Transformación de coordenadas universales a coordenadas de visualización	<b>7.10</b>	Funciones de visualización tridimensional OpenGL
<b>7.5</b>	Transformaciones de proyección	<b>7.11</b>	Algoritmos de recorte tridimensional
<b>7.6</b>	Proyecciones ortogonales	<b>7.12</b>	Planos de recorte opcionales en OpenGL
		<b>7.13</b>	Resumen

Para aplicaciones gráficas bidimensionales, las operaciones de visualización transfieren las posiciones desde el sistema de referencia en coordenadas universales a posiciones de píxeles en el plano del dispositivo de salida. Utilizando el contorno rectangular de la ventana de visualización y el visor, un paquete gráfico bidimensional recorta una escena y establece la correspondencia con las coordenadas del dispositivo de salida. Las operaciones de visualización tridimensionales, sin embargo, son más complejas, ya que hay muchas más opciones en lo que respecta al modo de construir la escena y de generar vistas de dicha escena sobre un dispositivo de salida.

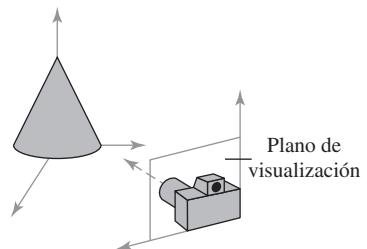
## 7.1 PANORÁMICA DE LOS CONCEPTOS DE VISUALIZACIÓN TRIDIMENSIONAL

---

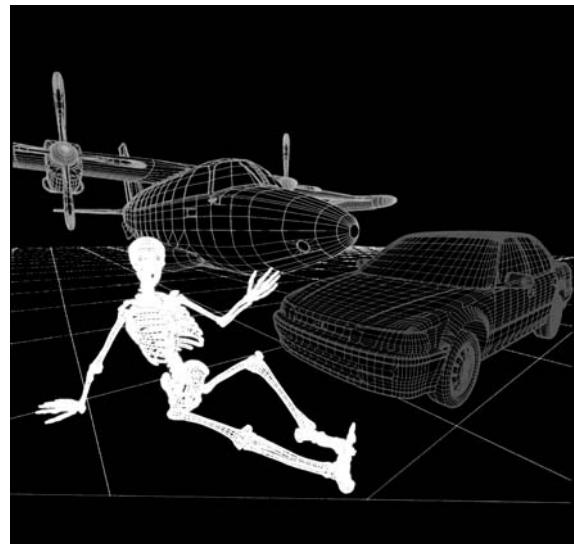
Cuando modelamos una escena tridimensional, cada objeto de la escena se suele definir mediante un conjunto de superficies que forman un conjunto cerrado alrededor del interior del objeto. Y para algunas aplicaciones puede que también tengamos que especificar información acerca de la estructura interior del objeto. Además de proporcionar procedimientos que generan vistas de las características superficiales de un objeto, los paquetes gráficos proporcionan en ocasiones rutinas para mostrar los componentes internos o vistas en sección transversal de un objeto sólido. Las funciones de visualización procesan a las descripciones de los objetos mediante una serie de procedimientos que proyectan una vista especificada de los objetos sobre la superficie de un dispositivo de salida. Muchos procesos de visualización tridimensional, como las rutinas de recorte, son similares a los que podemos encontrar en una pipeline de visualización bidimensional. Pero la visualización tridimensional requiere algunas otras tareas que no están presentes en su equivalente bidimensional. Por ejemplo, se necesitan rutinas de proyección para transferir la escena a una vista sobre una superficie plana; asimismo, es necesario identificar las partes visibles de una escena y, para tener una imagen realista, es necesario tener en cuenta los efectos de iluminación y las características de las superficies.

### Visualización de una escena tridimensional

Para obtener una imagen de una escena tridimensional definida en coordenadas universales, primero necesitamos definir un sistema de referencia de coordenadas para los parámetros de visualización, o parámetros de la «cámara». Este sistema de referencia define la posición y orientación de un *plano de visualización* (o *plano de proyección*) que se corresponde con el plano de la película de una cámara (Figura 7.1). Entonces, se transfieren las descripciones de los objetos al sistema de coordenadas de visualización y se proyectan sobre el



**FIGURA 7.1.** Sistema de coordenadas de referencia para obtener una vista seleccionada de una escena tridimensional.



**FIGURA 7.2.** Visualización alámbrica de tres objetos en la que se han eliminado las líneas posteriores; las imágenes están extraídas de una base de datos comercial de formas de objetos. Cada objeto de la base de datos está definido mediante una cuadrícula tridimensional de puntos que pueden visualizarse en forma alámbrica o como un sólido con superficie sombreada. (*Cortesía de Viewpoint DataLabs.*)

plano de visualización. Podemos generar una vista de un objeto sobre el dispositivo de salida en modo alámbrico, como en la Figura 7.2, o podemos aplicar técnicas de iluminación y de representación de superficies para obtener un sombreado realista de las superficies visibles.

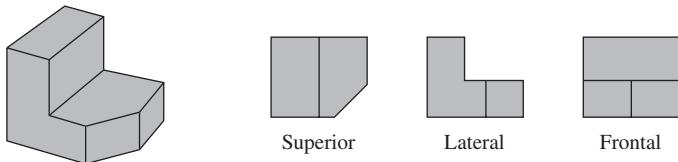
## Proyecciones

A diferencia de la imagen de una cámara, podemos elegir diferentes métodos para proyectar una escena sobre el plano de visualización. Un método para obtener la descripción de un objeto sólido sobre un plano de visualización consiste en proyectar puntos de la superficie del objeto en una serie de líneas paralelas. Esta técnica, denominada *proyección paralela*, se utiliza en dibujos de ingeniería y de arquitectura para representar un objeto mediante una serie de vistas que indiquen sus dimensiones precisas, como en la Figura 7.3.

Otro método para generar una vista de una escena tridimensional consiste en proyectar los puntos hasta el plano de visualización según una serie de trayectorias convergentes. Este proceso, denominado *proyección en perspectiva*, hace que los objetos que están situados más lejos de la posición de visualización se muestren con un tamaño menor que los objetos del mismo tamaño que se encuentren más cerca. Las escenas generadas utilizando una proyección en perspectiva parecen más realistas, ya que ésta es la manera en que nuestros ojos y los objetivos de las cámaras forman las imágenes. En la vista con proyección en perspectiva mostrada en la Figura 7.4, las líneas paralelas según la dirección de visualización parecen converger en un punto distante del fondo, y los aviones situados más lejos parecen más pequeños que el que está despegando en primer plano.

## Pistas de profundidad

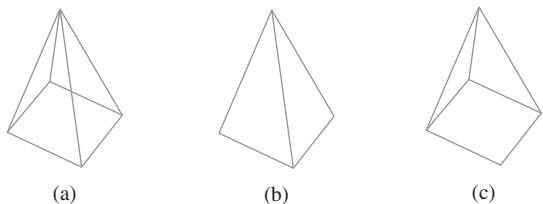
Con muy pocas excepciones, la información de profundidad tiene una gran importancia en una escena tridimensional, para que podamos identificar fácilmente (para una dirección de visualización concreta) cuál es la



**FIGURA 7.3.** Tres vistas en proyección paralela de un objeto, que muestran las proporciones relativas desde distintas posiciones de visualización.



**FIGURA 7.4.** Una vista con proyección en perspectiva de una escena de un aeropuerto. (Cortesía de Evans & Sutherland.)

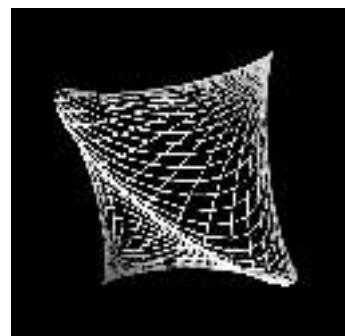


**FIGURA 7.5.** La representación alámbrica de la pirámide en (a) no contiene información de profundidad para indicar si la dirección de visualización es (b) hacia abajo desde una posición situada por encima de la cúspide o (c) hacia arriba desde una posición situada por debajo de la base.

parte la frontal y cuál es la parte posterior de cada objeto visualizado. La Figura 7.5 ilustra la ambigüedad que puede producirse cuando se muestra la representación alámbrica de un objeto sin información de profundidad. Hay muchas formas de incluir la información de profundidad en la representación bidimensional de objetos sólidos.

Un método simple para indicar la profundidad en las visualizaciones alámbricas consiste en hacer variar el brillo de los segmentos lineales de acuerdo con su distancia con respecto a la posición de visualización. La Figura 7.6 muestra una representación alámbrica de un objeto que incluye *pistas de profundidad*. Las líneas más próximas a la posición de visualización se muestran con la intensidad más alta, mientras que las líneas más alejadas se van mostrando con intensidades decrecientes. Este tipo de pistas de profundidad se aplican seleccionando unos valores de intensidad máximos y mínimos y un rango de distancias a lo largo de las cuales dicha intensidad puede variar.

Otra aplicación de las pistas de profundidad es modelar el efecto de la atmósfera sobre la intensidad percibida de los objetos. Los objetos más distantes parecen más tenues que los objetos más próximos, debido a la dispersión de la luz por parte de las partículas de polvo, de la niebla y del humo. Algunos efectos atmosféricos pueden incluso cambiar el color percibido de un objeto y las pistas de profundidad nos permiten modelar estos efectos.



**FIGURA 7.6.** Una representación alámbrica de un objeto con información de profundidad, de modo que el brillo de las líneas disminuye desde la parte frontal del objeto hacia la parte posterior.

### Identificación de líneas y superficies visibles

También podemos clarificar las relaciones de profundidad en una representación alámbrica de una escena utilizando técnicas distintas de la variación de las intensidades con la distancia. Una posibilidad consiste simplemente en resaltar las líneas visibles o en mostrarlas con un color diferente. Otra técnica, comúnmente utilizada en los dibujos de ingeniería, consiste en mostrar las líneas no visibles como líneas punteadas. También podemos eliminar las líneas no visibles de la imagen como en la Figura 7.5(b) y 7.5(c). Pero al eliminarse las líneas ocultas también se elimina información acerca de la forma de las superficies posteriores de un objeto, y la realidad es que las representaciones alámbricas se utilizan generalmente para obtener una indicación de la apariencia global de un objeto, incluyendo tanto su parte frontal como la posterior.

Cuando hay que generar una vista realista de una escena, se eliminan completamente las partes posteriores de los objetos, de modo que sólo se muestran las superficies visibles. En este caso, se aplican procedimientos de representación superficial para que los píxeles de la pantalla sólo contengan la información de color correspondiente a las superficies frontales.

### Representación de superficies

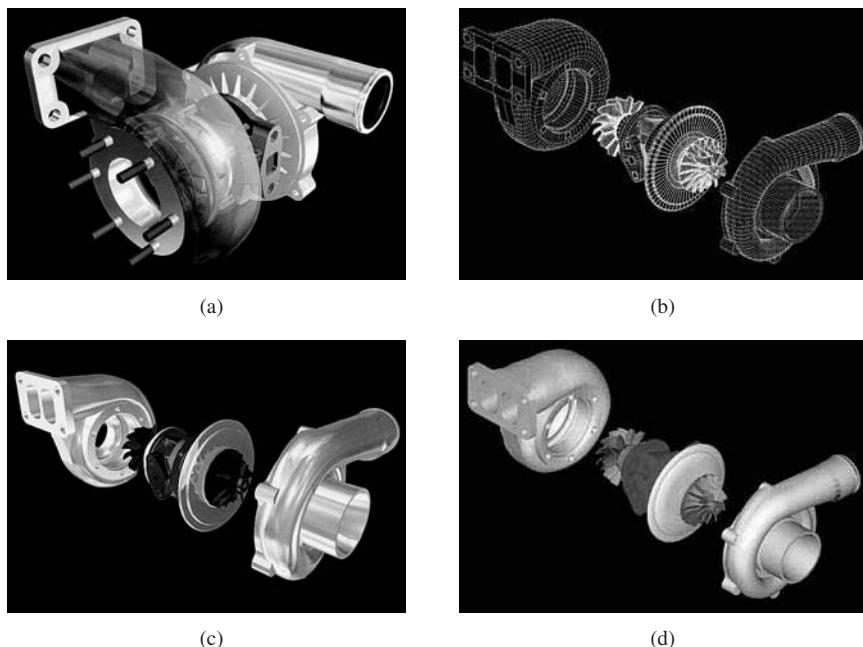
Puede conseguirse un mayor realismo en las imágenes si se representan las superficies de los objetos utilizando las condiciones de iluminación de una escena y las características asignadas a las superficies. Las condiciones de iluminación se establecen especificando el color y la ubicación de las fuentes de luz, y también pueden definirse efectos de iluminación de fondo. Las propiedades de las superficies de los objetos incluyen, por ejemplo, información sobre si una superficie es transparente u opaca, o si es suave o rugosa. Podemos asignar valores a una serie de parámetros para modelar superficies tales como el cristal, el plástico, superficies de madera veteada o incluso la apariencia rugosa de una naranja. En la Figura 7.7, se combinan métodos de representación superficial junto con técnicas de proyección en perspectiva y de identificación de superficies visibles con el fin de conseguir un alto grado de realismo en la escena mostrada.



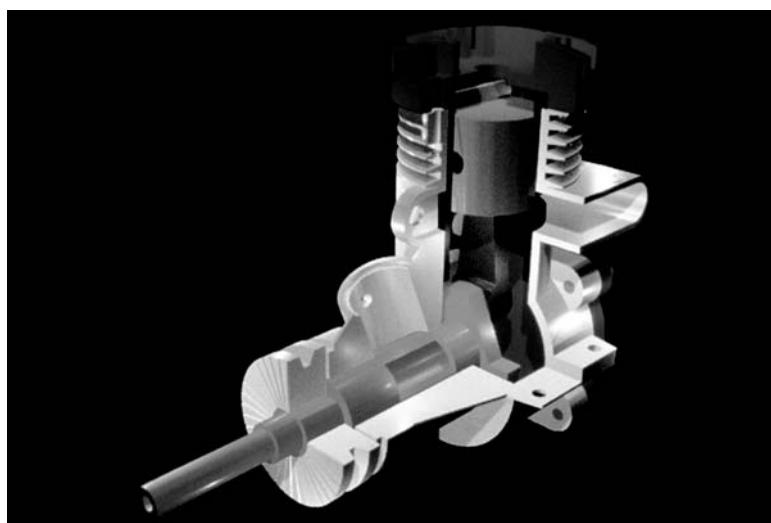
**FIGURA 7.7** Una visualización realista de una habitación, conseguida mediante una proyección en perspectiva, efectos de iluminación y propiedades seleccionadas de la superficie. (Cortesía de John Snyder, Jed Lengyel, Devendra Kalra y Al Barr, California Institute of Technology. Copyright © 1992 Caltech.)

## Despiece y secciones transversales

Muchos paquetes gráficos permiten definir los objetos como estructuras jerárquicas, con lo que pueden almacenarse los detalles internos. Entonces, pueden utilizarse despieces y secciones transversales de dichos objetos para mostrar la estructura interna y las relaciones de las partes del objeto. La Figura 7.8 muestra diferentes tipos de despieces para un diseño mecánico. Una alternativa al despiece de un objeto con el fin de mostrar sus partes componentes consiste en generar una sección transversal, como en la Figura 7.9, que elimina partes de las superficies visibles con el fin de que se vea la estructura interna.



**FIGURA 7.8.** Una turbina completamente montada (a) puede mostrarse mediante un despiece con representación alámbrica (b), mediante un despiece renderizado (c), o mediante un despiece renderizado con codificación de colores (d). (Cortesía de Autodesk, Inc.)



**FIGURA 7.9.** Sección transversal con codificación de colores del motor de una segadora, donde se muestra la estructura y las relaciones de los componentes internos. (Cortesía de Autodesk, Inc.)

## Visualización tridimensional y estereoscópica

Otros métodos para dar una sensación de realismo a una escena infográfica incluyen la utilización de monitores tridimensionales y vistas estereoscópicas. Como hemos visto en el Capítulo 2, pueden obtenerse imágenes tridimensionales reflejando una imagen bidimensional en un espejo vibratorio y flexible. Las vibraciones del espejo están sincronizadas con la visualización de la escena en el TRC. A medida que el espejo, la longitud focal varía, de modo que cada punto de la escena se refleja a una posición espacial que se corresponde con su profundidad.

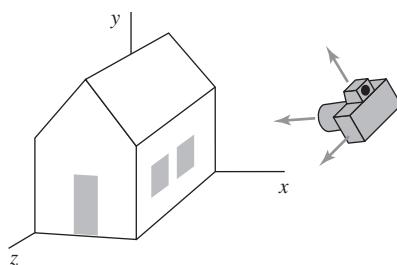
Los dispositivos estereoscópicos presentan dos vistas de una escena: una para el ojo izquierdo y la otra para el ojo derecho. Las posiciones de visualización se corresponden con las posiciones de los ojos de la persona que contempla la escena. Estas dos vistas se suelen mostrar en ciclos alternativos de refresco de un monitor convencional. Cuando contemplamos el monitor mediante unas gafas especiales que oscurecen alternativamente primero una de las lentes y luego la otra, en sincronización con los ciclos de refresco del monitor, se puede ver la escena con un efecto tridimensional.

## 7.2 PIPELINE DE VISUALIZACIÓN TRIDIMENSIONAL

Los procedimientos para generar una vista infográfica de una escena tridimensional son análogos, hasta cierto punto, a los procesos que tienen lugar cuando se toma una fotografía. En primer lugar, necesitamos seleccionar una posición de visualización, que se correspondería con el lugar donde situaríamos la cámara. La posición de visualización se seleccionará dependiendo de si queremos mostrar una vista frontal, trasera, lateral, superior o inferior de la escena. También podemos seleccionar una posición en medio de un grupo de objetos o incluso dentro de un único objeto, como por ejemplo un edificio o una molécula. Entonces, debemos decidir cuál debe ser la orientación de la cámara (Figura 7.10). ¿Hacia dónde queremos apuntar la cámara desde la posición de visualización y cómo debemos rotarla con respecto a la línea de visualización con el fin de definir qué dirección de la imagen corresponde a «arriba»? Finalmente, cuando tomamos la fotografía, la escena se recorta al tamaño de una ventana de recorte seleccionada, que se corresponde con la apertura o tipo de objetivo de una cámara, proyectando sólo la luz de las superficies visibles sobre la película fotográfica.

Necesitamos tener presente, sin embargo, que la analogía de la cámara no se puede llevar mucho más allá, ya que con un programa infográfico tenemos mucha más flexibilidad y muchas más opciones para generar vistas de una escena que con una cámara real. Podemos decidir utilizar una proyección paralela o en perspectiva, podemos eliminar selectivamente partes de una escena que estén colocadas a lo largo de la línea de visión, podemos alejar el plano de proyección de la posición de la «cámara» e incluso podemos obtener una imagen de los objetos situados detrás de nuestra cámara sintética.

Algunas de las operaciones de visualización para una escena tridimensional son iguales o similares a las que se emplean en una pipeline de visualización bidimensional (Sección 6.1). Se utiliza un visor bidimensional para posicionar una vista proyectada de la escena tridimensional sobre el dispositivo de salida, y se emplea

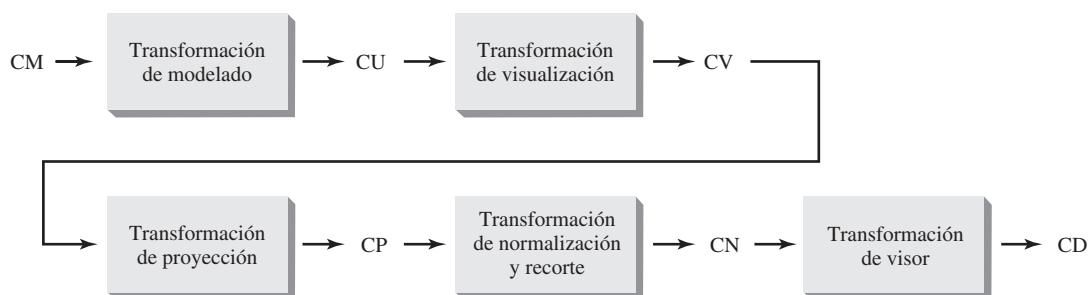


**FIGURA 7.10.** La fotografía de una escena implica seleccionar la posición y orientación de la cámara.

una ventana de recorte bidimensional para seleccionar la vista que haya que asignar al visor. También hay que definir una ventana de visualización en coordenadas de pantalla, igual que se hace en las aplicaciones bidimensionales. Las ventanas de recorte, los visores y las ventanas de visualización se suelen especificar como rectángulos con sus lados paralelos a los ejes de coordenadas. Sin embargo, en visualización tridimensional la ventana de recorte se posiciona sobre un plano de visualización seleccionado y las escenas se recortan de acuerdo con un volumen que lo encierra, el cual está definido mediante un conjunto de *planos de recorte*. La posición de visualización, el plano de visualización, la ventana de recorte y los planos de recorte se especifican con respecto al sistema de coordenadas de visualización.

La Figura 7.11 muestra los pasos generales de procesamiento para la creación y transformación de una escena tridimensional a coordenadas de dispositivo. Una vez modelada la escena en coordenadas universales, se selecciona un sistema de coordenadas de visualización cuya descripción de la escena se convierte a este nuevo sistema. El sistema de coordenadas de visualización define los parámetros de visualización, incluyendo la posición y orientación del plano de proyección (plano de visualización), que sería análogo al plano de la película de la cámara. A continuación se define una ventana de recorte bidimensional sobre el plano de proyección, que se correspondería con un objetivo de la cámara seleccionado, y se establece una región de recorte tridimensional. Esta región de recorte se denomina **volumen de visualización** y su forma y tamaño dependen de las dimensiones de la ventana de recorte, del tipo de proyección que seleccionemos y de las posiciones límite seleccionadas según la dirección de visualización. Después, se realizan operaciones de proyección para convertir la descripción de la escena en coordenadas de visualización a un conjunto de coordenadas sobre el plano de proyección. Los objetos se mapean a coordenadas normalizadas y todas las partes de la escena que caen fuera del volumen de visualización se recortan. Las operaciones de recorte pueden aplicarse después de completar todas las transformaciones de coordenadas independientes del dispositivo (de coordenadas universales a coordenadas normalizadas). De esta forma, las transformaciones de coordenadas pueden confecionarse con el fin de conseguir la máxima eficiencia.

Al igual que en la visualización bidimensional, los límites del visor pueden especificarse en coordenadas normalizadas o en coordenadas del dispositivo. A la hora de desarrollar los algoritmos de visualización, asumiremos qué hay que especificar del visor en coordenadas del dispositivo y que las coordenadas normalizadas deben transferirse a coordenadas del visor después de las operaciones de recorte. Hay otras cuantas tareas que es preciso llevar a cabo, como especificar las superficies visibles y aplicar los procedimientos de representación superficial. El paso final consiste en establecer la correspondencia entre las coordenadas del visor y las coordenadas del dispositivo, dentro de una ventana de visualización seleccionada. Las descripciones de la escena en coordenadas del dispositivo se expresan en ocasiones con un sistema de referencia que cumple la regla de la mano izquierda, de modo que pueden utilizarse distancias positivas con respecto a la pantalla de visualización para medir los valores de profundidad de la escena.



**FIGURA 7.11.** Pipeline general tridimensional de transformación de la imagen, desde las coordenadas de modelado a las coordenadas universales y desde éstas a las coordenadas de visualización y a las coordenadas de proyección. Finalmente, se efectúa la transformación a coordenadas normalizadas y después a coordenadas del dispositivo.

## 7.3 PARÁMETROS DE COORDENADAS DE VISUALIZACIÓN TRIDIMENSIONAL

Establecer un sistema de referencia de visualización tridimensional es similar al caso bidimensional analizado en la Sección 6.2. Primero se selecciona una posición en coordenadas universales  $\mathbf{P}_0 = (x_0, y_0, z_0)$  para el origen de la visualización, que se denominará **punto de vista** o **posición de visualización**. (Algunas veces el punto de vista también se denomina *posición del ojo* o *posición de la cámara*.) También se especifican un **vector vertical  $\mathbf{V}$** , que define la dirección  $y_{\text{view}}$ . Para un espacio tridimensional, necesitamos también asignar una dirección para uno de los dos restantes ejes de coordenadas. Esto se suele llevar a cabo con un segundo vector que define el eje  $z_{\text{view}}$ , estando la dirección de visualización definida sobre este eje. La Figura 7.12 ilustra el posicionamiento de un sistema de referencia de visualización tridimensional dentro de un sistema de coordenadas universales.

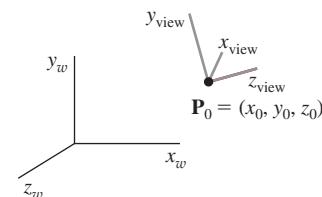
### Vector normal del plano de visualización

Puesto que la dirección de visualización suele estar definida según el eje  $z_{\text{view}}$ , normalmente se asume que el **plano de visualización**, también llamado **plano de proyección**, es perpendicular a este eje. Así, la orientación del plano de visualización y la dirección del eje  $z_{\text{view}}$  positivo pueden definirse mediante un **vector normal del plano de visualización  $\mathbf{N}$** , como se muestra en la Figura 7.13. Se utiliza un parámetro escalar adicional para establecer la posición del plano de visualización en un cierto valor de coordenada  $z_{vp}$  según el eje  $z_{\text{view}}$ , como se ilustra en la Figura 7.14. Este parámetro se suele especificar como una distancia con respecto al origen de visualización según la dirección de visualización, que normalmente se toma en la dirección  $z_{\text{view}}$  negativa. Así, el plano de visualización es siempre paralelo al plano  $x_{\text{view}}y_{\text{view}}$  y la proyección de los objetos sobre el plano de visualización se corresponde con la vista de la escena que se mostrará en el dispositivo de salida.

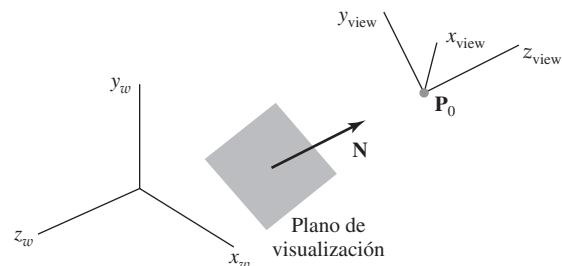
El vector  $\mathbf{N}$  puede especificarse de diversas formas. En algunos sistemas gráficos, la dirección de  $\mathbf{N}$  se define según la línea que va del origen de coordenadas universales hasta un punto seleccionado. Otros sistemas definen  $\mathbf{N}$  en la dirección que va desde un punto de referencia  $\mathbf{P}_{\text{ref}}$  hasta el origen de coordenadas de visualización  $\mathbf{P}_0$ , como en la Figura 7.15. En este caso, el punto de referencia suele denominarse *punto observado* de la escena, siendo la dirección de visualización opuesta a la dirección de  $\mathbf{N}$ .

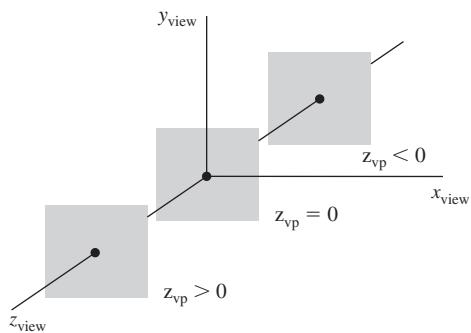
También podríamos definir el vector normal al plano de visualización, y otras direcciones de vectores, utilizando **ángulos de dirección**. Estos son los tres ángulos  $\alpha$ ,  $\beta$  y  $\gamma$  que una línea espacial forma con los ejes  $x$ ,  $y$  y  $z$ , respectivamente. Pero normalmente resulta bastante más fácil especificar la dirección de un vector mediante dos puntos de una escena que mediante los ángulos de dirección.

**FIGURA 7.12.** Un sistema de coordenadas de visualización que cumple la regla de la mano derecha, con ejes  $x_{\text{view}}$ ,  $y_{\text{view}}$  y  $z_{\text{view}}$  relativo a un sistema de referencia en coordenadas universales que también cumple la regla de la mano derecha.

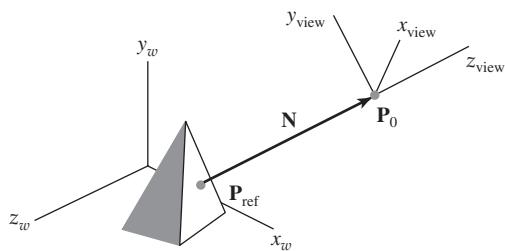


**FIGURA 7.13.** Orientación del plano de visualización y del vector normal a dicho plano  $\mathbf{N}$ .





**FIGURA 7.14.** Tres posibles posiciones para el plano de visualización según el eje  $z_{\text{view}}$ .



**FIGURA 7.15.** Especificación del vector normal al plano de visualización  $\mathbf{N}$  como la dirección desde un punto de referencia seleccionado  $\mathbf{P}_{\text{ref}}$  hasta el origen de las coordenadas de visualización  $\mathbf{P}_0$ .

## El vector vertical

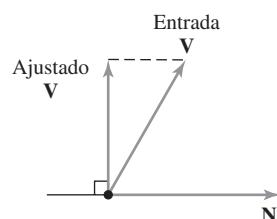
Una vez seleccionado el vector normal al plano de visualización  $\mathbf{N}$ , podemos establecer la dirección del vector vertical  $\mathbf{V}$ . Este vector se emplea para establecer la dirección positiva del eje  $y_{\text{view}}$ .

Usualmente,  $\mathbf{V}$  se define seleccionando una posición relativa al origen de coordenadas universales, de modo que la dirección del vector vertical va desde el origen de coordenadas universales a dicha posición seleccionada. Puesto que el vector normal al plano de visualización  $\mathbf{N}$  define la dirección del eje  $z_{\text{view}}$ , el vector  $\mathbf{V}$  debe ser perpendicular a  $\mathbf{N}$ . Pero, en general, puede resultar difícil determinar una dirección de  $\mathbf{V}$  que sea precisamente perpendicular a  $\mathbf{N}$ . Por tanto, las rutinas de visualización suelen ajustar la dirección del vector  $\mathbf{V}$  definida por el usuario como se muestra en la Figura 7.16, de modo que se proyecta  $\mathbf{V}$  sobre un plano que sea perpendicular al vector normal al plano de visualización.

Podemos seleccionar la dirección que queramos para el vector vertical  $\mathbf{V}$ , siempre y cuando éste no sea paralelo a  $\mathbf{N}$ . Una elección conveniente suele ser según una dirección paralela al eje universal  $y_w$ ; en otras palabras, podemos definir  $\mathbf{V} = (0, 1, 0)$ .

## Sistema de referencia de coordenadas de visualización uvn

En los paquetes gráficos se utilizan en ocasiones coordenadas de visualización que cumplen la regla de la mano izquierda, estando la visualización definida en la dirección  $z_{\text{view}}$ . Con un sistema que cumpla la regla de la mano izquierda, los valores crecientes de  $z_{\text{view}}$  representan puntos progresivamente más alejados de la posición de visualización, según la línea de visión. Pero suelen ser más comunes los sistemas de visualización



**FIGURA 7.16.** Ajuste de la dirección de entrada del vector vertical  $\mathbf{V}$  a una orientación perpendicular al vector normal al plano de visualización  $\mathbf{N}$ .

que cumplen la regla de la mano derecha, ya que éstos tienen la misma orientación que el sistema de referencia en coordenadas universales. Esto permite que el paquete gráfico sólo tenga que tratar con una única orientación de coordenadas tanto para el sistema de referencia universal como para el de visualización. Aunque algunos de los primeros paquetes gráficos que aparecieron definían las coordenadas de visualización mediante un sistema de referencia que cumplía la regla de la mano izquierda, los estándares gráficos actuales deciden las coordenadas de visualización sobre la regla de la mano derecha. De todos modos, a menudo se emplean sistemas de referencia que cumplen la regla de la mano izquierda para representar las coordenadas de pantalla y para la transformación de normalización.

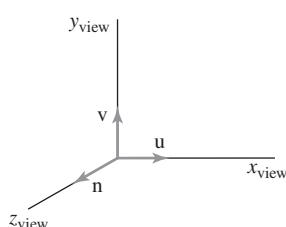
Puesto que la normal al plano de visualización  $\mathbf{N}$  define la dirección del eje  $z_{\text{view}}$  y el vector vertical  $\mathbf{V}$  se utiliza para obtener la dirección del eje  $y_{\text{view}}$ , lo único que nos queda es determinar la dirección del eje  $x_{\text{view}}$ . Utilizando los valores introducidos para  $\mathbf{N}$  y  $\mathbf{V}$ , calculamos un tercer vector  $\mathbf{U}$  que sea perpendicular tanto a  $\mathbf{N}$  como a  $\mathbf{V}$ . El vector  $\mathbf{U}$  definirá entonces la dirección del eje  $x_{\text{view}}$  positivo. Podemos determinar la dirección correcta de  $\mathbf{U}$  realizando el producto vectorial de  $\mathbf{V}$  y  $\mathbf{N}$  de modo que se genere un sistema de referencia de visualización que cumpla la regla de la mano derecha. El producto vectorial de  $\mathbf{N}$  y  $\mathbf{U}$  también nos permite obtener el valor ajustado de  $\mathbf{V}$ , que será perpendicular tanto a  $\mathbf{N}$  como a  $\mathbf{U}$ , según el eje  $y_{\text{view}}$  positivo. De acuerdo con estos procedimientos, se obtiene el siguiente conjunto de vectores unitarios de coordenadas para un sistema de coordenadas de visualización que cumpla la regla de la mano derecha:

$$\begin{aligned}\mathbf{n} &= \frac{\mathbf{N}}{|\mathbf{N}|} = (n_x, n_y, n_z) \\ \mathbf{u} &= \frac{\mathbf{V} \times \mathbf{n}}{|\mathbf{V}|} = (u_x, u_y, u_z) \\ \mathbf{v} &= \mathbf{n} \times \mathbf{u} = (v_x, v_y, v_z)\end{aligned}\tag{7.1}$$

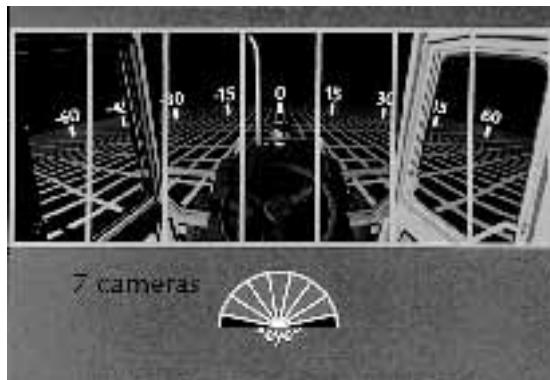
El sistema de coordenadas formado por estos tres vectores unitarios se suele denominar **sistema de referencia uvn de coordenadas de visualización** (Figura 7.17).

## Generación de efectos de visualización tridimensionales

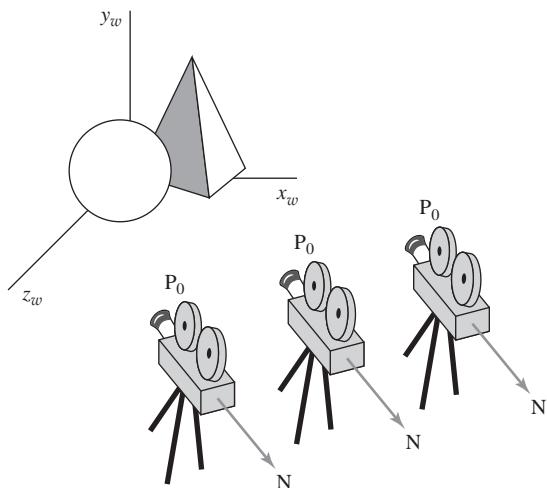
Variando los parámetros de visualización, podemos obtener diferentes vistas de los objetos de una escena. Por ejemplo, desde una posición de visualización fija, podríamos cambiar la posición de  $\mathbf{N}$  para mostrar los objetos situados en las posiciones que rodean el origen del sistema de coordenadas de visualización. También podemos variar  $\mathbf{N}$  para crear una imagen compuesta que contenga múltiples vistas tomadas desde una posición de cámara fija. La Figura 7.18 muestra una imagen de gran angular creada para un entorno de realidad virtual. El efecto de gran angular se consigue generando siete vistas de la escena desde la misma posición de visualización, pero con pequeños desplazamientos de la dirección de visualización y combinando luego las vistas con el fin de mostrar una imagen compuesta. De forma similar, podemos generar vistas estereoscópicas desplazando la dirección de visualización. En este caso, sin embargo, también tendremos que desplazar el punto de vista para simular la posición de los dos ojos.



**FIGURA 7.17.** Un sistema de visualización que cumple la regla de la mano derecha, definido mediante los vectores unitarios  $\mathbf{u}$ ,  $\mathbf{v}$  y  $\mathbf{n}$ .



**FIGURA 7.18.** Una vista de gran angular para imagen de realidad virtual generada con siete secciones, cada una producida con una dirección de visualización ligeramente distinta. (Cortesía del National Center for Supercomputing Applications, Universidad de Illinois en Urbana-Champaign.)

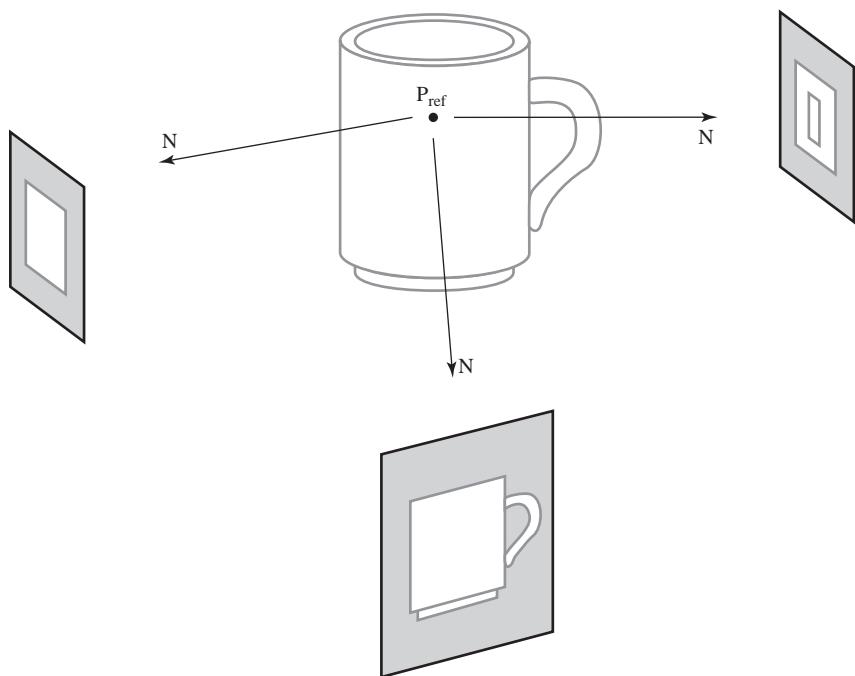


**FIGURA 7.19.** Panorámica de una escena, obtenida cambiando la posición de visualización, pero conservando una posición fija para  $\mathbf{N}$ .

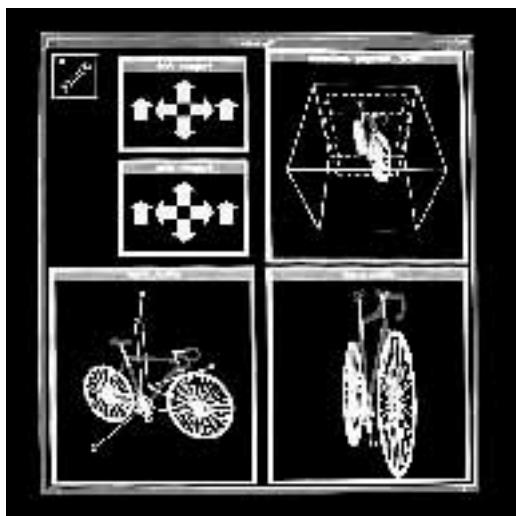
En las aplicaciones interactivas, el vector normal  $\mathbf{N}$  es el parámetro de normalización que más a menudo se suele cambiar. Por supuesto, cuando cambiamos la dirección de  $\mathbf{N}$ , también tenemos que cambiar los vectores de los otros ejes, con el fin de mantener un sistema de referencia de visualización que cumpla la regla de la mano derecha.

Si queremos simular un efecto panorámico de animación, como por ejemplo cuando una cámara se mueve a través de una escena o sigue a un objeto que se está moviendo a través de una escena, podemos mantener la dirección  $\mathbf{N}$  fija a medida que desplazamos el punto de vista, como se ilustra en la Figura 7.19. Y para mostrar diferentes vistas de un objeto, como por ejemplo una vista lateral y una vista frontal, podemos mover el punto de vista alrededor del objeto, como en la Figura 7.20. Alternativamente, pueden generarse diferentes vistas de un objeto de un grupo de objetos utilizando transformaciones geométricas, sin cambiar los parámetros de visualización.

La Figura 7.21 muestra una interfaz desarrollada para la selección interactiva de los valores de los parámetros de visualización. Este software incluye un editor de visualización, mecanismos de selección de múltiples ventanas de visualización, menús y otras herramientas de interfaz.



**FIGURA 7.20.** Visualización de un objeto desde diferentes direcciones utilizando un punto de referencia fijo.



**FIGURA 7.21.** Una interfaz interactiva para controlar los parámetros de visualización, desarrollada en la Universidad de Manchester utilizando la herramienta PHIGS Toolkit. (Cortesía de T. L. J. Howard, J. G. Williams y W. T. Hewitt, Department of Computer Science, Universidad de Manchester, Reino Unido.)

## 7.4 TRANSFORMACIÓN DE COORDENADAS UNIVERSALES A COORDENADAS DE VISUALIZACIÓN

En la *pipeline* de visualización tridimensional, el primer paso después de construir una escena consiste en transferir las descripciones de los objetos al sistema de coordenadas de visualización. Esta conversión de las descripciones de los objetos es equivalente a una secuencia de transformaciones que superpone el sistema de

coordenadas de visualización sobre el sistema de coordenadas universales. Podemos realizar esta conversión utilizando los métodos de transformación de sistemas de coordenadas descritos en la Sección 5.15:

- (1) Desplazar el origen de coordenadas de visualización al origen del sistema de coordenadas universales.
- (2) Aplicar rotaciones para alinear los ejes  $x_{\text{view}}$ ,  $y_{\text{view}}$  y  $z_{\text{view}}$  con los ejes de coordenadas universales  $x_w$ ,  $y_w$  y  $z_w$ , respectivamente.

El origen de coordenadas de visualización se encuentra en la posición  $\mathbf{P} = (x_0, y_0, z_0)$  en coordenadas universales. Por tanto, la matriz para desplazar el origen de coordenadas de visualización al origen de coordenadas universales es:

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & -x_0 \\ 0 & 1 & 0 & -y_0 \\ 0 & 0 & 1 & -z_0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (7.2)$$

Para la transformación de rotación, podemos utilizar los vectores unitarios  $\mathbf{u}$ ,  $\mathbf{v}$  y  $\mathbf{n}$  para formar la matriz de rotación compuesta que superpone los ejes de visualización sobre el sistema de referencia universal. Esta matriz de transformación es:

$$\mathbf{R} = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (7.3)$$

donde los elementos de la matriz  $\mathbf{R}$  son las componentes de los vectores de los ejes  $\mathbf{uvn}$ .

La matriz de transformación de coordenadas se obtiene entonces calculando el producto de las dos matrices anteriores de desplazamiento y de rotación:

$$\mathbf{M}_{WC,VC} = \mathbf{R} \cdot \mathbf{T}$$

$$= \begin{bmatrix} u_x & u_y & u_z & -\mathbf{u} \cdot \mathbf{P}_0 \\ v_x & v_y & v_z & -\mathbf{v} \cdot \mathbf{P}_0 \\ n_x & n_y & n_z & -\mathbf{n} \cdot \mathbf{P}_0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (7.4)$$

Los factores de desplazamiento de esta matriz se calculan como el producto escalar de cada uno de los vectores unitarios  $\mathbf{u}$ ,  $\mathbf{v}$  y  $\mathbf{n}$  por  $\mathbf{P}_0$ , que representa un vector que va desde el origen de coordenadas universales hasta el origen de coordenadas de visualización. En otras palabras, los factores de desplazamiento son el negado de las proyecciones de  $\mathbf{P}_0$  sobre cada uno de los ejes de coordenadas de visualización (el negado de las componentes de  $\mathbf{P}_0$  en coordenadas de visualización). Estos elementos de la matriz se evalúan de la forma siguiente:

$$-\mathbf{u} \cdot \mathbf{P}_0 = -x_0 u_x - y_0 u_y - z_0 u_z \quad (7.5)$$

$$-\mathbf{v} \cdot \mathbf{P}_0 = -x_0 v_x - y_0 v_y - z_0 v_z$$

$$-\mathbf{n} \cdot \mathbf{P}_0 = -x_0 n_x - y_0 n_y - z_0 n_z$$

La Matriz 7.4 transfiere las descripciones de los objetos en coordenadas universales al sistema de referencia de visualización.

## 7.5 TRANSFORMACIONES DE PROYECCIÓN

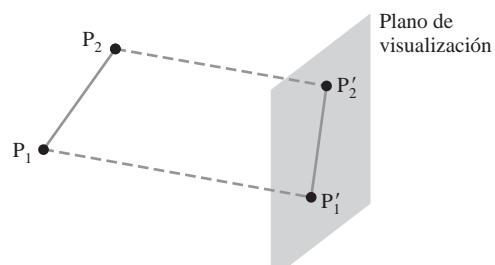
En la siguiente fase de la pipeline de visualización tridimensional, después de realizada la transformación a coordenadas de visualización, las descripciones de los objetos se proyectan sobre el plano de visualización. Los paquetes gráficos soportan en general tanto proyecciones paralelas como en perspectiva.

En una **proyección paralela**, las coordenadas se transfieren al plano de visualización según una serie de líneas paralelas. La Figura 7.22 ilustra una proyección paralela para un segmento de línea recta definido mediante los extremos  $P_1$  y  $P_2$ . Una proyección paralela preserva las proporciones relativas de los objetos, y éste es el método utilizado en diseño asistido por computadora con el fin de generar imágenes a escala de objetos tridimensionales. Todas las líneas paralelas de una escena se muestran como paralelas cuando se las contempla mediante una proyección paralela. Hay dos métodos generales para obtener una vista de proyección paralela de un objeto: podemos proyectar el objeto según líneas que sean perpendiculares al plano de visualización, o podemos utilizar un ángulo oblicuo con respecto al plano de visualización.

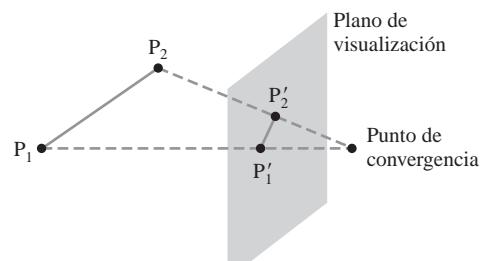
Para una **proyección en perspectiva**, las posiciones de los objetos se transforman a las coordenadas de proyección según una serie de líneas que convergen en un punto situado detrás del plano de visualización. En la Figura 7.23 se proporciona un ejemplo de proyección en perspectiva para un segmento de línea recta definido mediante los extremos  $P_1$  y  $P_2$ . A diferencia de una proyección paralela, la proyección en perspectiva no preserva las proporciones relativas de los objetos. Sin embargo, las vistas en perspectiva de una escena son más realistas, porque los objetos distantes tienen un tamaño más pequeño en la imagen proyectada.

## 7.6 PROYECCIONES ORTOGONALES

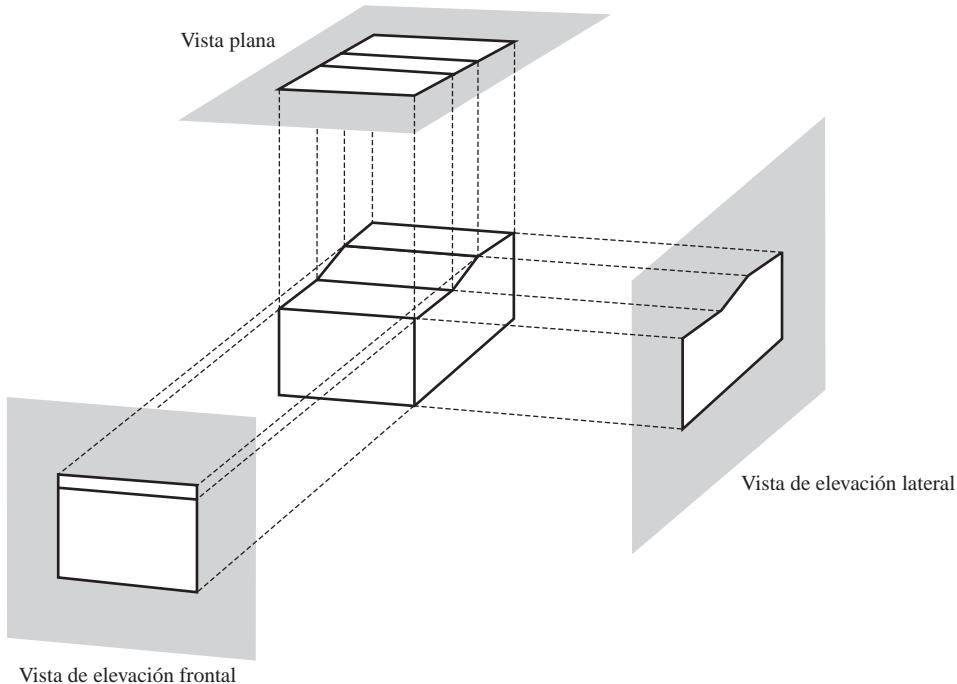
Una transformación de un objeto a un plano de visualización según líneas paralelas al vector normal al plano de visualización  $N$  se denomina **proyección ortogonal** (o también **proyección ortográfica**). Esto produce una transformación de proyección paralela en la que las líneas de proyección son perpendiculares al plano de visualización. Las proyecciones ortogonales se suelen utilizar para generar las vistas frontal, lateral y superior de un objeto, como se muestra en la Figura 7.24. Las proyecciones ortogonales frontal, lateral y posterior de un objetos se denominan *elevaciones*, mientras que una proyección ortogonal superior se denomina *vista plana*. Los gráficos de ingeniería y arquitectura suelen emplear estas proyecciones ortográficas, ya



**FIGURA 7.22.** Proyección paralela de un segmento de línea sobre un plano de visualización.



**FIGURA 7.23.** Proyección en perspectiva de un segmento de línea sobre un plano de visualización.



**FIGURA 7.24.** Proyecciones ortogonales de un objeto, donde se muestran la vista plana y las vistas de elevación.

que las longitudes y los ángulos se representan de manera precisa y pueden medirse a partir de los propios gráficos.

### Proyecciones ortogonales axonométricas e isométricas

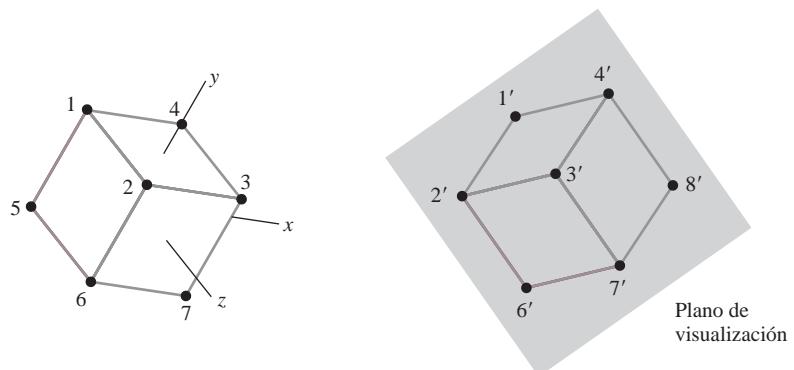
También podemos formar proyecciones ortogonales que muestren más de una cara de un objeto. Dichas vistas se denominan proyecciones ortogonales **axonométricas**. La proyección axonométrica más comúnmente utilizada es la proyección **isométrica**, que se genera alineando el plano de proyección (o el objeto) de modo que el plano intersecte todos los ejes de coordenadas sobre los que está definido el objeto, denominados *ejes principales*, a la misma distancia del origen. La Figura 7.25 muestra una proyección isométrica de un cubo. Podemos obtener la proyección isométrica mostrada en esta figura alineando el vector normal al plano de visualización según una diagonal del cubo. Hay ocho posiciones, una en cada octante, para obtener una vista isométrica. Los tres ejes principales se acortan de forma igual en una proyección isométrica, por lo que se mantienen las proporciones relativas. Este no es el caso en una proyección axonométrica general, donde los factores de escala pueden ser diferentes para las tres direcciones principales.

### Coordenadas de proyección ortogonal

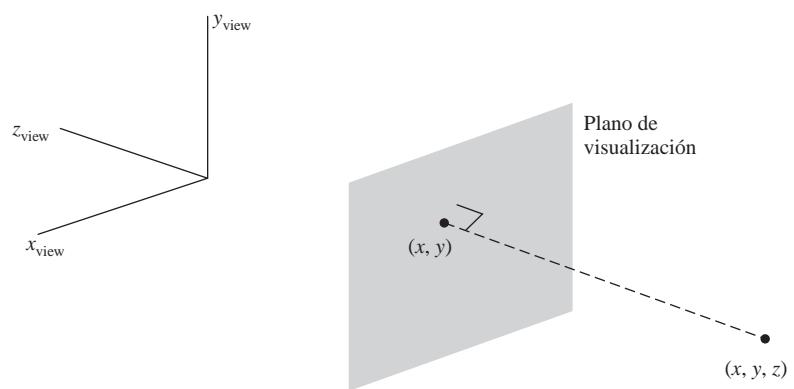
Con la dirección de proyección paralela al eje  $z_{\text{view}}$ , las ecuaciones de transformación para una proyección ortogonal son triviales. Para cualquier posición  $(x, y, z)$ , como en la Figura 7.26, las coordenadas de proyección serán:

$$x_p = x, \quad y_p = y \quad (7.6)$$

El valor de la coordenada  $z$  para cualquier transformación de proyección se preserva para su uso en los procedimientos de determinación de la visibilidad. Y cada punto en coordenadas tridimensionales de una escena se convierte a una posición dentro del espacio normalizado.



**FIGURA 7.25.** Una proyección isométrica de un cubo.



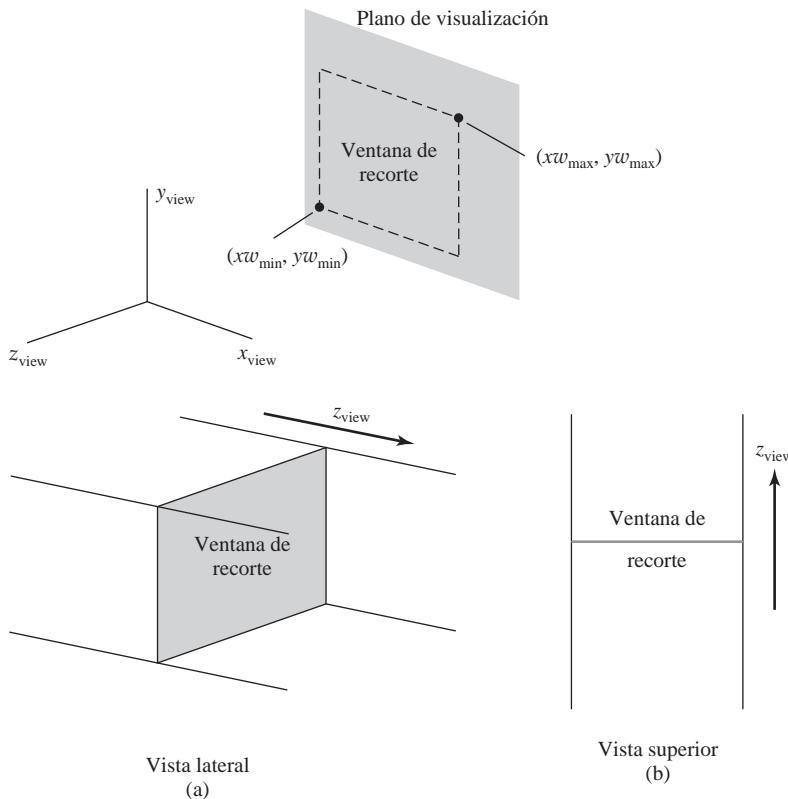
**FIGURA 7.26.** Proyección ortogonal de un punto en el espacio sobre un plano de visualización.

### Ventana de recorte y volumen de visualización de proyección ortogonal

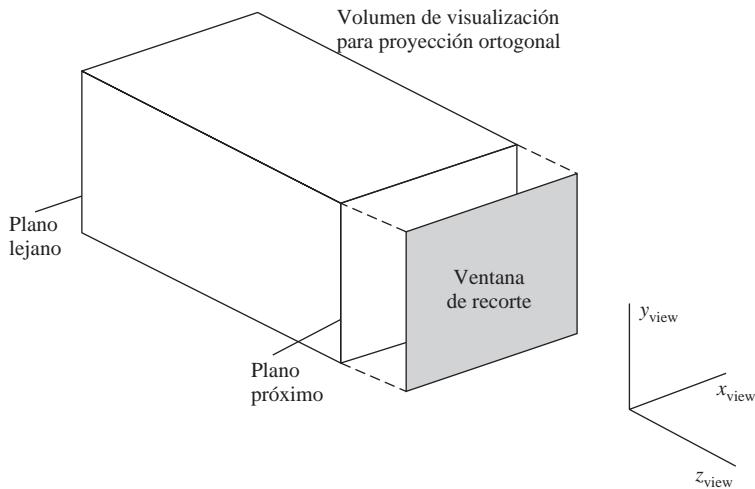
En la analogía de la cámara, el tipo del objetivo es uno de los factores que determina la cantidad de la escena que se transfiere al plano de la película. Un objetivo de gran angular permite captar una parte mayor de la escena comparado con un objetivo normal. En las aplicaciones infográficas, empleamos para este propósito la *ventana de recorte* rectangular. Al igual que la visualización bidimensional, los paquetes gráficos sólo suelen permitir utilizar rectángulos de recorte en posición estándar. Por tanto, definimos una ventana de recorte para la visualización tridimensional exactamente igual que lo hacíamos para el caso bidimensional, seleccionando posiciones de coordenadas bidimensionales para las esquinas inferior izquierda y superior derecha. Para visualización tridimensional, la ventana de recorte se posiciona sobre el plano de visualización, con sus lados paralelos a los ejes  $x_{\text{view}}$  e  $y_{\text{view}}$ , como se muestra en la Figura 7.27. Si queremos utilizar alguna otra forma geométrica o alguna otra orientación para la ventana de recorte, será necesario que desarrollemos nuestros propios procedimientos de visualización.

Los lados de la ventana de recorte especifican los límites  $x$  e  $y$  para la parte de la escena que queremos mostrar. Estos límites se emplean para formar los lados superior, inferior y laterales de una región de recorte denominada **volumen de visualización de proyección ortogonal**. Puesto que las líneas de proyección son perpendiculares al plano de visualización, estos cuatro límites son planos también perpendiculares al plano de visualización y que pasan por las aristas de la ventana de recorte, formando una región de recorte infinita, como se ilustra en la Figura 7.28.

Podemos limitar la extensión del volumen de visualización ortogonal en la dirección  $z_{\text{view}}$  seleccionando las posiciones de uno o dos planos limitantes adicionales que sean paralelos al plano de visualización. Estos dos planos se denominan **plano de recorte próximo-lejano** o **planos de recorte frontal-posterior**. Los planos próximo y lejano nos permiten excluir objetos que se encuentren delante o detrás de aquella parte de la escena que queremos mostrar. Con la dirección de visualización definida según el eje  $z_{\text{view}}$  negativo, normal-

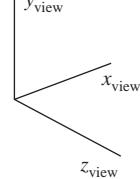


**FIGURA 7.27.** Una ventana de recorte sobre el plano de visualización, en la que las coordenadas mínimas y máximas se especifican en el sistema de referencia de visualización.



**FIGURA 7.28.** Volumen de visualización infinito para proyección ortogonal.

mente tendremos  $z_{\text{far}} < z_{\text{near}}$ , de modo que el plano lejano está más alejado según el eje  $z_{\text{view}}$  negativo. Algunas bibliotecas gráficas proporcionan estos dos planos como opciones, mientras que otras bibliotecas exigen que se utilicen. Cuando se especifican los planos próximo y lejano, se obtiene un volumen de visualización ortogonal finito que es un *paralelepípedo rectangular*, como se muestra en la Figura 7.29, donde también se indica una posible colocación del plano de visualización. Nuestra vista de la escena contendrá entonces únicamente aquellos objetos que se encuentren dentro del volumen de visualización, eliminándose mediante los algoritmos de recorte todas aquellas partes de la escena que caigan fuera del volumen de visualización.



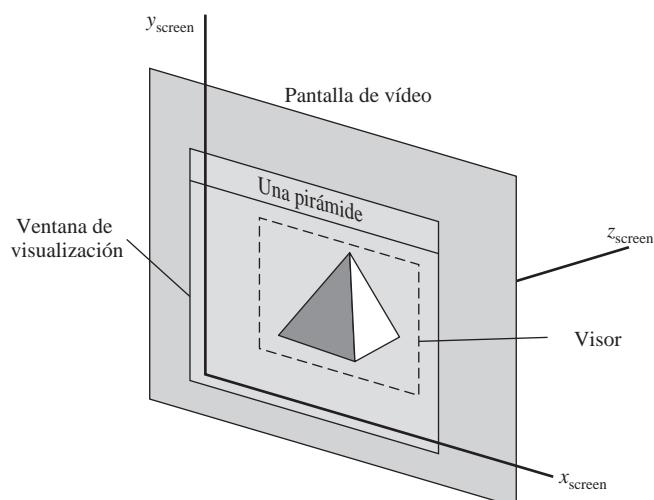
Los paquetes gráficos proporcionan diversos grados de flexibilidad a la hora de situar los planos de recorte próximo y lejano, incluyendo opciones para especificar planos de recorte adicionales en otras posiciones de la escena. En general, los planos próximo y lejano pueden estar situados en cualquier posición relativa entre sí, con el fin de conseguir diversos efectos de visualización, incluyendo posiciones que se encuentren en lados opuestos del plano de visualización. De forma similar, el plano de visualización puede en ocasiones situarse en cualquier posición relativa a los planos de recorte próximo y lejano, aunque a menudo suele coincidir con el plano de recorte próximo. Sin embargo, al proporcionar numerosas opciones de posicionamiento para los planos de recorte y de visualización, los paquetes gráficos suelen perder eficiencia a la hora de procesar escenas tridimensionales.

### Transformación de normalización para una proyección ortogonal

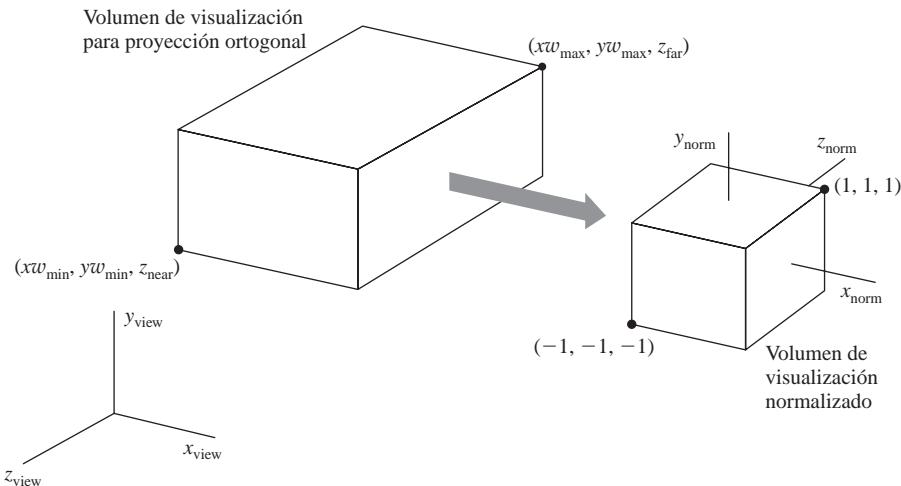
Utilizando una transferencia ortogonal de las coordenadas sobre el plano de visualización, obtenemos la proyección proyectada de cualquier punto del espacio ( $x, y, z$ ) como simplemente ( $x, y$ ). Así, una vez establecidos los límites del volumen de visualización, las coordenadas dentro de este paralelepípedo rectangular serán las coordenadas de proyección y pueden mapearse sobre un **volumen de visualización normalizado** sin necesidad de efectuar ningún procesamiento de proyección adicional. Algunos paquetes gráficos utilizan un cubo unitario para este volumen de visualización normalizado, normalizándose cada una de las coordenadas  $x$ ,  $y$  y  $z$  en el rango que va de 0 a 1. Otra técnica para la transformación de normalización consiste en utilizar un cubo simétrico, con las coordenadas en el rango que va de -1 a 1.

Puesto que las coordenadas de pantalla se suelen especificar según un sistema de referencia que cumple con la regla de la mano izquierda (Figura 7.30), las coordenadas normalizadas también suelen especificarse de la misma forma. Esto permite interpretar directamente las distancias en la dirección de visualización como distancias con respecto a la pantalla (el plano de visualización). Así, podemos convertir las coordenadas de proyección en posiciones dentro de un sistema de referencia normalizado que cumple la regla de la mano izquierda, y estas coordenadas se transformarán entonces mediante la transformación de visor en coordenadas de pantalla según un sistema que cumpla la regla de la mano izquierda.

Para ilustrar la transformación de normalización vamos a suponer que el volumen de visualización para proyección ortogonal debe asignarse al cubo de normalización simétrico dentro de un sistema de referencia que cumpla la regla de la mano izquierda. Asimismo, las posiciones de la coordenada  $z$  para los planos próximo y lejano se designan mediante  $z_{\text{near}}$  y  $z_{\text{far}}$ , respectivamente. La Figura 7.31 ilustra esta transformación de



**FIGURA 7.30.** Un sistema de coordenadas de pantalla que cumple la regla de la mano izquierda.



**FIGURA 7.31.** Transformación de normalización desde un volumen de visualización de proyección ortogonal al cubo de normalización simétrico, dentro de un sistema de referencia que cumple la regla de la mano izquierda.

normalización. La posición  $(x_{\min}, y_{\min}, z_{\text{near}})$  se mapea sobre la posición normalizada  $(-1, -1, -1)$  y la posición  $(x_{\max}, y_{\max}, z_{\text{far}})$  se mapea sobre  $(1, 1, 1)$ .

La transformación del volumen de visualización paralelepípedico rectangular en un cubo normalizado es similar a los métodos analizados en la Sección 6.3 para convertir la ventana de recorte en un cuadrado simétrico normalizado. La transformación de normalización para las posiciones  $x$  y  $y$  dentro del volumen de visualización ortogonal está dada por la matriz de normalización 6.9. Además, tenemos que transformar los valores de la coordenada  $z$  desde el rango que va de  $z_{\text{near}}$  a  $z_{\text{far}}$  al intervalo que va de  $-1$  a  $1$ , utilizando cálculos similares.

Por tanto, la transformación de normalización para el volumen de visualización ortogonal es:

$$\mathbf{M}_{\text{ortho, norm}} = \begin{bmatrix} \frac{2}{xw_{\max} - xw_{\min}} & 0 & 0 & -\frac{xw_{\max} + xw_{\min}}{xw_{\max} - xw_{\min}} \\ 0 & \frac{2}{yw_{\max} - yw_{\min}} & 0 & -\frac{yw_{\max} + yw_{\min}}{yw_{\max} - yw_{\min}} \\ 0 & 0 & \frac{-2}{z_{\text{near}} - z_{\text{far}}} & \frac{z_{\text{near}} + z_{\text{far}}}{z_{\text{near}} - z_{\text{far}}} \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (7.7)$$

Esta matriz se multiplica a la derecha por la transformación de visualización compuesta  $\mathbf{R} \cdot \mathbf{T}$  (Sección 7.4) para realizar la transformación completa desde coordenadas universales a coordenadas normalizadas de proyección ortogonal.

En este etapa de la *pipeline* de visualización, todas las transformaciones de coordenadas independientes del dispositivo se habrán completado y podrán concatenarse en una única matriz compuesta. Por tanto, la manera más eficiente de realizar los procedimientos de recorte consiste en aplicarlos después de la transformación de normalización. Después del recorte, pueden aplicarse los procedimientos de comprobación de la visibilidad, de representación de superficie y de transformación de visor para generar la imagen final de la escena en pantalla.

## 7.7 PROYECCIONES PARALELAS OBLICUAS

---

En general, una vista en proyección paralela de una escena se obtiene transfiriendo las descripciones de los objetos al plano de visualización según unas trayectorias de proyección que pueden tener cualquier dirección relativa seleccionada con respecto al vector normal del plano de visualización. Cuando la ruta de proyección no es perpendicular al plano de visualización, esta proyección se denomina **proyección paralela oblicua**. Utilizando este tipo de proyección, podemos combinar distintas vistas de un objeto, como la vista frontal, lateral y superior que se muestran en la Figura 7.32. Las proyecciones paralelas oblicuas se definen utilizando un vector de dirección para las líneas de proyección, y esta dirección puede especificarse de varias formas.

### Proyecciones paralelas oblicuas en diseño

Para aplicaciones de diseño de ingeniería y arquitectura, la proyección oblicua paralela se especifica a menudo mediante dos ángulos,  $\alpha$  y  $\phi$ , como se muestra en la Figura 7.33. Una posición del espacio  $(x, y, z)$  se proyecta sobre  $(x_p, y_p, z_{vp})$  en un plano de visualización, que estará situado en la ubicación  $z_{vp}$  según el eje de visualización  $z$ . La posición  $(x, y, z_{vp})$  es el punto correspondiente de proyección ortogonal. La línea de proyección paralela oblicua que va de  $(x, y, z)$  a  $(x_p, y_p, z_{vp})$  tiene un ángulo de intersección  $\alpha$  con la línea del plano de proyección que une  $(x_p, y_p, z_{vp})$  y  $(x, y, z_{vp})$ . Esta línea del plano de visualización, de longitud  $L$ , forma un ángulo  $\phi$  con la dirección horizontal del plano de proyección. Al ángulo  $\alpha$  puede asignársele un valor entre 0 y  $90^\circ$  y el ángulo  $\phi$  puede variar entre 0 y  $360^\circ$ . Podemos expresar las coordenadas de proyección en términos de  $x, y, L$  y  $\phi$  de la forma siguiente:

$$x_p = x + L \cos \phi \quad (7.8)$$

$$y_p = y + L \sin \phi$$

La longitud  $L$  depende del ángulo  $\alpha$  y de la distancia perpendicular del punto  $(x, y, z)$  con respecto al plano de visualización:

$$\tan \alpha = \frac{z_{vp} - z}{L} \quad (7.9)$$

Así,

$$\begin{aligned} L &= \frac{z_{vp} - z}{\tan \alpha} \\ &= L_1(z_{vp} - z) \end{aligned} \quad (7.10)$$

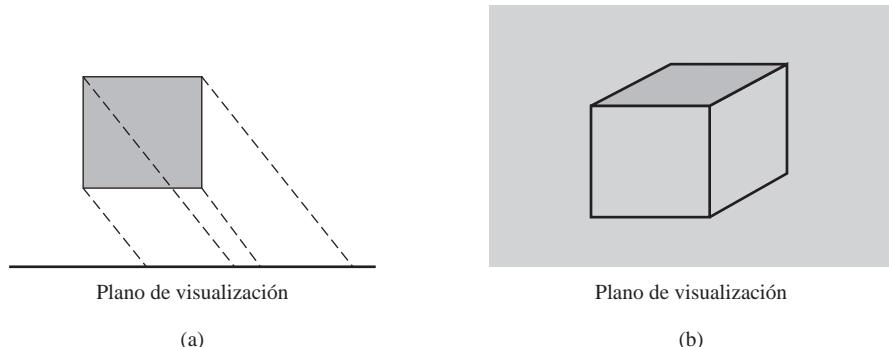
donde  $L_1 = \cot \alpha$ , que también es el valor de  $L$  cuando  $z_{vp} + z = 1$ . Podemos entonces escribir las Ecuaciones 7.8 de proyección paralela oblicua como:

$$x_p = x + L_1(z_{vp} - z) \cos \phi \quad (7.11)$$

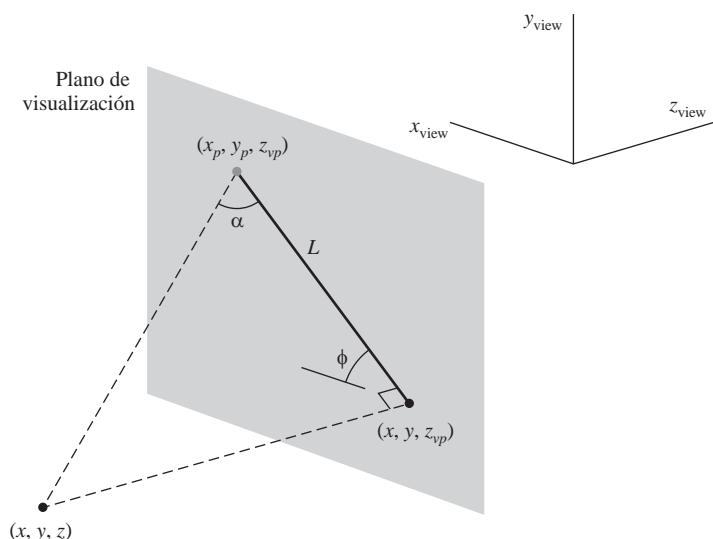
$$y_p = y + L_1(z_{vp} - z) \sin \phi$$

Podemos obtener una proyección ortogonal cuando  $L_1 = 0$  (lo que ocurre para el ángulo de proyección  $\alpha = 90^\circ$ ).

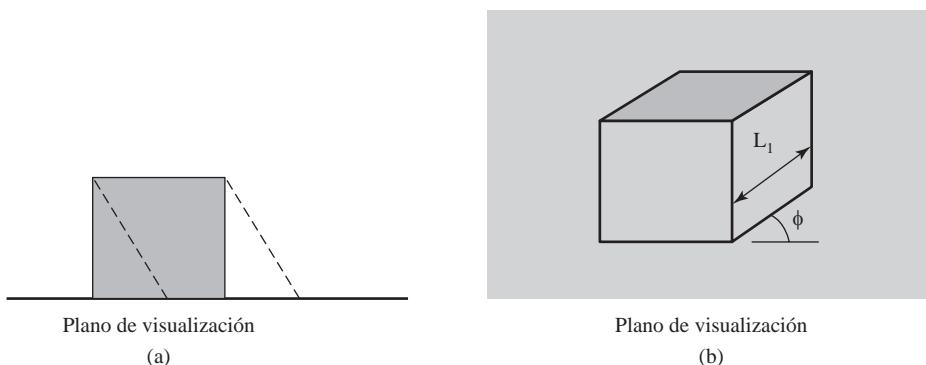
Las Ecuaciones 7.11 representan una transformación de inclinación según el eje  $z$  (Sección 5.14). De hecho, el efecto de una proyección paralela oblicua consiste en desplazar los planos de  $z$  constante y proyectarlos sobre el plano de visualización. Las posiciones  $(x, y)$  sobre cada plano de  $z$  constante se desplaza según una cantidad proporcional a la distancia de un plano con respecto al plano de visualización, de modo que todos los ángulos, distancias y líneas paralelas del plano se proyectan de manera precisa. Este efecto se muestra en



**FIGURA 7.32.** Una proyección paralela oblicua de un cubo, mostrada en una vista superior (a), puede producir una vista (b) que contenga múltiples superficies del cubo.



**FIGURA 7.33.** Proyección paralela oblicua del punto  $(x, y, z)$  a la posición  $(x_p, y_p, z_{vp})$  sobre un plano de proyección situado en la posición  $z_{vp}$  según el eje  $z_{view}$ .



**FIGURA 7.34.** Una proyección paralela oblicua (a) de un cubo (vista superior) sobre un plano de visualización que coincide con la cara frontal del cubo. Esto produce la combinación de vistas frontal, lateral y superior que se muestra en (b).

la Figura 7.34, donde el plano de visualización está situado sobre la cara frontal de un cubo. La cara posterior del cubo se proyecta y solapa con el plano frontal sobre la superficie de visualización. Una arista lateral del cubo que conecte los planos frontal y posterior se proyectará para formar una línea de longitud  $L_1$  que formará un ángulo  $\phi$  con una línea horizontal del plano de proyección.

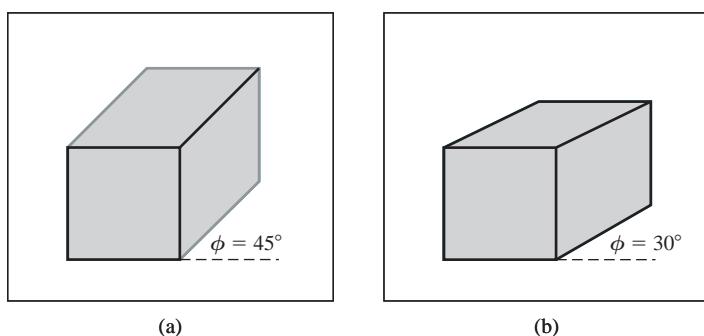
### Perspectivas caballera y cabinet

Las selecciones típicas para el ángulo  $\phi$  son  $30^\circ$  y  $45^\circ$ , lo que permite mostrar una vista combinada de las caras frontal, lateral y superior (o frontal, lateral e inferior) de un objeto. Dos valores de  $\alpha$  comúnmente utilizados son aquellos para los que  $\tan \alpha = 1$  y  $\tan \alpha = 2$ . Para el primer caso,  $\alpha = 45^\circ$  y las vistas obtenidas se denominan perspectiva **caballera**. Todas las líneas perpendiculares al plano de proyección se proyectan sin que cambie su longitud. En la Figura 7.35 se muestran ejemplos de perspectiva caballera para un cubo.

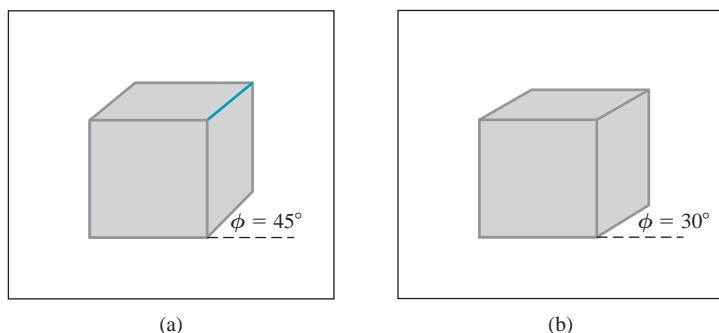
Cuando el ángulo de proyección  $\alpha$  se selecciona de modo que  $\tan \alpha = 2$ , la vista resultante se denomina perspectiva **cabinet**. Para este ángulo ( $\approx 63.4^\circ$ ), las líneas perpendiculares a la superficie de visualización se proyectan con la mitad de su longitud. La perspectiva isométrica parece más realista que la perspectiva caballera, debido a esta reducción en la longitud de las perpendiculares. La Figura 7.36 muestra ejemplos de perspectiva cabinet para un cubo.

### Vector de proyección paralela oblicua

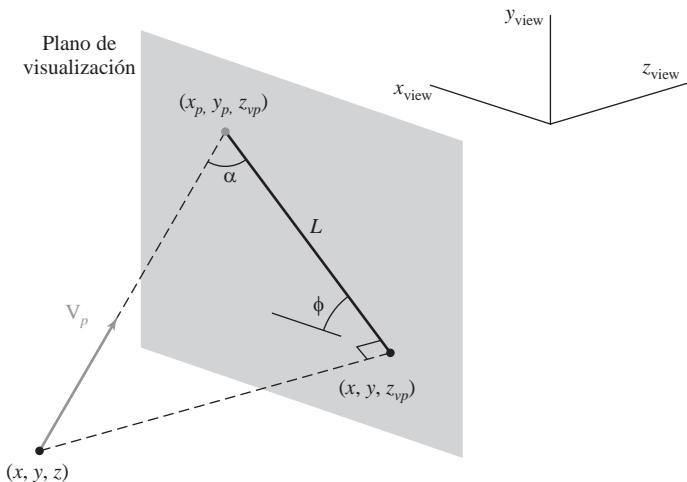
En las bibliotecas de programación gráfica que soportan proyecciones paralelas oblicuas, la dirección de proyección sobre el plano de visualización se especifica mediante un **vector de proyección paralela**  $\mathbf{V}_p$ . Este vector de dirección puede definirse mediante una posición de referencia relativa al punto de vista, como hicimos con el vector normal al plano de visualización, o mediante cualesquiera otros puntos. Algunos paquetes utilizan un punto de referencia relativo al centro de la ventana de recorte con el fin de definir la dirección de la proyección paralela. Si el vector de dirección se especifica en coordenadas universales, primero habrá que transformarlo a coordenadas de visualización utilizando la matriz de rotación descrita en la Sección 7.4 (el



**FIGURA 7.35.** Proyección caballera de un cubo sobre un plano de visualización para dos valores del ángulo  $\phi$ . La profundidad del cubo se proyecta con una longitud igual a la de la anchura y la altura.



**FIGURA 7.36.** Perspectiva cabinet de un cubo sobre un plano de visualización para dos valores del ángulo  $\alpha$ . La profundidad se proyecta con una longitud igual a la mitad de la anchura y altura del cubo.



**FIGURA 7.37.** Proyección paralela oblicua del punto  $(x, y, z)$  sobre n plano de visualización, según una línea de proyección definida mediante el vector  $\mathbf{V}_p$ .

vector de proyección no se ve afectado por el desplazamiento, ya que se trata simplemente de una dirección sin ninguna posición fija).

Una vez establecido el vector de proyección  $\mathbf{V}_p$  en coordenadas de visualización, todos los puntos de la escena se transfieren al plano de visualización según una serie de líneas que son paralelas a este vector. La Figura 7.37 ilustra la proyección paralela oblicua de un punto en el espacio sobre el plano de visualización. Podemos denotar las componentes del vector de proyección con respecto al sistema de coordenadas de visualización como  $\mathbf{V}_p = (V_{px}, V_{py}, V_{pz})$ , donde  $V_{py}/V_{px} = \tan \phi$ . Entonces, comparando los triángulos similares de la Figura 7.37, tendremos:

$$\frac{x_p - x}{z_{vp} - z} = \frac{V_{px}}{V_{pz}}$$

$$\frac{y_p - y}{z_{vp} - z} = \frac{V_{py}}{V_{pz}}$$

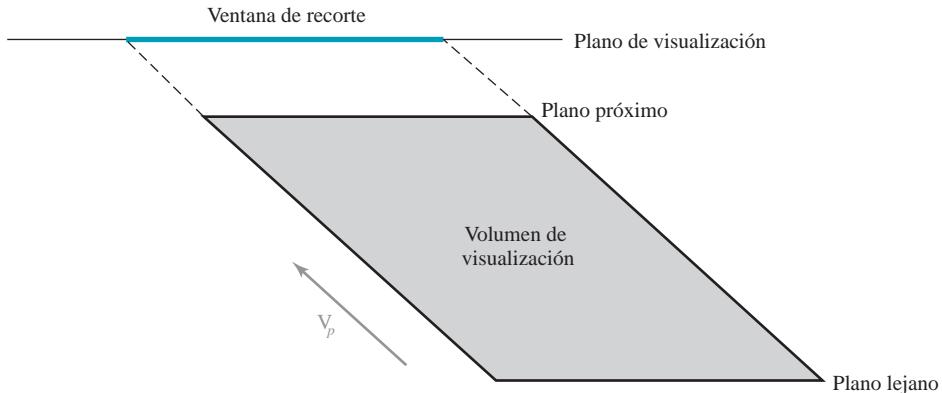
Y podemos escribir el equivalente de las Ecuaciones 7.11 de proyección paralela oblicua en términos del vector de proyección, como:

$$\begin{aligned} x_p &= x + (z_{vp} - z) \frac{V_{px}}{V_{pz}} \\ y_p &= y + (z_{vp} - z) \frac{V_{py}}{V_{pz}} \end{aligned} \tag{7.12}$$

Las coordenadas de proyección paralela oblicua de la Ecuación 7.12 se reducen a las coordenadas de proyección ortogonal de la Ecuación 7.6 cuando  $V_{px} = V_{py} = 0$ .

### Ventana de recorte y volumen de visualización de proyección paralela oblicua

El volumen de visualización para una proyección paralela oblicua se define utilizando los mismos procedimientos que en la proyección ortogonal. Seleccionamos una ventana de recorte sobre el plano de visualización con las coordenadas  $(xw_{\min}, yw_{\min})$  y  $(xw_{\max}, yw_{\max})$ , que especifican las esquinas inferior izquierda y



**FIGURA 7.38.** Vista superior de un volumen de visualización finito para una proyección paralela oblicua en la dirección del vector  $V_p$ .

superior derecha del rectángulo de recorte. Las partes superior, inferior y laterales del volumen de visualización se definen entonces mediante la dirección de proyección y los lados de la ventana de recorte. Además, podemos limitar la extensión del volumen de visualización añadiendo un plano próximo y otro lejano, como en la Figura 7.38. El volumen de visualización finito para proyección paralela oblicua es un paralelepípedo oblicuo.

Las proyecciones paralelas oblicuas pueden verse afectadas por los cambios en la posición del plano de visualización, dependiendo de cómo haya que especificar la dirección de proyección. En algunos sistemas, la dirección de proyección paralela oblicua es paralela a la línea que conecta el punto de referencia con el centro de la ventana de recorte. Por tanto, si se mueve la posición del plano de visualización o de la ventana de recorte sin ajustar el punto de referencia, cambiará la forma del volumen de visualización.

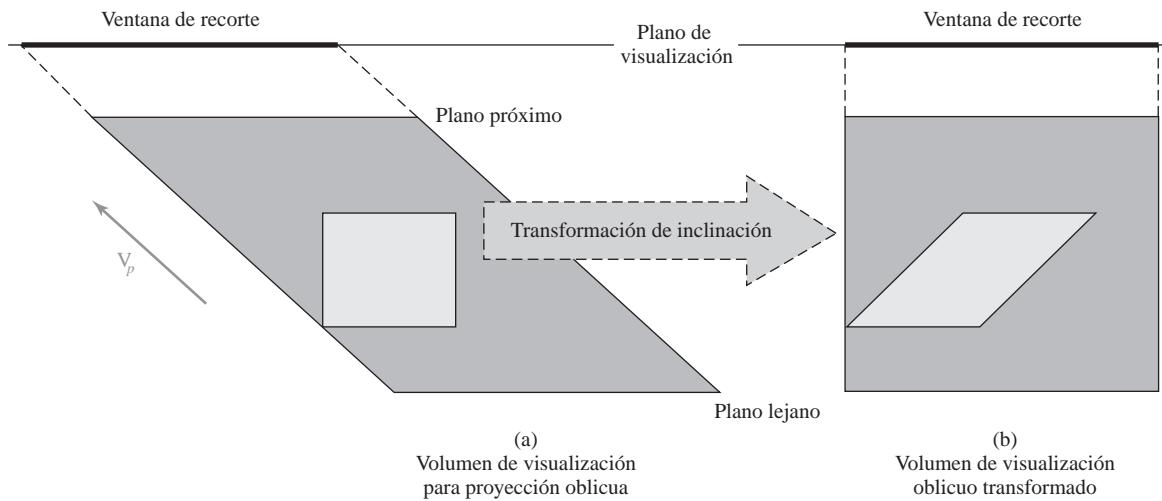
### Matriz de transformación para proyección paralela oblicua

Utilizando los parámetros del vector de proyección de las Ecuaciones 7.12, podemos expresar los elementos de la matriz de transformación para una proyección paralela oblicua como:

$$\mathbf{M}_{\text{oblicua}} = \begin{bmatrix} 1 & 0 & -\frac{V_{px}}{V_{pz}} & z_{vp} \frac{V_{px}}{V_{pz}} \\ 0 & 1 & -\frac{V_{py}}{V_{pz}} & z_{vp} \frac{V_{py}}{V_{pz}} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (7.13)$$

Esta matriz desplaza los valores de las coordenadas  $x$  e  $y$  según una cantidad proporcional a la distancia con respecto al plano de visualización, que se encuentra en la posición  $z_{vp}$  sobre el eje  $z_{\text{view}}$ . Los valores  $z$  de los puntos del espacio no se ven modificados. Si  $V_{px} = V_{py} = 0$ , tendremos una proyección ortogonal y la Matriz 7.13 se reduce a la matriz identidad.

Para una proyección paralela oblicua general, la Matriz 7.13 representa una transformación de inclinación según el eje  $z$ . Todos los puntos dentro del volumen de proyección oblicuo se verán inclinados según una cantidad proporcional a su distancia con respecto al plano de visualización. El efecto obtenido es la inclinación del volumen de visualización oblicuo para obtener un paralelepípedo rectangular, como se ilustra en la Figura 7.39. Así, los puntos dentro del



**FIGURA 7.39.** Vista superior de una transformación de proyección paralela oblicua. El volumen de visualización para proyección oblicua se convierte en un paralelepípedo rectangular y los objetos del volumen de visualización, como por ejemplo el bloque verde, se asignan a coordenadas de proyección ortogonal.

volumen de visualización se proyectarán para obtener las coordenadas de proyección ortogonal mediante la transformación de proyección paralela oblicua.

### Transformación de normalización para una proyección paralela oblicua

Puesto que las ecuaciones de proyección paralela oblicua convierten las descripciones de los objetos a coordenadas ortogonales, podemos aplicar los procedimientos de normalización después de esta transformación. El volumen de visualización oblicuo habrá sido convertido en un paralelepípedo rectangular, por lo que utilizaremos los mismos procedimientos que en la Sección 7.6.

Siguiendo el ejemplo de normalización de la Sección 7.6, volvemos a realizar el mapeo sobre el cubo normalizado simétrico, dentro de un sistema de coordenadas que cumpla la regla de la mano izquierda. Así, la transformación completa desde coordenadas de visualización a coordenadas normalizadas para una proyección paralela oblicua será:

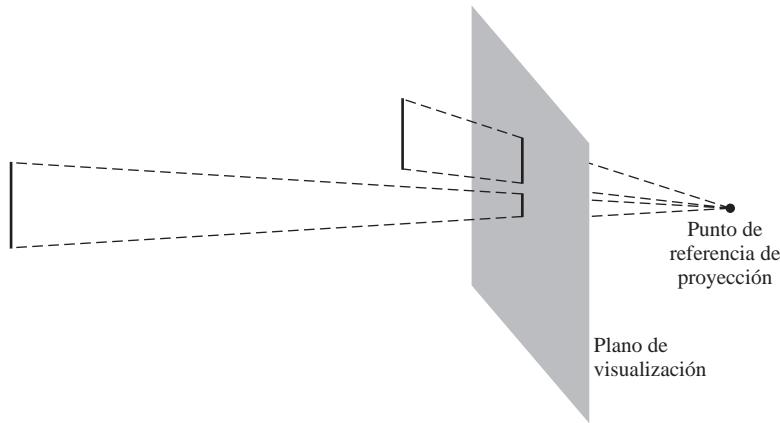
$$\mathbf{M}_{\text{oblique,norm}} = \mathbf{M}_{\text{ortho,norm}} \cdot \mathbf{M}_{\text{oblique}} \quad (7.14)$$

La transformación  $\mathbf{M}_{\text{oblique}}$  es la Matriz 7.13, que convierte la descripción de la escena a coordenadas de proyección ortogonal, y la transformación  $\mathbf{M}_{\text{ortho,norm}}$  es la Matriz 7.7, que mapea el contenido del volumen de visualización ortogonal sobre el cubo de normalización simétrico.

Para completar las transformaciones de visualización (con la excepción del mapeo a coordenadas de pantalla del visor), concatenamos la Matriz 7.14 a la izquierda de la transformación  $\mathbf{M}_{WC,VC}$  de la Sección 7.4. Después podemos aplicar las rutinas de recorte al volumen de visualización normalizado, tras lo cual determinaremos los objetos visibles, aplicaremos los procedimientos de representación de superficies y realizaremos la transformación de visor.

## 7.8 PROYECCIONES EN PERSPECTIVA

Aunque una vista en proyección paralela de una escena es fácil de generar y preserva las proporciones relativas de los objetos, no proporciona una representación realista. Para simular la imagen de una cámara, tene-



**FIGURA 7.40.** Proyección en perspectiva de dos segmentos lineales de igual longitud situados a diferentes distancias del plano de visualización.

mos que considerar que los rayos de luz reflejados en los objetos de la escena describen una serie de trayectorias convergentes hasta el plano de la película de la cámara. Podemos aproximar este efecto de óptica geométrica proyectando los objetos hasta el plano de visualización según una serie de trayectorias convergentes dirigidas al denominado **punto de referencia de proyección** (o **centro de proyección**). Los objetos se muestran entonces con un efecto de acortamiento y la proyección de los objetos distantes es más pequeña que la de los objetos del mismo tamaño que se encuentren más próximos al plano de visualización (Figura 7.40).

### Transformación de coordenadas para la proyección en perspectiva

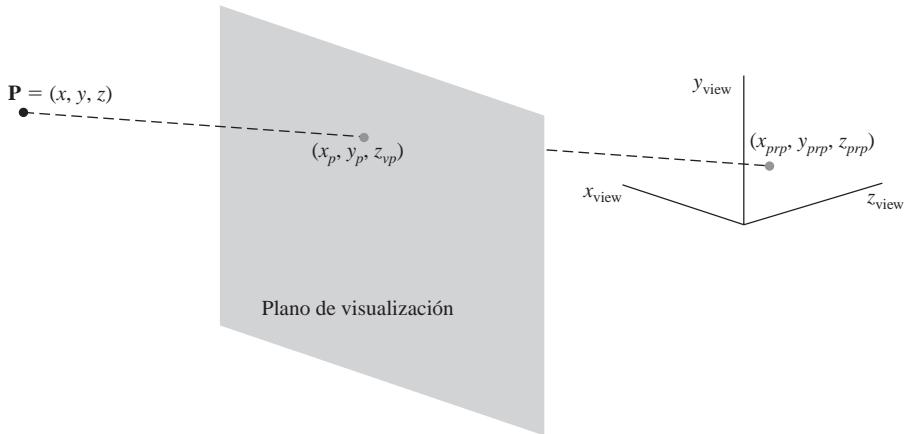
En ocasiones, podemos seleccionar el punto de referencia de proyección como otro de los parámetros de visualización dentro del paquete gráfico, pero algunos sistemas sitúan este punto de convergencia en una posición fija, como por ejemplo en el punto de vista. La Figura 7.41 muestra la trayectoria de proyección de un punto en el espacio  $(x, y, z)$  hacia un punto de referencia de proyección cualquiera situado en  $(x_{prp}, y_{prp}, z_{prp})$ . La línea de proyección interseca el plano de visualización en el punto  $(x_p, y_p, z_{vp})$ , donde  $z_{vp}$  es alguna posición seleccionada para el plano de visualización sobre el eje  $z_{view}$ . Podemos escribir las ecuaciones que describen las coordenadas a lo largo de esta línea de proyección en perspectiva utilizando la forma paramétrica:

$$\begin{aligned} x' &= x - (x - x_{prp})u \\ y' &= y - (y - y_{prp})u \quad 0 \leq u \leq 1 \\ z' &= z - (z - z_{prp})u \end{aligned} \tag{7.15}$$

El punto  $(x', y', z')$  representa cualquier punto situado a lo largo de la línea de proyección. Cuando  $u = 0$ , estaremos en la posición  $\mathbf{P} = (x, y, z)$ . En el otro extremo de la línea,  $u = 1$  y tendremos las coordenadas del punto de referencia de proyección  $(x_{prp}, y_{prp}, z_{prp})$ . Sobre el plano de visualización,  $z' = z_{vp}$  y podemos despejar el parámetro  $u$  en la ecuación  $z'$  en esta posición de la línea de proyección:

$$u = \frac{z_{vp} - z}{z_{prp} - z} \tag{7.16}$$

Sustituyendo este valor de  $u$  en las ecuaciones correspondientes a  $x'$  e  $y'$ , obtenemos las ecuaciones generales de transformación de perspectiva:



**FIGURA 7.41.** Proyección en perspectiva de un punto  $\mathbf{P}$  con coordenadas  $(x, y, z)$  hacia un punto de referencia de proyección seleccionado. La intersección con el plano de visualización es  $(x_p, y_p, z_{vp})$ .

$$x_p = x \left( \frac{z_{prp} - z_{vp}}{z_{prp} - z} \right) + x_{prp} \left( \frac{z_{vp} - z}{z_{prp} - z} \right) \quad (7.17)$$

$$y_p = y \left( \frac{z_{prp} - z_{vp}}{z_{prp} - z} \right) + y_{prp} \left( \frac{z_{vp} - z}{z_{prp} - z} \right)$$

Los cálculos para un mapeo en perspectiva son más complejos que las ecuaciones de proyección paralela, ya que los denominadores de los cálculos de perspectiva 7.17 están en función de la coordenada  $z$  de cada punto en el espacio. Por tanto, necesitamos formular los procedimientos de proyección en perspectiva de manera un poco distinta, con el fin de concatenar este mapeo con las otras transformaciones de visualización. Pero echemos primero un vistazo a algunas de las propiedades de las Ecuaciones 7.17.

### Ecuaciones de la proyección en perspectiva: casos especiales

A menudo se imponen diversas restricciones sobre los parámetros de la proyección en perspectiva. Dependiendo de cada paquete gráfico concreto, el posicionamiento del punto de referencia de proyección en el plano de visualización puede no ser completamente arbitrario.

Para simplificar los cálculos de la perspectiva, el punto de referencia de proyección puede limitarse a las posiciones situadas a lo largo del eje  $z_{view}$ , con lo que:

$$(1) \quad x_{prp} = y_{prp} = 0:$$

$$x_p = x \left( \frac{z_{prp} - z_{vp}}{z_{prp} - z} \right), \quad y_p = y \left( \frac{z_{prp} - z_{vp}}{z_{prp} - z} \right) \quad (7.18)$$

Y en ocasiones el punto de referencia de proyección está fijo en el origen de coordenadas, de modo que

$$(2) \quad (x_{prp}, y_{prp}, z_{prp}) = (0, 0, 0):$$

$$x_p = x \left( \frac{z_{vp}}{z} \right), \quad y_p = y \left( \frac{z_{vp}}{z} \right) \quad (7.19)$$

Si el plano de visualización es el plano  $uv$  y no hay restricciones en lo que respecta a la colocación del punto de referencia de proyección, tendremos,

$$(3) \quad z_{vp} = 0:$$

$$x_p = x \left( \frac{z_{prp}}{z_{prp} - z} \right) - x_{prp} \left( \frac{z}{z_{prp} - z} \right) \quad (7.20)$$

$$y_p = y \left( \frac{z_{prp}}{z_{prp} - z} \right) - y_{prp} \left( \frac{z}{z_{prp} - z} \right)$$

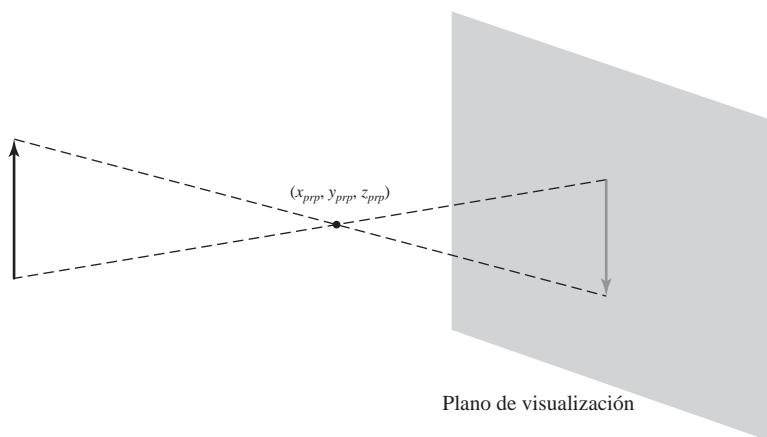
Con el plano  $uv$  como plano de visualización y situando el punto de referencia de proyección en el eje  $z_{view}$ , las ecuaciones de la proyección en perspectiva son:

$$(4) \quad x_{prp} = y_{prp} = z_{vp} = 0:$$

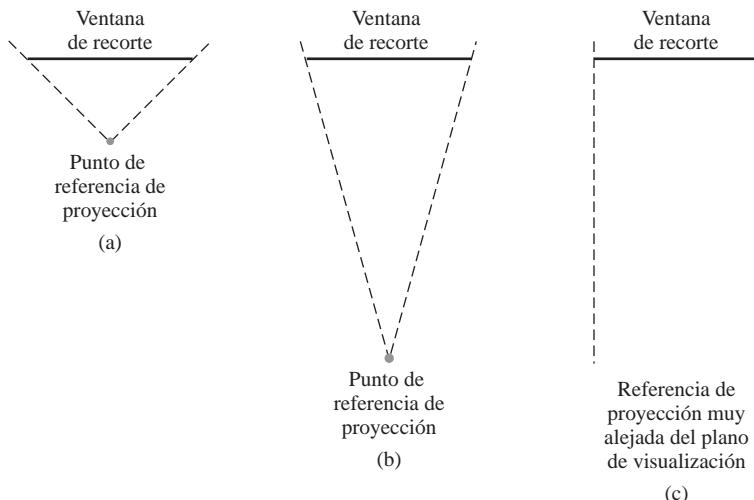
$$x_p = x \left( \frac{z_{prp}}{z_{prp} - z} \right), \quad y_p = y \left( \frac{z_{prp}}{z_{prp} - z} \right) \quad (7.21)$$

Por supuesto, no podemos poner el punto de referencia de proyección en el plano de visualización. En ese caso, toda la escena se proyectaría sobre un único punto. El plano de visualización suele situarse entre el punto de referencia de proyección y la escena, pero en general podríamos colocarlo en cualquier lugar excepto en el punto de proyección. Si el punto de referencia de proyección está situado entre el plano de visualización y la escena, los objetos se verán invertidos en el plano de visualización (Figura 7.42). Con la escena situada entre el plano de visualización y el punto de proyección, los objetos simplemente se agrandan al ser proyectados hacia afuera desde el punto de referencia hacia el plano de visualización.

Los efectos de perspectiva también dependen de la distancia entre el punto de referencia de proyección y el plano de visualización, como se ilustra en la Figura 7.43. Si el punto de referencia de proyección está próximo al plano de visualización, los efectos de perspectiva se enfatizan, es decir, los objetos más próximos parecen mucho mayores que los objetos más distantes del mismo tamaño. De forma similar, cuando alejamos el punto de referencia de proyección del plano de visualización, la diferencia de tamaño entre los objetos próximos y lejanos se reduce. Cuando el punto de referencia de proyección está muy lejos del plano de visualización, la proyección en perspectiva se aproxima a la proyección paralela.



**FIGURA 7.42.** La proyección en perspectiva de un objeto se invierte cuando el punto de referencia de proyección está situado entre el objeto y el plano de visualización.



**FIGURA 7.43.** Cambio de los efectos de la perspectiva al alejar el punto de referencia de proyección del plano de visualización.

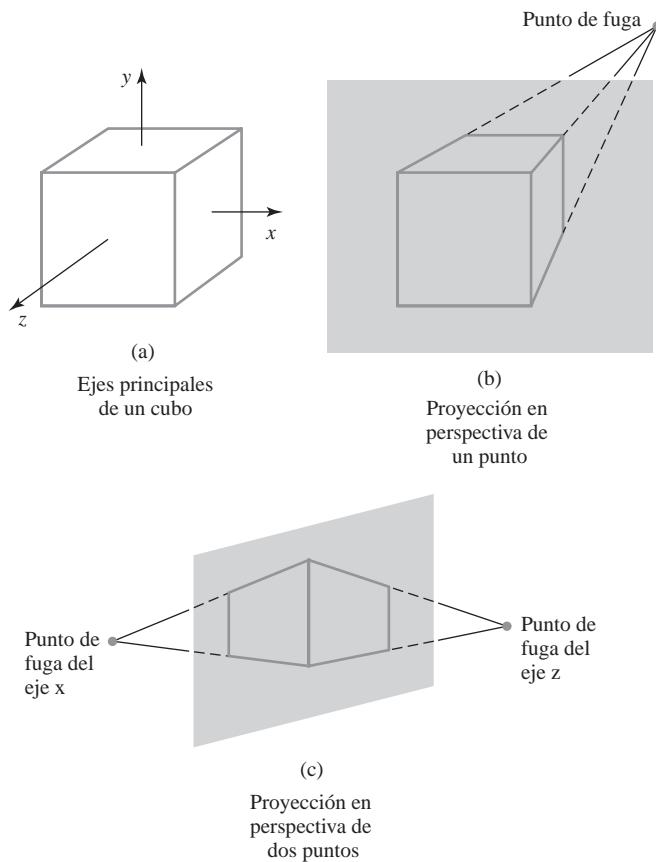
### Puntos de fuga para las proyecciones en perspectiva

Cuando se proyecta una escena sobre un plano de visualización utilizando mapeo en perspectiva, las líneas paralelas al plano de visualización se proyectan como líneas paralelas. Pero todas las demás líneas paralelas de la escena que no sean paralelas al plano de visualización se proyectarán como líneas convergentes, como podemos ver en la Figura 7.43. El punto en el que un conjunto de líneas paralelas proyectadas parecen converger se denomina **punto de fuga**. Cada conjunto de líneas paralelas proyectadas tiene un punto de fuga propio.

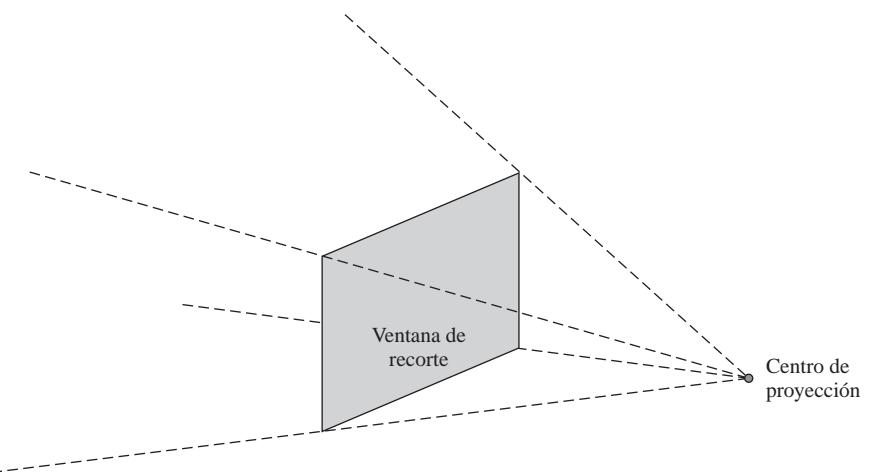
Para un conjunto de líneas que son paralelas a uno de los ejes principales de un objeto, el punto de fuga se denomina **punto de fuga principal**. Podemos controlar el número de puntos de fuga principales (uno, dos o tres) con la orientación de plano de proyección, y las proyecciones en perspectiva se clasifican por ello como proyecciones de un punto, de dos puntos o de tres puntos. El número de puntos de fuga principales en una proyección es igual al número de ejes principales que intersectan al plano de visualización. La Figura 7.44 ilustra la apariencia de sendas proyecciones en perspectiva de uno y dos puntos para un cubo. En la vista proyectada (b), el plano de visualización se alinea en paralelo al plano  $xy$  del objeto, de modo que sólo se intersecta el eje  $z$  del objeto. Esta orientación produce una proyección en perspectiva de un punto, estando el punto de fuga situado en el eje  $z$ . Para la vista mostrada en (c), el plano de proyección intersecta tanto al eje  $x$  como al eje  $z$ , pero no al eje  $y$ . La proyección en perspectiva de dos puntos resultante contiene puntos de fuga tanto para el eje  $x$  como para el eje  $z$ . El realismo no se incrementa mucho en las proyecciones en perspectiva de tres puntos si las comparamos con las de dos puntos, por lo que las de tres puntos no se utilizan tan a menudo en los diagramas de arquitectura e ingeniería.

### Volumen de visualización para proyección en perspectiva

De nuevo, creamos un volumen de visualización especificando la posición de una ventana de recorte rectangular sobre el plano de visualización. Pero ahora, los planos de contorno para el volumen de visualización, no son paralelos, porque las líneas de proyección no son paralelas. La parte superior, la inferior y las laterales del volumen de visualización son planos que pasan por los lados de la ventana y por el punto de referencia de proyección. Esto forma un volumen de visualización que es una pirámide rectangular infinita con su vértice situado en el centro de proyección (Figura 7.45). Todos los objetos situados fuera de esta pirámide serán eliminados por las rutinas de recorte. Un volumen de visualización para proyección en perspectiva se suele denominar **pirámide de visión**, porque se aproxima al *cono de visión* de nuestros ojos o de una cámara. La imagen mostrada de una escena incluirá únicamente aquellos objetos que estén situados dentro de la pirámide, exac-



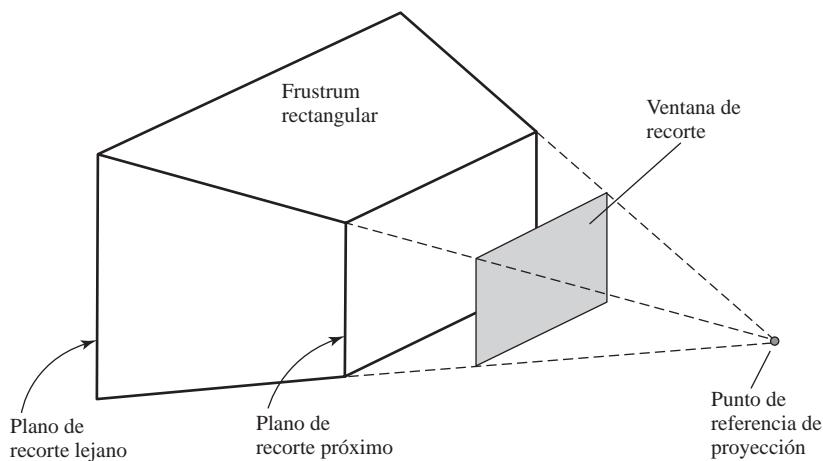
**FIGURA 7.44.** Puntos de fuga principales para las proyecciones en perspectiva de un cubo. Cuando el cubo de la figura (a) se proyecta sobre un plano de visualización que sólo intersecta con el eje  $z$ , se genera un único punto de fuga en la dirección  $z$  (b). Cuando el cubo se proyecta sobre un plano de visualización que intersecta con los ejes  $z$  y  $x$ , aparecen dos puntos de fuga (c).



**FIGURA 7.45.** Una pirámide de visión infinita para una proyección en perspectiva.

tamente de la misma forma que nosotros no podemos ver los objetos que están más allá de nuestra visión periférica y que caen fuera del cono de visión.

Añadiendo planos de recorte próximo y lejano que sean perpendiculares al eje  $z_{\text{view}}$  (y paralelos al plano de visualización), cortamos partes del volumen de visualización infinito con el fin de formar una pirámide truncada o **frustum**. La Figura 7.46 ilustra la forma de un volumen de visualización finito para proyección en



**FIGURA 7.46.** Frustrum de proyección en perspectiva con el plano de visualización «delante» del plano de recorte próximo.

perspectiva, con un plano de visualización que está colocado entre el plano de recorte próximo y el punto de referencia de proyección. Algunas veces, los paquetes gráficos obligan a definir los planos próximo y lejano, mientras que en otros paquetes son opcionales.

Usualmente, los planos de recorte próximo y lejano se encuentran a un mismo lado del punto de referencia de proyección, estando el plano lejano más lejos del punto de proyección que el plano próximo, según la dirección de visualización. Y también, al igual que en una proyección paralela, podemos utilizar los planos próximo y lejano simplemente para limitar la escena que hay que visualizar. Pero con una proyección en perspectiva también podemos usar el plano de recorte próximo con el fin de eliminar de la escena los objetos de gran tamaño que estén muy cerca del plano de visualización y que al proyectarse tuvieran una forma irreconocible en la ventana de recorte. De manera similar, el plano de recorte lejano puede emplearse para eliminar los objetos muy alejados del punto de referencia de proyección y que se proyectarían para formar pequeños puntitos sobre el plano de visualización. Algunos sistemas restringen la colocación del plano de visualización en relación con los planos próximo y lejano, mientras que otros sistemas permiten situarlo en cualquier punto excepto en la posición del punto de referencia de proyección. Si el plano de visualización está «detrás» del punto de referencia de proyección, los objetos se verán invertidos, como se muestra en la Figura 7.42.

### Matriz de transformación para la proyección en perspectiva

A diferencia de una proyección paralela, no podemos utilizar directamente los coeficientes de las coordenadas  $x$  e  $y$  en las Ecuaciones 7.17 para determinar los elementos de la matriz de proyección en perspectiva, porque los denominadores de los coeficientes son función de la coordenada  $z$ . Pero podemos emplear una representación en coordenadas homogéneas tridimensionales para expresar las ecuaciones de la proyección en perspectiva en la forma:

$$x_p = \frac{x_h}{h}, \quad y_p = \frac{y_h}{h} \quad (7.22)$$

donde el parámetro homogéneo tiene el valor:

$$h = z_{ppr} - z \quad (7.23)$$

Los numeradores en 7.22 son iguales que en las Ecuaciones 7.17:

$$\begin{aligned} x_h &= x(z_{ppr} - z_{vp}) + x_{ppr}(z_{vp} - z) \\ y_h &= y(z_{ppr} - z_{vp}) + y_{ppr}(z_{vp} - z) \end{aligned} \quad (7.24)$$

Así, podemos formar una matriz de transformación para convertir una posición en el espacio a coordenadas homogéneas de modo que la matriz sólo contenga los parámetros de la proyección en perspectiva y no valores de coordenadas. La transformación para la proyección en perspectiva de un punto definido en coordenadas de visualización se realiza entonces en dos pasos. En primer lugar, calculamos las coordenadas homogéneas utilizando la matriz de transformación de perspectiva:

$$\mathbf{P}_h = \mathbf{M}_{\text{pers}} \cdot \mathbf{P} \quad (7.25)$$

donde  $\mathbf{P}_h$  es la representación en forma de matriz columna del punto de coordenadas homogéneas ( $x_h, y_h, z_h, h$ ) y  $\mathbf{P}$  es la representación en forma de matriz columna de la posición de coordenadas ( $x, y, z, 1$ ). (En realidad, la matriz de perspectiva se concatenaría con las otras matrices de transformación de visualización y luego la matriz compuesta se aplicaría a la descripción en coordenadas universales de una escena con el fin de obtener las coordenadas homogéneas). En segundo lugar, después de haber aplicado otros procesos, como la transformación de normalización y las rutinas de recorte, las coordenadas homogéneas se dividen por el parámetro  $h$  para obtener las verdaderas posiciones de coordenadas transformadas.

Resulta sencillo definir los elementos de la matriz para obtener los valores  $x_h$  e  $y_h$  de coordenadas homogéneas de 7.24, pero también debemos estructurar la matriz para preservar la información de profundidad (valor  $z$ ). En caso contrario, las coordenadas  $z$  se verían distorsionadas por el parámetro de división homogéneo  $h$ . Podemos hacer esto definiendo los elementos de la matriz para la transformación  $z$  de modo que se normalicen las coordenadas  $z_p$  de la proyección en perspectiva. Hay varias formas de elegir los elementos de la matriz para generar las coordenadas homogéneas 7.24 y el valor  $z_p$  normalizado para una posición del espacio ( $x, y, z$ ). La siguiente matriz representa una de las posibles maneras de formular una matriz de proyección de perspectiva:

$$\mathbf{M}_{\text{pers}} = \begin{bmatrix} z_{\text{prp}} - z_{\text{vp}} & 0 & -x_{\text{prp}} & x_{\text{prp}}z_{\text{prp}} \\ 0 & z_{\text{prp}} - z_{\text{vp}} & -y_{\text{prp}} & y_{\text{prp}}z_{\text{prp}} \\ 0 & 0 & s_z & t_z \\ 0 & 0 & -1 & z_{\text{prp}} \end{bmatrix} \quad (7.26)$$

Los parámetros  $s_z$  y  $t_z$  son los factores de cambio de escala y de traslación para la normalización de los valores proyectados de las coordenadas  $z$ . Los valores específicos de  $s_z$  y  $t_z$  dependerán del rango de normalización que seleccionemos.

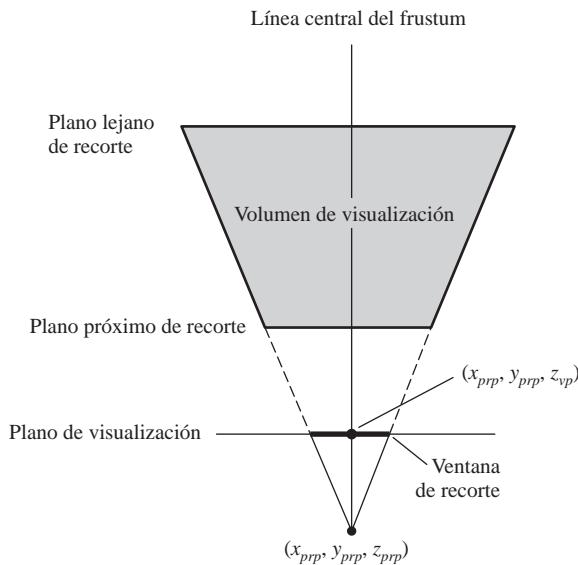
La Matriz 7.26 convierte la descripción de una escena en coordenadas homogéneas de proyección paralela. Sin embargo, el frustum de visualización puede tener cualquier orientación, por lo que estas coordenadas transformadas podrían corresponderse con una proyección paralela oblicua. Esto ocurre si el frustum de visualización no es simétrico. En cambio, si el frustum de visualización para la proyección en perspectiva es simétrico, las coordenadas de proyección paralela resultantes se corresponden con una proyección ortogonal. Analicemos por separado estas dos posibilidades.

### Frustum de proyección en perspectiva simétrico

La línea que parte del punto de referencia de proyección y que pasa por el centro de la ventana de recorte y del volumen de visualización es la línea central del frustum de proyección en perspectiva. Si esta línea central es perpendicular al plano de visualización, tendremos un **frustum simétrico** (con respecto a su línea central) como en la Figura 7.47.

Puesto que la línea central del frustum intersecta con el plano de visualización en el punto ( $x_{\text{prp}}, y_{\text{prp}}, z_{\text{vp}}$ ), podemos expresar las posiciones de las esquinas de la ventana de recorte en términos de las dimensiones de la ventana:

$$xw_{\min} = x_{\text{prp}} - \frac{\text{anchura}}{2}, \quad xw_{\max} = x_{\text{prp}} + \frac{\text{anchura}}{2}$$



**FIGURA 7.47.** Frustrum de visualización para proyección en perspectiva simétrica, con el plano de visualización situado entre el punto de referencia de proyección y el plano de recorte próximo. Este frustrum es simétrico con respecto a la línea central cuando se mira desde arriba, desde abajo o desde cualquiera de los lados.

$$yw_{\min} = y_{prp} - \frac{\text{altura}}{2}, \quad yw_{\max} = y_{prp} + \frac{\text{altura}}{2}$$

Por tanto, podríamos especificar una vista de proyección en perspectiva simétrica de una escena utilizando la anchura y la altura de la ventana de recorte en lugar de las coordenadas de la ventana. Esto especifica de manera no ambigua la posición de la ventana de recorte, ya que ésta es simétrica con respecto a las coordenadas  $x$  e  $y$  del punto de referencia de proyección.

Otra forma de especificar una proyección en perspectiva simétrica consiste en utilizar parámetros que aproximen las propiedades del objetivo de una cámara. Las fotografías se generan mediante una proyección en perspectiva simétrica de una escena sobre el plano de la película. Los rayos luminosos reflejados por los objetos de una escena se proyectan sobre el plano de la película desde el «cono de visión» de la cámara. Este cono de visión puede especificarse mediante un **ángulo de campo visual**, que es una medida del tamaño del objetivo de la cámara. Un gran ángulo de campo visual, por ejemplo, se corresponderá con un objetivo de gran angular. En infografía, el cono de visión se aproxima mediante un frustum simétrico y podemos utilizar un ángulo del campo de visión para especificar el tamaño angular del frustum. Normalmente, el ángulo del campo de visión será el ángulo existente entre el plano superior de recorte y el plano inferior de recorte del frustum, como se muestra en la Figura 7.48.

Para un punto de referencia de proyección y para una posición del plano de visualización dados, el ángulo del campo de visión determina la altura de la ventana de recorte (Figura 7.49), pero no la anchura. Necesitamos por tanto un parámetro adicional, pero no la anchura. Necesitamos por tanto un parámetro adicional para definir completamente las dimensiones de la ventana de recorte, y ese segundo parámetro podría ser la anchura de la ventana o la relación de aspecto (anchura/altura) de la ventana de recorte. Fijándonos en los triángulos rectos del diagrama de la Figura 7.49, vemos que:

$$\tan\left(\frac{\theta}{2}\right) = \frac{\text{altura}/2}{z_{prp} - z_{vp}} \quad (7.27)$$

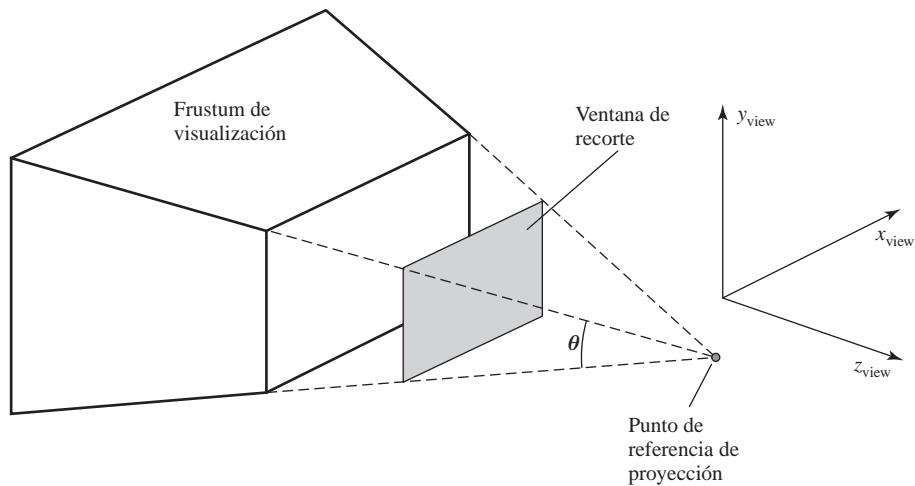
de modo que la altura de la ventana de recorte puede calcularse como:

$$\text{altura} = 2(z_{prp} - z_{vp}) \tan\left(\frac{\theta}{2}\right) \quad (7.28)$$

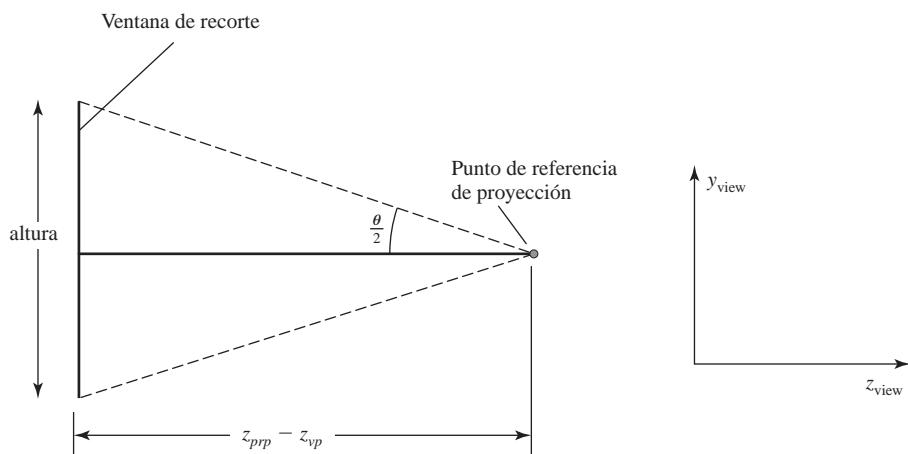
Por tanto, los elementos diagonales con el valor  $z_{ppr} + z_{vp}$  en la Matriz 7.26 pueden sustituirse por cualquiera de las dos siguientes expresiones:

$$\begin{aligned} z_{ppr} - z_{vp} &= \frac{\text{altura}}{2} \cot\left(\frac{\theta}{2}\right) \\ &= \frac{\text{anchura} \cdot \cot(\theta/2)}{2 \cdot \text{aspecto}} \end{aligned} \quad (7.29)$$

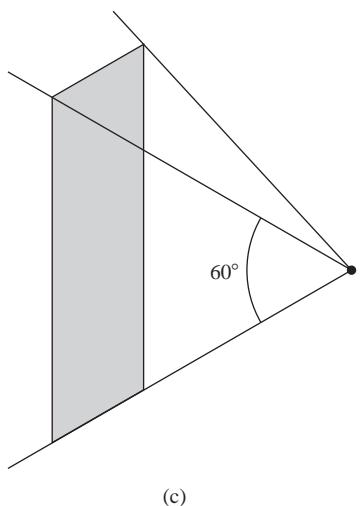
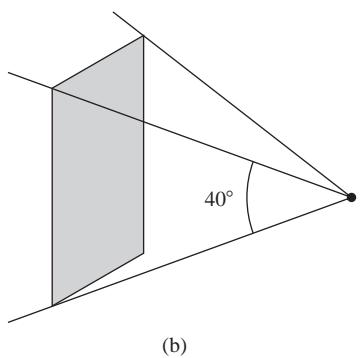
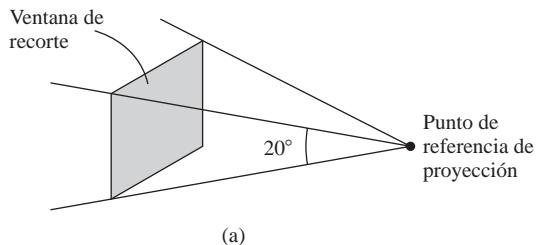
En algunas bibliotecas gráficas, se utilizan posiciones fijas para el plano de visualización y el punto de referencia de proyección, por lo que una proyección en perspectiva simétrica quedará completamente especificada mediante el ángulo del campo visual, la relación de aspecto de la ventana de recorte y las distancias desde la posición de visualización hasta los planos de recorte próximo y lejano. Normalmente, se aplica la misma relación de aspecto a la especificación del visor.



**FIGURA 7.48.** Ángulo del campo de visión  $\theta$  para un volumen de visualización simétrico de proyección en perspectiva, con la ventana de recorte situada entre el plano próximo de recorte y el punto de referencia de proyección.



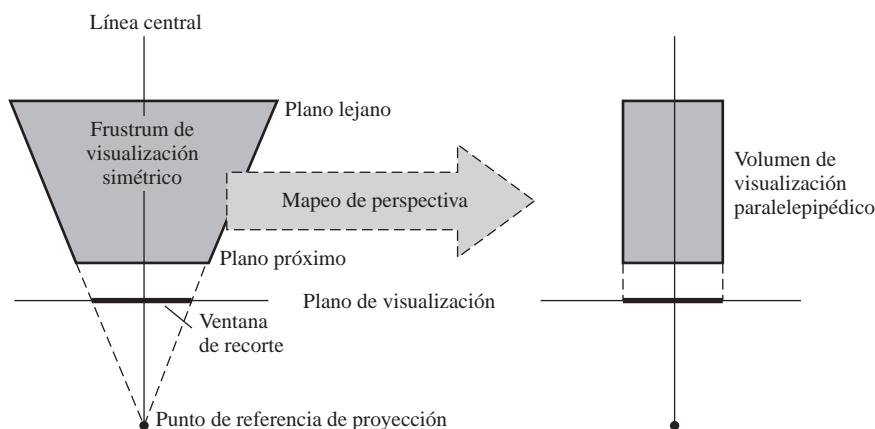
**FIGURA 7.49.** Relación entre el ángulo del campo visual  $\theta$ , la altura de la ventana de recorte y la distancia entre el punto de referencia de proyección y el plano de visualización.



**FIGURA 7.50.** Al incrementar el ángulo del campo de visión, se incrementa la altura de la ventana de recorte y también los efectos de acortamiento derivados de la proyección en perspectiva.

Si el ángulo del campo de visión se reduce en una aplicación concreta, los efectos de acortamiento derivados de la predicción en perspectiva también se reducen. Esto sería comparable a alejar el punto de referencia de proyección del plano de visualización. Asimismo, reducir el ángulo del campo de visión hace que disminuya la altura de la ventana de recorte, y esto proporciona un método para ampliar pequeñas regiones de una escena. De forma similar, un gran ángulo del campo de visión da como resultado una mayor altura de la ventana de recorte (la escena se empequeñece) y se incrementan los efectos de perspectiva, que es exactamente lo mismo que pasa cuando situamos el punto de referencia de proyección muy cerca del plano de visualización. La Figura 7.50 ilustra los efectos de diversos ángulos del campo de visión para una ventana de recorte de anchura fija.

Cuando el volumen de visualización para proyección en perspectiva es un frustum simétrico, la transformación en perspectiva mapea los puntos situados en el interior del frustum a una serie de coordenadas de pro-

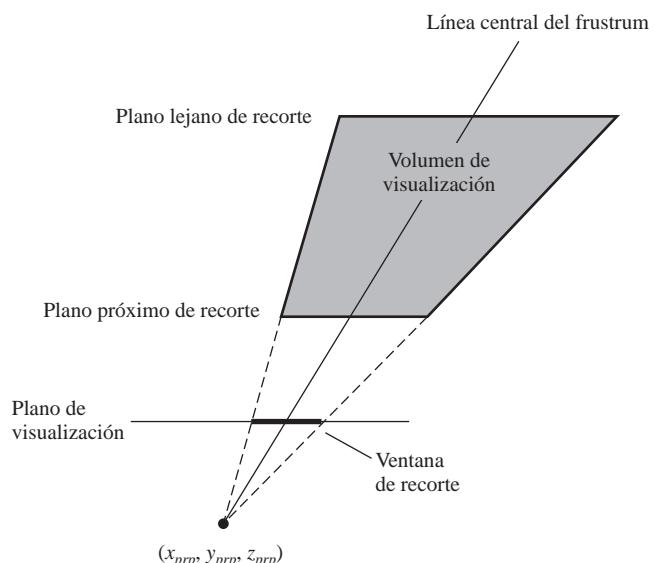


**FIGURA 7.51.** Un frustum de visualización simétrico se mapea a un paralelepípedo rectangular mediante la transformación de proyección en perspectiva.

yección ortogonal dentro de un paralelepípedo rectangular. La línea central del paralelepípedo coincide con la del frustum, puesto que esta línea ya es perpendicular al plano de visualización (Figura 7.51). Esto es consecuencia del hecho de que todos los puntos situados a lo largo de una línea de proyección dentro del frustum se mapean al mismo punto  $(x_p, y_p)$  del plano de visualización. Así, cada línea de proyección es convertida en la transformación de perspectiva en una línea perpendicular al plano de visualización y, por tanto, paralela a la línea central del frustum. Una vez convertido el frustum simétrico en un volumen de visualización de proyección ortogonal, podemos pasar a aplicar la transformación de normalización.

### Frustum de proyección en perspectiva oblicuo

Si la línea central de un volumen de visualización para proyección en perspectiva no es perpendicular al plano de visualización, lo que tendremos es un **frustum oblicuo**. La Figura 7.52 ilustra la apariencia general de un volumen de visualización oblicuo para proyección en perspectiva. En este caso, podemos transformar primero el volumen de visualización en un frustum simétrico y luego en un volumen de visualización normalizado.



**FIGURA 7.52.** Frustum oblicuo (cuando se lo ve desde al menos un lado o mediante una vista superior), con el plano de visualización situado entre el punto de referencia de proyección y el plano de recorte próximo.

Un volumen de visualización para proyección en perspectiva oblicuo puede convertirse en un frustum simétrico aplicando la matriz de inclinación del eje  $z$  dada en la Ecuación 5.115. Esta transformación desplaza todas las posiciones de cualquier plano que sea perpendicular al eje  $z$  según una cantidad proporcional a la distancia del plano con respecto a una posición de referencia especificada sobre el eje  $z$ . En este caso, la posición de referencia es  $z_{prp}$ , que es la coordenada  $z$  del punto de referencia de proyección, y tendremos que efectuar el desplazamiento según una cantidad que mueva el centro de la ventana de recorte a la posición  $(x_{prp}, y_{prp})$  sobre el plano de visualización. Puesto que la línea central del frustum pasa por el centro de la ventana de recorte, este desplazamiento ajustará la línea central de modo que quede en posición perpendicular al plano de visualización, como en la Figura 7.47.

Los cálculos para la transformación de inclinación, así como para las transformaciones de perspectiva y de normalización, se simplifican enormemente si hacemos que el punto de referencia de proyección sea el origen del sistema de coordenadas de visualización. Podemos hacer esto sin pérdida de generalidad efectuando una traslación de todas las coordenadas de la escena de modo que nuestro punto de referencia de proyección seleccionado quede situado sobre el origen de coordenadas. O bien, podemos desde el principio definir el sistema de referencia de coordenadas de visualización de modo que su origen se encuentre en el punto de proyección deseado para la escena. De hecho, algunas bibliotecas gráficas fijan el punto de referencia de proyección en el origen de coordenadas.

Tomando el punto de referencia de proyección como  $(x_{prp}, y_{prp}, z_{prp}) = (0, 0, 0)$ , obtenemos los elementos de la matriz de inclinación requerida:

$$\mathbf{M}_{z \text{ shear}} = \begin{bmatrix} 1 & 0 & sh_{zx} & 0 \\ 0 & 1 & sh_{zy} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (7.30)$$

También podemos simplificar los elementos de la matriz de proyección de perspectiva un poco más si situamos el plano de visualización en la posición del plano de recorte próximo. Y como lo que queremos ahora es desplazar el centro de la ventana de recorte a las coordenadas  $(0, 0)$  en el plano de visualización, tendremos que seleccionar los valores de los parámetros de inclinación de modo que,

$$\begin{bmatrix} 0 \\ 0 \\ z_{\text{near}} \\ 1 \end{bmatrix} = \mathbf{M}_{z \text{ shear}} \cdot \begin{bmatrix} \frac{xw_{\min} + xw_{\max}}{2} \\ \frac{yw_{\min} + yw_{\max}}{2} \\ z_{\text{near}} \\ 1 \end{bmatrix} \quad (7.31)$$

Por tanto, los parámetros de esta transformación de inclinación son:

$$\begin{aligned} sh_{zx} &= \frac{xw_{\min} + xw_{\max}}{2 z_{\text{near}}} \\ sh_{zy} &= -\frac{yw_{\min} + yw_{\max}}{2 z_{\text{near}}} \end{aligned} \quad (7.32)$$

De forma similar, teniendo el punto de referencia de proyección en el origen del sistema de coordenadas de visualización y tomando el plano próximo de recorte como plano de visualización la matriz de proyección de perspectiva 7.26 se simplifica, quedando,

$$\mathbf{M}_{\text{pers}} = \begin{bmatrix} -z_{\text{near}} & 0 & 0 & 0 \\ 0 & -z_{\text{near}} & 0 & 0 \\ 0 & 0 & s_z & t_z \\ 0 & 0 & -1 & 0 \end{bmatrix} \quad (7.33)$$

Las expresiones para los parámetros de cambio de escala y de translación de la coordenada  $z$  serán determinados por los requisitos de normalización.

Concatenando la matriz simplificada de proyección de perspectiva 7.33 con la matriz de inclinación 7.30, obtenemos la siguiente matriz de proyección de perspectiva oblicua, que podemos utilizar para convertir las coordenadas de una escena a coordenadas homogéneas de proyección ortogonal. El punto de referencia de proyección para esta transformación será el origen del sistema de coordenadas de visualización, mientras que el plano próximo de recorte será el plano de visualización.

$$\mathbf{M}_{\text{obliquepers}} = \mathbf{M}_{\text{pers}} \cdot \mathbf{M}_{z \text{ shear}}$$

$$= \begin{bmatrix} -z_{\text{near}} & 0 & \frac{xw_{\min} + xw_{\max}}{2} & 0 \\ 0 & -z_{\text{near}} & \frac{yw_{\min} + yw_{\max}}{2} & 0 \\ 0 & 0 & s_z & t_z \\ 0 & 0 & -1 & 0 \end{bmatrix} \quad (7.34)$$

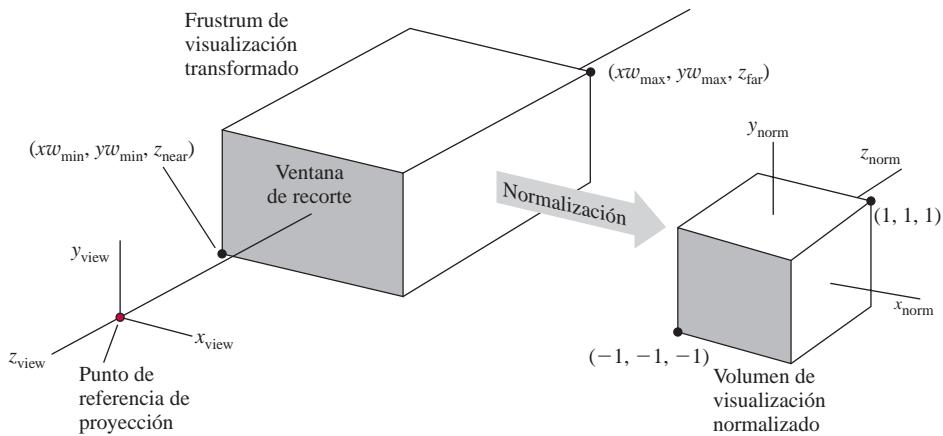
Aunque ya no disponemos de opciones para colocar arbitrariamente el punto de referencia de proyección y el plano de visualización, esta matriz proporciona un método eficiente para generar una vista de proyección en perspectiva de una escena, sin sacrificar un alto grado de flexibilidad.

Si seleccionamos las coordenadas de la ventana de recorte de modo que  $xw_{\max} = -xw_{\min}$  y  $yw_{\max} = -yw_{\min}$ , el frustum de visualización será simétrico y la Matriz 7.34 se reduce a la Matriz 7.33. Esto se debe a que el punto de referencia de proyección se encuentra ahora en el origen del sistema de coordenadas de visualización. También podríamos usar las Ecuaciones 7.29 con  $z_{\text{ppr}} = 0$  y  $z_{\text{vp}} = z_{\text{near}}$ , para especificar los dos primeros elementos diagonales de esta matriz en términos del ángulo del campo visual y de las dimensiones de la ventana de recorte.

### Coordenadas de transformación normalizadas para proyección en perspectiva

La Matriz 7.34 transforma las posiciones de los objetos en coordenadas de visualización, para obtener coordenadas homogéneas de proyección en perspectiva. Cuando dividimos las coordenadas homogéneas con el parámetro homogéneo  $h$ , obtenemos las coordenadas reales de proyección, que son coordenadas de proyección ortogonal. Así, esta proyección en perspectiva transforma todos los puntos situados dentro del frustum de visualización, obteniéndose posiciones situadas dentro de un volumen de visualización paralelepípedo rectangular. El paso final del proceso de transformación de perspectiva consiste en mapear este paralelepípedo sobre un *volumen de visualización normalizado*.

Seguiremos el mismo procedimiento de normalización que ya hemos empleado para una proyección paralela. El frustum de visualización transformado, que es un paralelepípedo rectangular, se mapea sobre un cubo normalizado simétrico dentro de un sistema de referencia que cumple con la regla de la mano izquierda (Figura 7.53). Ya hemos incluido los parámetros de normalización para las coordenadas  $z$  en la matriz de proyección en perspectiva 7.34, pero todavía necesitamos determinar los valores de estos parámetros cuando se realiza la transformación al cubo de normalización simétrico. Asimismo, necesitamos determinar los parámetros de transformación de normalización para las coordenadas  $x$  e  $y$ . Puesto que la línea central del volumen



**FIGURA 7.53.** Transformación de normalización que aplica un volumen de visualización para proyección en perspectiva transformado (paralelepípedo rectangular) sobre el cubo de normalización simétrico dentro de un sistema de referencia que cumple con la regla de la mano izquierda, utilizando el plano próximo de recorte como plano de visualización y situando el punto de referencia de proyección en el origen del sistema de coordenadas de visualización.

de visualización paralelepípedo rectangular es ahora el eje  $z_{\text{view}}$ , no hace falta ninguna traslación en las transformaciones de normalización de  $x$  e  $y$ . Lo único que nos hace falta son los parámetros de cambio de escala para  $x$  o  $y$  en relación con el origen de coordenadas. La matriz de cambio de escala para llevar a cabo la normalización  $xy$  es:

$$\mathbf{M}_{\text{xy scale}} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (7.35)$$

Concatenando la matriz de cambio de escala  $xy$  con la matriz 7.34, se obtiene la siguiente matriz de normalización para una transformación de proyección en perspectiva:

$$\begin{aligned} \mathbf{M}_{\text{normpers}} &= \mathbf{M}_{\text{xy scale}} \cdot \mathbf{M}_{\text{obliquepers}} \\ &= \begin{bmatrix} -z_{\text{near}}s_x & 0 & s_x \frac{xw_{\min} + xw_{\max}}{2} & 0 \\ 0 & -z_{\text{near}}s_y & s_y \frac{yw_{\min} +yw_{\max}}{2} & 0 \\ 0 & 0 & s_z & t_z \\ 0 & 0 & -1 & 0 \end{bmatrix} \end{aligned} \quad (7.36)$$

A partir de esta transformación, obtenemos las coordenadas homogéneas:

$$\begin{bmatrix} x_h \\ y_h \\ z_h \\ h \end{bmatrix} = \mathbf{M}_{\text{normpers}} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (7.37)$$

Y las coordenadas de proyección serán:

$$\begin{aligned}
 x_p &= \frac{x_h}{h} = \frac{-z_{\text{near}} s_x x + s_x (xw_{\min} + xw_{\max}) / 2}{-z} \\
 y_p &= \frac{y_h}{h} = \frac{-z_{\text{near}} s_y y + s_y (yw_{\min} + yw_{\max}) / 2}{-z} \\
 z_p &= \frac{z_h}{h} = \frac{s_z z + t_z}{-z}
 \end{aligned} \tag{7.38}$$

Para normalizar esta transformación de perspectiva, queremos que las coordenadas de proyección sean  $(x_p, y_p, z_p) = (-1, -1, -1)$  cuando las coordenadas de entrada sean  $(x, y, z) = (xw_{\min}, yw_{\min}, z_{\text{near}})$ , y que las coordenadas de proyección tengan el valor  $(x_p, y_p, z_p) = (1, 1, 1)$  cuando las coordenadas de entrada sean  $(x, y, z) = (xw_{\max}, yw_{\max}, z_{\text{far}})$ . Por tanto, cuando resolvemos las Ecuaciones 7.38 para calcular los parámetros de normalización utilizando estas condiciones, se obtiene:

$$\begin{aligned}
 s_x &= \frac{2}{xw_{\max} - xw_{\min}}, & s_y &= \frac{2}{yw_{\max} - yw_{\min}} \\
 s_z &= \frac{z_{\text{near}} + z_{\text{far}}}{z_{\text{near}} - z_{\text{far}}}, & t_z &= \frac{2z_{\text{near}}z_{\text{far}}}{z_{\text{near}} - z_{\text{far}}}
 \end{aligned} \tag{7.39}$$

Y los elementos de la matriz de transformación normalizada para una proyección en perspectiva general serán:

$$\mathbf{M}_{\text{normpers}} = \begin{bmatrix} \frac{-2z_{\text{near}}}{xw_{\max} - xw_{\min}} & 0 & \frac{xw_{\max} + xw_{\min}}{xw_{\max} - xw_{\min}} & 0 \\ 0 & \frac{-2z_{\text{near}}}{yw_{\max} - yw_{\min}} & \frac{yw_{\max} + yw_{\min}}{yw_{\max} - yw_{\min}} & 0 \\ 0 & 0 & \frac{z_{\text{near}} + z_{\text{far}}}{z_{\text{near}} - z_{\text{far}}} & -\frac{2z_{\text{near}}z_{\text{far}}}{z_{\text{near}} - z_{\text{far}}} \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{7.40}$$

Si el volumen de visualización para proyección en perspectiva hubiera sido especificado originalmente como un frustum simétrico, podríamos expresar los elementos de la transformación de perspectiva normalizada en términos del ángulo del campo visual y de las dimensiones de la ventana de recorte. Así, utilizando las Ecuaciones 7.29, con el punto de referencia de proyección en el origen y el plano de visualización coincidente con el plano próximo de recorte, tendremos:

$$\mathbf{M}_{\text{normsympers}} = \begin{bmatrix} \frac{\cot(\frac{\theta}{2})}{\text{aspecto}} & 0 & 0 & 0 \\ 0 & \cot\left(\frac{\theta}{2}\right) & 0 & 0 \\ 0 & 0 & \frac{z_{\text{near}} + z_{\text{far}}}{z_{\text{near}} - z_{\text{far}}} & -\frac{2z_{\text{near}}z_{\text{far}}}{z_{\text{near}} - z_{\text{far}}} \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{7.41}$$

La transformación completa desde coordenadas universales a coordenadas normalizadas de proyección en perspectiva es la matriz compuesta formada al concatenar esta matriz de perspectiva a la izquierda del producto de transformación de visualización  $\mathbf{R} \cdot \mathbf{T}$ . A continuación, podemos aplicar las rutinas de recorte al volumen de visualización normalizado. Las tareas restantes serán la determinación de visibilidad, la representación de superficies y la transformación del visor.

## 7.9 TRANSFORMACIÓN DEL VISOR Y COORDENADAS DE PANTALLA TRIDIMENSIONALES

---

Una vez completada la transformación a coordenadas de proyección normalizadas, pueden aplicarse los procedimientos de recorte de manera eficiente al cubo simétrico (o al cubo unitario). Después de aplicados esos procedimientos de recorte, el contenido del volumen de visualización normalizado puede transferirse a coordenadas de pantalla. Para las posiciones  $x$  e  $y$  en la ventana de recorte normalizada, este procedimiento es igual a la transformación de visor bidimensional que hemos analizado en la Sección 6.3. Pero las posiciones del volumen de visualización tridimensional también tienen una profundidad (coordenada  $z$ ) y necesitamos retener esta información de profundidad para realizar las comprobaciones de visibilidad y aplicar los algoritmos de representación de superficies. Así que podemos considerar ahora la transformación de visor como un mapeo sobre las **coordenadas de pantalla tridimensionales**.

Las ecuaciones de transformación  $x$  e  $y$  desde la ventana de recorte normalizada a las correspondientes posiciones dentro de un visor rectangular están dadas por la Matriz 6.10. Podemos adaptar dicha matriz a las aplicaciones tridimensionales incluyendo los parámetros necesarios para la transformación de los valores  $z$  a coordenadas de pantalla. A menudo, los valores  $z$  normalizados dentro del cubo simétrico se renormalizan en el rango que va de 0 a 1.0. Esto permite referenciar la pantalla de video como  $z = 0$ , con lo que el procesamiento de los valores de profundidad puede llevarse a cabo de manera conveniente sobre el intervalo unitario que va de 0 a 1. Si incluimos esta renormalización de  $z$ , la transformación del volumen de visualización normalizado a coordenadas de pantalla tridimensionales es:

$$\mathbf{M}_{\text{normviewvol,3D screen}} = \begin{bmatrix} \frac{xv_{\max} - xv_{\min}}{2} & 0 & 0 & \frac{xv_{\max} + xv_{\min}}{2} \\ 0 & \frac{yv_{\max} - yv_{\min}}{2} & 0 & \frac{yv_{\max} + yv_{\min}}{2} \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (7.42)$$

En coordenadas normalizadas, la cara  $z_{\text{norm}} = -1$  del cubo simétrico se corresponde con el área de la ventana de recorte. Y esta cara del cubo normalizado se mapea sobre el visor rectangular, que ahora estará referenciado en  $z_{\text{screen}} = 0$ . Así, la esquina inferior izquierda del área de pantalla del visor estará en la posición  $(xv_{\min}, yv_{\min}, 0)$  y la esquina superior izquierda se encontrará en  $(xv_{\max}, yv_{\max}, 0)$ .

Cada posición  $xy$  del visor se corresponde con una posición del búfer de refresco, que contiene la información de color para dicho punto de la pantalla. Y el valor de profundidad para cada punto de la pantalla se almacena en otra área de búfer, denominada *búfer de profundidad*. En capítulos posteriores, analizaremos los algoritmos para determinar las posiciones de las superficies visibles y sus colores.

Posicionemos el visor rectangular sobre la pantalla exactamente igual que hacíamos en las aplicaciones bidimensionales. La esquina inferior izquierda del visor se suele colocar en una posición cuyas coordenadas se especifican en relación con la esquina inferior izquierda de la ventana de visualización. Y las proporciones de los objetos se mantienen si hacemos que la relación de aspecto de este área del visor sea igual que la de la ventana de recorte.

## 7.10 FUNCIONES DE VISUALIZACIÓN TRIDIMENSIONAL OpenGL

---

La biblioteca GLU (OpenGL Utility) incluye una función para especificar los parámetros de visualización tridimensional y otra función para configurar una transformación de proyección en perspectiva simétrica. Otras funciones, como las de proyección ortogonal, proyección en perspectiva oblicua y transformación de visor están contenidas en la biblioteca básica OpenGL. Además, hay disponibles funciones GLUT para definir y manipular ventanas de visualización (Sección 6.4).

### Función de transformación de visualización OpenGL

Cuando especificamos los parámetros de visualización en OpenGL, se forma una matriz y se la concatena con la matriz actual de visualización del modelo. En consecuencia, esta matriz de visualización se combina con cualesquiera transformaciones geométricas que hayamos podido también especificar. Después, esta matriz compuesta se aplica para transformar las descripciones de los objetos en coordenadas universales y expresarlas en coordenadas de visualización. Podemos activar el modo de visualización del modelo mediante la instrucción:

```
glMatrixMode (GL_MODELVIEW);
```

Los parámetros de visualización se especifican mediante la siguiente función GLU, que se encuentra en la biblioteca OpenGL Utility porque invoca las rutinas de traslación y rotación de la biblioteca básica OpenGL:

```
gluLookAt (x0, y0, z0, xref, yref, zref, Vx, Vy, Vz);
```

Es necesario asignar valores de coma flotante y doble precisión a todos los parámetros de esta función. Esta función designa el origen del sistema de referencia de visualización mediante el punto  $\mathbf{P}_0 = (x_0, y_0, z_0)$  en coordenadas universales; la posición de referencia se designa mediante  $\mathbf{P}_{\text{ref}} = (x_{\text{ref}}, y_{\text{ref}}, z_{\text{ref}})$  y el vector vertical será  $\mathbf{V} = (V_x, V_y, V_z)$ . El eje  $z_{\text{view}}$  positivo para el sistema de coordenadas de visualización estará en la dirección  $\mathbf{N} = \mathbf{P}_0 + \mathbf{P}_{\text{ref}}$  y los vectores unitarios de eje para el sistema de referencia de visualización se calculan mediante las Ecuaciones 7.1.

Puesto que la dirección de visualización está definida según el eje  $-z_{\text{view}}$ , la posición de referencia  $\mathbf{P}_{\text{ref}}$  también se denomina «punto observado». Normalmente, este punto observado es alguna posición en el centro de la escena que podemos usar como referencia para especificar los parámetros de proyección. Y podemos considerar la posición de referencia como el punto al que dirigiríamos una cámara que estuviera ubicada en el origen de visualización. La orientación vertical de la cámara se especifica mediante el vector  $\mathbf{V}$ , que se ajusta a una dirección perpendicular a  $\mathbf{N}$ .

Los parámetros de visualización especificados mediante la función `gluLookAt` se utilizan para formar la matriz de transformación de visualización 7.4 de la que hemos hablado en la Sección 7.4. Esta matriz se forma como una combinación de una traslación (que desplaza el origen de visualización al origen de coordenadas universales) y una rotación, que alinea los ejes de visualización con los ejes universales.

Si no invocamos la función `gluLookAt`, los parámetros de visualización predeterminados en OpenGL son:

$$\mathbf{P}_0 = (0, 0, 0)$$

$$\mathbf{P}_{\text{ref}} = (0, 0, -1)$$

$$\mathbf{V} = (0, 1, 0)$$

Para estos valores predeterminados, el sistema de referencia de visualización coincide con el de coordenadas universales, con la dirección de visualización según el eje  $z_{\text{world}}$  negativo. En muchas aplicaciones, podemos usar sin problemas los valores predeterminados como parámetros de visualización.

## Función de proyección ortogonal OpenGL

Las matrices de proyección se almacenan en el modo de proyección OpenGL. Por tanto, para definir una matriz de transformación de proyección, debemos primero invocar dicho modo con la instrucción,

```
glMatrixMode (GL_PROJECTION);
```

Entonces, cuando ejecutemos cualquier comando de transformación, la matriz resultante se concatenará con la matriz de proyección actual.

Los parámetros de proyección ortogonal se eligen mediante la función:

```
glOrtho (xwmin, xwmax, ywmin, ywmax, dnear, dfar);
```

Todos los valores de los parámetros en esta función deben ser números en coma flotante y doble precisión. Utilizamos `glOrtho` para seleccionar las coordenadas de la ventana de recorte y las distancias entre el origen de visualización y los planos de recorte próximo y lejano. En OpenGL no hay ninguna opción para definir la situación del plano de visualización. El plano de recorte próximo coincide siempre con el plano de visualización, por lo que la ventana de recorte estará siempre situada sobre el plano próximo del volumen de visualización.

La función `glOrtho` genera una proyección paralela que es perpendicular al plano de visualización (el plano de recorte próximo). Así, esta función crea un volumen de visualización finito de proyección ortogonal para la ventana de recorte y para los planos de recorte especificados. En OpenGL, los planos de recorte próximo y lejano no son opcionales; siempre hay que especificarlos para toda transformación de proyección.

Los parámetros `dnear` y `dfar` denotan las distancias en la dirección  $z_{\text{view}}$  negativa, a partir del origen del sistema de coordenadas de visualización. Por ejemplo, si `dfar = 55.0`, entonces el plano lejano de recorte estará en la posición  $z_{\text{far}} = -55.0$ . Un valor negativo de alguno de los parámetros denotará una distancia «por detrás» del origen de visualización, según el eje  $z_{\text{view}}$  positivo. Podemos asignar los valores que queramos (positivos, negativos o cero) a estos parámetros, siempre que se cumpla que `dnear < dfar`.

El volumen de visualización resultante para esta transformación de proyección es un paralelepípedo rectangular. Las coordenadas dentro de este volumen de visualización se transforman a ubicaciones dentro del cubo normalizado simétrico, en un sistema de referencia que cumple con la regla de la mano izquierda, utilizando la Matriz 7.7 con  $z_{\text{near}} = -\text{dnear}$  y  $z_{\text{far}} = -\text{dfar}$ .

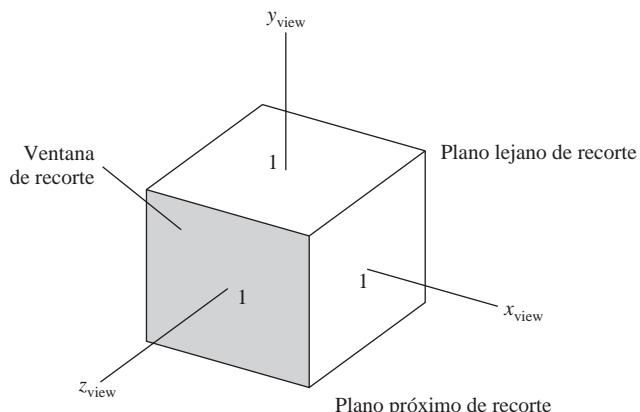
Los valores predeterminados de los parámetros para la función de proyección ortogonal OpenGL son  $\pm 1$ , que producen un volumen de visualización que es un cubo normalizado simétrico en el sistema de coordenadas de visualización, que cumple la regla de la mano derecha. Estos valores predeterminados son equivalentes a ejecutar la instrucción:

```
glOrtho (-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
```

La ventana de recorte predeterminada es, por tanto, un cuadrado normalizado simétrico y el volumen de normalización predeterminado es un cubo normalizado simétrico con  $z_{\text{near}} = 1.0$  (por detrás de la posición de visualización) y  $z_{\text{far}} = -1.0$ . La Figura 7.54 muestra la apariencia y la posición del volumen de visualización predeterminado para proyección ortogonal.

Para aplicaciones bidimensionales, utilizábamos la función `gluOrtho2D` para definir la ventana de recorte. También podríamos haber usado la función `glOrtho` para especificar la ventana de recorte, siempre y cuando asignáramos a los parámetros `dnear` y `dfar` valores situados en lados opuestos del origen de coordenadas. De hecho, una llamada a `gluOrtho2D` es equivalente a otra llamada a `glOrtho` con `dnear = -1.0` y `dfar = 1.0`.

No hay ninguna función en OpenGL para generar una proyección oblicua. Para producir una vista de proyección oblicua de una escena, podríamos definir nuestra propia matriz de proyección, como en la Ecuación 7.14; después, tendríamos que hacer de esta matriz la matriz de proyección OpenGL actual, utilizando las funciones de matrices que hemos analizado en la Sección 5.17. Otra forma de generar una vista en proyección oblicua consiste en rotar la escena hasta una posición apropiada, de modo que una posición ortogonal en la dirección  $z_{\text{view}}$  nos de la vista deseada.



**FIGURA 7.54.** Volumen de visualización predeterminado para proyección ortogonal. Los rangos de coordenadas para este cubo simétrico van de  $-1$  a  $+1$  en cada dirección. El plano de recorte próximo está en  $z_{\text{near}} = 1$  y el plano de recorte lejano en  $z_{\text{far}} = -1$ .

### Función OpenGL de proyección en perspectiva simétrica

Hay dos funciones disponibles para generar una vista de proyección en perspectiva de una escena. Una de estas funciones genera un frustum de visualización simétrico en torno a la dirección de visualización (el eje  $z_{\text{view}}$  negativo). La otra función puede usarse para una proyección en perspectiva simétrica o una proyección en perspectiva oblicua. Para ambas funciones, el punto de referencia de producción se encuentra en el origen del sistema de coordenadas de visualización y el plano próximo de recorte se encuentra sobre el plano de visualización.

El frustum de visualización simétrico para proyección en perspectiva se define mediante la función GLU,

```
gluPerspective (theta, aspect, dnear, dfar);
```

donde a cada uno de estos cuatro parámetros se le asigna un número de coma flotante y doble precisión. Los dos primeros parámetros definen el tamaño y la posición de la ventana de recorte sobre el plano próximo, mientras que los últimos dos parámetros especifican las distancias entre el punto de vista (origen de coordenadas) y los planos de recorte próximo y lejano. El parámetro `theta` representa el ángulo del campo visual, que es el ángulo entre los planos de recorte superior e inferior (Figura 7.48). Podemos asignar a este ángulo cualquier valor desde  $0^\circ$  hasta  $180^\circ$ . Al parámetro `aspect` se le asigna el valor correspondiente a la relación de aspecto (anchura/altura) de la ventana de recorte.

Para una proyección en perspectiva OpenGL, los planos de recorte próximo y lejano deben estar situados sobre el eje  $z_{\text{view}}$  negativo; ninguno de los dos puede estar «detrás» de la posición de visualización. Esta restricción no se aplica a una proyección ortogonal, pero impide la proyección en perspectiva invertida de un objeto cuando el plano de visualización se encuentra por detrás del punto de vista. Por tanto, tanto `dnear` como `dfar` deben tener valores numéricos positivos y las posiciones de los planos próximo y lejano se calculan como  $z_{\text{near}} = -\text{dnear}$  y  $z_{\text{far}} = -\text{dfar}$ .

Si no especificamos una función de proyección, nuestra escena se mostrará utilizando la proyección ortogonal predeterminada. En este caso, el volumen de visualización es el cubo normalizado simétrico que se muestra en la Figura 7.54.

El frustum de visualización definido mediante la función `gluPerspective` es simétrico en torno al eje  $z_{\text{view}}$  negativo. Y la descripción de la escena se convierte a coordenadas de proyección homogéneas normalizadas mediante la Matriz 7.41.

### Función general de proyección de perspectiva OpenGL

Podemos utilizar la siguiente función para especificar una proyección de perspectiva que tenga un frustum de visualización simétrico o un frustum de visualización oblicuo:

```
glFrustum (xwmin, xwmax, ywmin, ywmax, dnear, dfar);
```

Todos los parámetros de esta función deben tener valores de coma flotante y doble precisión. Al igual que las otras funciones de proyección de visualización, el plano próximo se encuentra sobre el plano de visualización y el punto de referencia de proyección está situado en la posición de visualización (origen de coordenadas). Esta función tiene los mismos parámetros que la función de proyección paralela ortogonal, pero ahora las distancias de los planos de recorte próximo y lejano deben ser positivas. Los primeros cuatro parámetros establecen las coordenadas de la ventana de recorte sobre el plano próximo, mientras que los últimos dos especifican la distancia entre el origen de coordenadas y los planos próximo y lejano de recorte a lo largo del eje  $z_{\text{view}}$  negativo. Las ubicaciones de los planos próximo y lejano serán  $z_{\text{near}} = -dnear$  y  $z_{\text{far}} = -dfar$ .

La ventana de recorte puede especificarse en cualquier punto de plano próximo. Si seleccionamos las coordenadas de la ventana de recorte de modo que  $xw_{\text{min}} = -xw_{\text{max}}$  y  $yw_{\text{min}} = -yw_{\text{max}}$ , obtendremos un frustum simétrico (teniendo como línea central el eje  $z_{\text{view}}$  negativo).

De nuevo, si no invocamos explícitamente un comando de proyección, OpenGL aplicará la proyección predeterminada ortogonal a la escena. El volumen de visualización en este caso será el cubo simétrico (Figura 7.54).

## Visores OpenGL y ventanas de visualización

Después de haber aplicado las rutinas de recorte en coordenadas normalizadas, el contenido de la ventana de recorte normalizada se transfiere a coordenadas de pantalla tridimensionales junto con la información de profundidad. El valor de color de cada posición  $xy$  del visor se almacena en el búfer de refresco (búfer de color) y la información de profundidad para cada posición  $xy$  se almacena en el búfer de profundidad.

Como hemos indicado en la Sección 6.4, un visor rectangular se define mediante la siguiente función OpenGL:

```
glViewport (xvmin, yvmin, vpWidth, vpHeight);
```

Los primeros dos parámetros de esta función especifican la posición entera de pantalla de la esquina inferior izquierda del visor, de forma relativa a la esquina inferior izquierda de la ventana de visualización. Los dos últimos parámetros, por su parte, proporcionan la anchura y altura enteras del visor. Para mantener las proporciones de los objetos en la escena, definiremos la relación de aspecto del visor de modo que sea igual a la de la ventana de recorte.

Las ventanas de visualización se crean y gestionan mediante rutinas GLUT y en la Sección 6.4 hemos analizado en detalle las diversas funciones de la biblioteca GLUT para ventanas de visualización. El visor predeterminado OpenGL tiene el tamaño y la posición de la ventana de visualización actual.

## Ejemplo de programa OpenGL para visualización tridimensional

Una vista de proyección en perspectiva de un cuadrado, como la que se muestra en la Figura 7.55, puede obtenerse utilizando el siguiente programa de ejemplo. El cuadrado está definido en el plano  $xy$  y seleccionamos un origen de coordenadas de visualización con el fin de ver la cara frontal con un cierto ángulo. Seleccionando el centro del cuadrado como punto observado, obtenemos una vista en perspectiva utilizando la función `glFrustum`. Si movemos el origen de coordenadas de visualización al otro lado del polígono, se mostraría la cara posterior como un objeto alámbrico.

```
#include <GL/glut.h>
GLint winWidth = 600, winHeight = 600; // Tamaño inicial ventana de visualización.
GLfloat x0 = 100.0, y0 = 50.0, z0 = 50.0; // Origen coordenadas de visualización.
GLfloat xref = 50.0, yref = 50.0, zref = 0.0; // Punto observado.
GLfloat Vx = 0.0, Vy = 1.0, Vz = 0.0; // Vector vertical.
```

```
/* Establecer límites de coordenadas para ventana de recorte: */
GLfloat xwMin = -40.0, ywMin = -60.0, xwMax = 40.0, ywMax = 60.0;

/* Establecer posición de los planos de recorte próximo y lejano: */
GLfloat dnear = 25.0, dfar = 125.0;

void init (void)
{
    glClearColor (1.0, 1.0, 1.0, 0.0);

    glMatrixMode (GL_MODELVIEW);
    gluLookAt (x0, y0, z0, xref, yref, zref, vx, vy, vz);
    glMatrixMode (GL_PROJECTION);
    glFrustum (xwMin, xwMax, ywMin, ywMax, dnear, dfar);
}

void displayFcn (void)
{
    glClear (GL_COLOR_BUFFER_BIT);

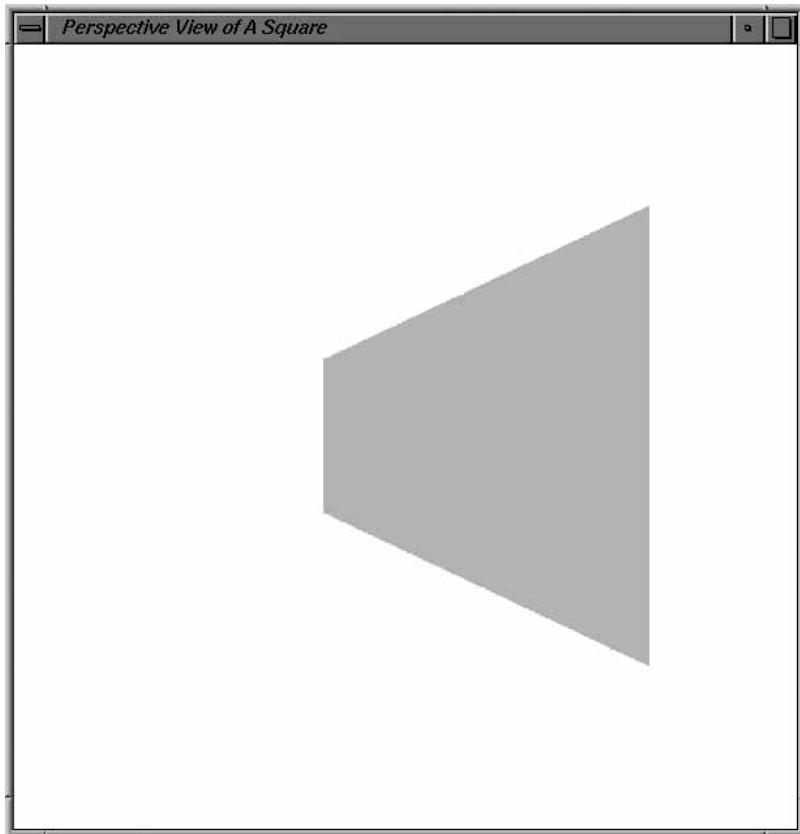
    /* Establecer parámetros para un área de relleno cuadrada. */
    glColor3f (0.0, 1.0, 0.0); // Seleccionar color de relleno verde.
    glPolygonMode (GL_FRONT, GL_FILL);
    glPolygonMode (GL_BACK, GL_LINE); // Cara posterior alámbrica.
    glBegin (GL_QUADS);
        glVertex3f (0.0, 0.0, 0.0);
        glVertex3f (100.0, 0.0, 0.0);
        glVertex3f (100.0, 100.0, 0.0);
        glVertex3f (0.0, 100.0, 0.0);
    glEnd ();
    glFlush ();
}

void reshapeFcn (GLint newWidth, GLint newHeight)
{
    glViewport (0, 0, newWidth, newHeight);

    winWidth = newWidth;
    winHeight = newHeight;
}

void main (int argc, char** argv)
{
    glutInit (&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    glutInitWindowPosition (50, 50);
    glutInitWindowSize (winWidth, winHeight);
    glutCreateWindow ("Perspective View of A Square");

    init ();
    glutDisplayFunc (displayFcn);
    glutReshapeFunc (reshapeFcn);
    glutMainLoop ();
}
```



**FIGURA 7.55.** Imagen generada por el programa de ejemplo de visualización tridimensional.

## 7.11 ALGORITMOS DE RECORTE TRIDIMENSIONAL

En el Capítulo 6 hemos hablado de las ventajas de utilizar el contorno normalizado de la ventana de recorte en los algoritmos de recorte bidimensionales. De forma similar, podemos aplicar los algoritmos de recorte tridimensionales al contorno normalizado del volumen de visualización. Esto permite implementar de manera muy eficiente la pipeline de visualización y los procedimientos de recorte. Todas las transformaciones independientes del dispositivo (geométricas y de visualización) se concatenan y aplican antes de ejecutar las rutinas de recorte. Y cada uno de los límites de recorte para el volumen de visualización normalizado es un plano paralelo a uno de los planos cartesianos, independientemente del tipo de proyección y de la forma original del volumen de visualización. Dependiendo de si el volumen de visualización ha sido normalizado a un cubo unitario o a un cubo simétrico con lado de longitud igual a 2, los planos de recorte estarán situados en las coordenadas 0 y 1 o -1 y 1. Para el cubo simétrico, las ecuaciones de los planos de recorte tridimensionales serán:

$$\begin{aligned} xw_{\min} &= -1, & xw_{\max} &= 1 \\ yw_{\min} &= -1, & yw_{\max} &= 1 \\ zw_{\min} &= -1, & zw_{\max} &= 1 \end{aligned} \tag{7.43}$$

Los límites de recorte  $x$  e  $y$  son los límites normalizados de la ventana de recorte, mientras que los límites de recorte  $z$  son las posiciones normalizadas de los planos de recorte próximo y lejano.

Los algoritmos de recorte para visualización tridimensional identifican y guardan todas las secciones de los objetos que se encuentran dentro del volumen de visualización normalizado, para mostrarlas en el dispo-

sitivo de salida. Todas las partes de los objetos que se encuentren fuera de los planos de recorte del volumen de visualización se eliminarán. Con ello, los algoritmos serán ahora extensiones de métodos bidimensionales, utilizando los planos de contorno normalizados del volumen de visualización en lugar de usar las líneas de contorno de la ventana de recorte normalizada.

## Recorte en coordenadas homogéneas tridimensionales

Las bibliotecas de generación de gráficos por computadora procesan las posiciones en el espacio como coordenadas homogéneas de cuatro dimensiones, de modo que todas las transformaciones pueden representarse como matrices 4 por 4. A medida que cada conjunto de coordenadas entra en la pipeline de visualización, se lo convierte a una representación en cuatro dimensiones:

$$(x, y, z) \rightarrow (x, y, z, 1)$$

Después de que unas ciertas coordenadas han pasado a través de las transformaciones geométrica, de visualización y de proyección, estarán en la forma homogénea:

$$\begin{bmatrix} x_h \\ y_h \\ z_h \\ h \end{bmatrix} = \mathbf{M} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (7.44)$$

donde la matriz  $\mathbf{M}$  representa la concatenación de todas las diversas transformaciones de coordenadas universales a coordenadas de proyección homogéneas normalizadas, y el parámetro homogéneo  $h$  puede no tener ya el valor 1. De hecho,  $h$  puede tener cualquier valor real, dependiendo de cómo hayamos representado los objetos en la escena y del tipo de proyección que utilicemos.

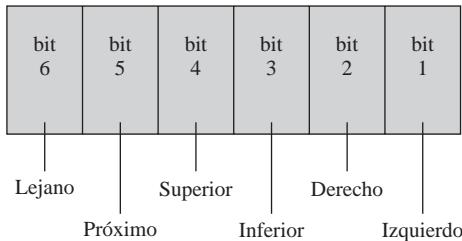
Si el parámetro homogéneo  $h$  tiene el valor 1, las coordenadas homogéneas serán iguales que las coordenadas de proyección cartesianas. Esto suele suceder en el caso de transformación de proyección paralela, pero una proyección en perspectiva produce un parámetro homogéneo que está en función de la coordenada  $z$  de cada punto concreto del espacio. El parámetro homogéneo de proyección en perspectiva puede incluso ser negativo, lo que ocurre cuando un cierto punto se encuentra por detrás del punto de referencia de proyección. Asimismo, la representación mediante splines regionales de las superficies de los objetos suele formularse en coordenadas homogéneas, donde el parámetro homogéneo puede ser positivo o negativo. Por tanto, si se realiza el recorte en coordenadas de proyección después de la división por el parámetro homogéneo  $h$ , puede perderse cierta información acerca de las coordenadas y puede que los objetos no se recorten correctamente.

Un método efectivo para tratar con todas las posibles transformaciones de proyección y todas las posibles representaciones de los objetos consiste en aplicar las rutinas de recorte a la representación en coordenadas homogéneas de los puntos del espacio. Además, como todos los volúmenes de visualización pueden convertirse a un cubo normalizado, basta con implementar un único procedimiento de recorte en el hardware para recortar los objetos en coordenadas homogéneas de acuerdo con los planos de recorte normalizados.

## Códigos de región tridimensional

Podemos ampliar el concepto de código de región (Sección 6.7) a tres dimensiones añadiendo simplemente un par de bits adicionales para tomar en consideración los planos de recorte próximo y lejano. Por tanto, ahora usaremos un código de región de seis bits, como se ilustra en la Figura 7.56. Las posiciones de los bits en este ejemplo de código de región están numeradas de derecha a izquierda, y hacen referencia a los planos de recorte izquierdo, derecho, inferior, superior, próximo y lejano, en dicho orden.

Las condiciones para asignar valores a los bits del código de región son básicamente las mismas que las de la Sección 6.7, añadiendo simplemente las dos condiciones adicionales para los planos de recorte próximo y lejano. Sin embargo, para una escena tridimensional, necesitamos aplicar las rutinas de recorte a las



**FIGURA 7.56.** Una posible ordenación de los límites de recorte del volumen de visualización, que se corresponden con las posiciones de bit en el código de región.

coordenadas de producción, que habrán sido transformadas a un espacio normalizado. Después de la transformación de proyección, cada punto de una escena tiene la representación de cuatro componentes  $\mathbf{P} = (x_h, y_h, z_h, h)$ . Suponiendo que estemos efectuando el recorte de acuerdo con los límites del cubo simétrico normalizado (Ecuaciones 7.43), un punto estará dentro de este volumen de visualización normalizado si las coordenadas de proyección del punto satisfacen las siguientes seis desigualdades:

$$-1 \leq \frac{x_h}{h} \leq 1, \quad -1 \leq \frac{y_h}{h} \leq 1, \quad -1 \leq \frac{z_h}{h} \leq 1 \quad (7.45)$$

A menos que se haya producido un error, el valor del parámetro homogéneo  $h$  será distinto de cero. Pero, antes de implementar los procedimientos del código de región, podemos primero comprobar si tenemos un parámetro homogéneo con un valor cero o con un valor extremadamente pequeño. Asimismo, el parámetro homogéneo puede ser positivo o negativo. Por tanto, suponiendo que  $h \neq 0$ , podemos escribir las desigualdades anteriores de la forma:

$$-h \leq x_h \leq h, \quad -h \leq y_h \leq h, \quad -h \leq z_h \leq h \quad \text{si } h > 0 \quad (7.46)$$

$$h \leq x_h \leq -h, \quad h \leq y_h \leq -h, \quad h \leq z_h \leq -h \quad \text{si } h < 0$$

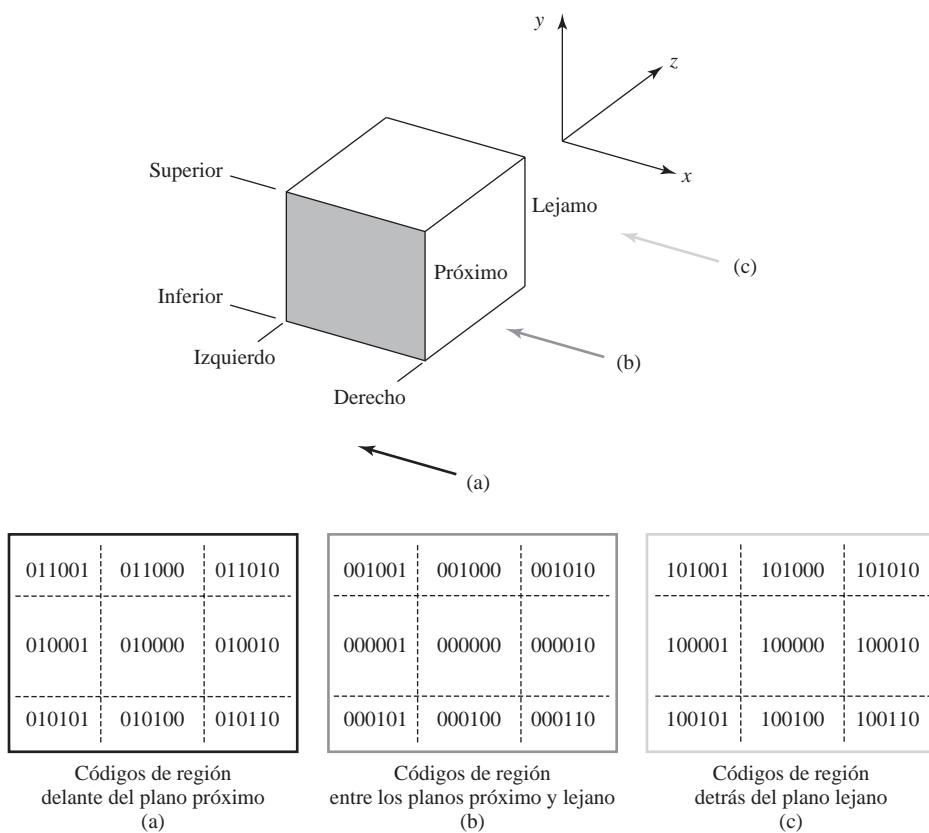
En la mayoría de los casos  $h > 0$  y podemos asignar los valores de bit del código de región para un determinado punto de acuerdo con las comprobaciones:

$$\begin{aligned} \text{bit 1} &= 1 && \text{si } h + x_h < 0 \quad (\text{izquierdo}) \\ \text{bit 2} &= 1 && \text{si } h - x_h < 0 \quad (\text{derecho}) \\ \text{bit 3} &= 1 && \text{si } h + y_h < 0 \quad (\text{inferior}) \\ \text{bit 4} &= 1 && \text{si } h - y_h < 0 \quad (\text{superior}) \\ \text{bit 5} &= 1 && \text{si } h + z_h < 0 \quad (\text{próximo}) \\ \text{bit 6} &= 1 && \text{si } h - z_h < 0 \quad (\text{lejano}) \end{aligned} \quad (7.47)$$

Estos valores de bit pueden definirse utilizando la misma técnica que en el recorte bidimensional. Es decir, simplemente utilizamos el bit de signo de uno de los cálculos  $h \pm x_h$ ,  $h \pm y_h$  o  $h \pm z_h$  para asignar el correspondiente valor al bit del código de región. La Figura 7.57 enumera los 27 códigos de región para un volumen de visualización. En aquellos casos en los que  $h < 0$  para algún punto, podríamos aplicar los mecanismos de recorte utilizando el segundo conjunto de desigualdades de la Ecuación 7.46, o podríamos invertir el signo de las coordenadas y efectuar el recorte utilizando las comprobaciones para  $h > 0$ .

## Recorte tridimensional de puntos y líneas

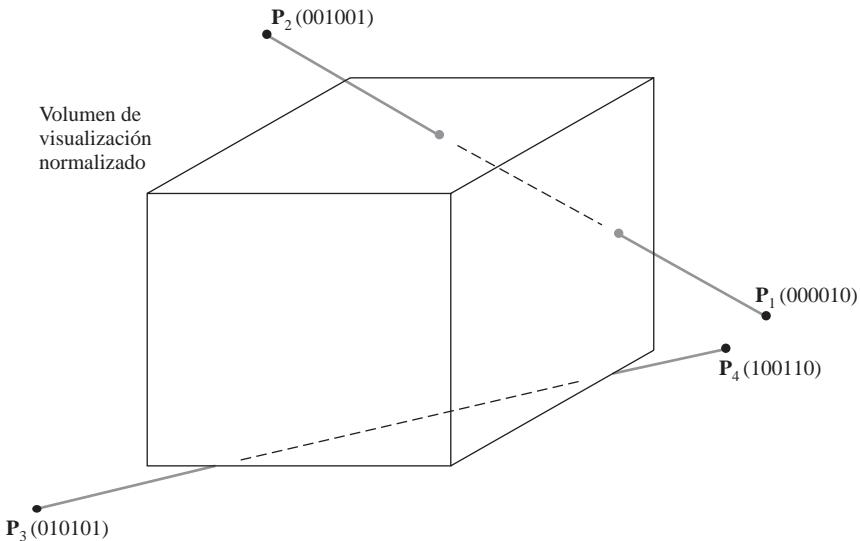
Para puntos y segmentos de línea recta que están definidos en una escena y no se encuentran detrás del punto de referencia de proyección, todos los parámetros homogéneos son positivos y los códigos de región pueden



**FIGURA 7.57.** Valores para el código de región tridimensional de seis bits que identifica las posiciones en el espacio en relación con los límites del volumen de visualización.

establecerse utilizando las condiciones 7.47. Entonces, una vez determinado el código de región para cada posición de la escena, podemos determinar fácilmente si un punto está fuera o dentro del volumen de visualización. Por ejemplo, un código de región 101000 nos dice que el punto está por encima y directamente detrás del volumen de visualización, mientras que el código de región 000000 indica un punto situado dentro del volumen (Figura 7.57). Por tanto, para el recorte de puntos, simplemente eliminaremos todos los puntos individuales cuyo código de región no sea 000000. En otras palabras, si cualquiera de los test 7.47 es negativo, el punto estará fuera del volumen de visualización.

Los métodos para el recorte de líneas tridimensionales son esencialmente iguales que para líneas bidimensionales. Podemos comprobar primero los códigos de región de los extremos de la línea para la aceptación o rechazos triviales de la línea. Si el código de región de ambos extremos de una línea es 000000, la línea estará completamente contenida dentro del volumen de visualización. De manera equivalente, podemos aceptar trivialmente la línea si la operación *or* lógica de los dos códigos de región de los extremos produce un valor igual a 0. Asimismo, podemos rechazar trivialmente la línea si la operación lógica *and* de los dos códigos de región de los extremos produce un valor distinto de 0. Este valor distinto de cero indica que los códigos de región de los dos extremos tienen un valor 1 en la misma posición de bit, por lo que la línea estará completamente fuera de los planos de recorte. Como ejemplo, la línea que va de  $P_3$  a  $P_4$  en la Figura 7.58 tiene los valores de código de región 010101 y 100110 para sus dos puntos extremos. Por tanto, esta línea está completamente por debajo del plano de recorte inferior. Si una línea no cumple ninguno de estos dos tests, analizaremos la ecuación de la línea para determinar si es necesario preservar alguna parte de la misma.



**FIGURA 7.58.** Códigos de región tridimensionales para dos segmentos de línea. La línea  $P_1P_2$  intersecta los límites de recorte derecho y superior del volumen de visualización, mientras que la línea  $P_3P_4$  está completamente situada por debajo del plano de recorte inferior.

Las ecuaciones de los segmentos de línea tridimensionales pueden expresarse de manera conveniente en forma paramétrica y los métodos de recorte de Cyrus-Beck o Liang-Barsky (Sección 6.7) pueden ampliarse a escenas tridimensionales. Para un segmento de línea con extremos  $P_1 = (x_{h1}, y_{h1}, z_{h1}, h_1)$  y  $P_2 = (x_{h2}, y_{h2}, z_{h2}, h_2)$ , podemos escribir la ecuación paramétrica que describe a los puntos situados a lo largo de la línea como:

$$\mathbf{P} = \mathbf{P}_1 + (\mathbf{P}_2 - \mathbf{P}_1)u \quad 0 \leq u \leq 1 \quad (7.48)$$

Cuando el parámetro de la línea tiene el valor  $u = 0$ , estaremos en la posición  $\mathbf{P}_1$ , mientras que si  $u = 1$  estaremos en el otro extremo de la línea,  $\mathbf{P}_2$ . Escribiendo la ecuación paramétrica de la línea explícitamente en términos de las coordenadas homogéneas, tendremos,

$$\begin{aligned} x_h &= x_{h1} + (x_{h2} - x_{h1})u \\ y_h &= y_{h1} + (y_{h2} - y_{h1})u \quad 0 \leq u \leq 1 \\ z_h &= z_{h1} + (z_{h2} - z_{h1})u \\ h &= h_1 + (h_2 - h_1)u \end{aligned} \quad (7.49)$$

Utilizando los códigos de región de los extremos de un segmento de línea, podemos primero determinar con qué planos de recorte intersecta. Si uno de los códigos de región de los extremos tiene un valor 0 en una determinada posición de bit mientras que el otro código tiene un valor 1 en la misma posición de bit, la línea cruzará dicho plano de recorte. En otras palabras, uno de los tests 7.47 genera un valor negativo, mientras que el mismo test para el otro extremo de la línea produce un valor no negativo. Para hallar el punto de intersección con este plano de recorte, primero utilizamos las ecuaciones apropiadas de 7.49 para determinar el valor correspondiente al parámetro  $u$ . Después, calculamos las coordenadas del punto de intersección.

Como ejemplo del procedimiento de cálculo de intersección, vamos a considerar un segmento de línea  $\overline{P_1P_2}$  de la Figura 7.58. Esta línea intersecta el plano de recorte derecho, que puede describirse con la ecuación  $x_{\max} = 1$ . Por tanto, determinamos el valor de intersección del parámetro  $u$  haciendo igual a 1 la coordenada de proyección  $x$ :

$$x_p = \frac{x_h}{h} = \frac{x_{h1} + (x_{h2} - x_{h1})u}{h_1 + (h_2 - h_1)u} = 1 \quad (7.50)$$

Despejando el parámetro  $u$  obtenemos:

$$u = \frac{x_{h1} - h_1}{(x_{h1} - h_1) - (x_{h2} - h_2)} \quad (7.51)$$

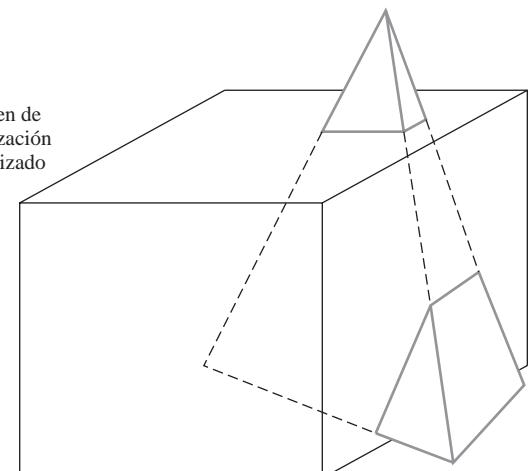
A continuación, determinamos los valores  $y_p$  y  $z_p$  en este plano de recorte utilizando el valor calculado de  $u$ . En este caso, los valores de intersección  $y_p$  y  $z_p$  se encuentran dentro de los límites  $\pm 1$  del volumen de visualización y la línea pasa al interior de dicho volumen. Por tanto, procedemos a continuación a localizar el punto de intersección con el plano de recorte superior. Eso completaría el procesamiento para este segmento de línea, porque los puntos de intersección con los planos de recorte superior y derecho identifican la parte de la línea contenida dentro del volumen de visualización, así como las secciones de la línea que caen fuera de dicho volumen.

Cuando una línea intersecta un plano de recorte pero no pasa al interior del volumen de visualización, continuamos el procesamiento de la línea como en el caso del recorte bidimensional. La sección de la línea que cae fuera de dicho límite de recorte se elimina y actualizamos la información de código de región y los valores del parámetro  $u$  para la parte de la línea que está dentro de dicho límite. Después, comprobamos la sección restante de la línea de acuerdo con los otros planos de recorte para su posible rechazo o para realizar cálculos adicionales de intersección.

Los segmentos de línea en las escenas tridimensionales no suelen estar aislados, sino que suelen ser componentes de la descripción de los objetos sólidos de la escena, con lo que necesitamos procesar las líneas como parte de las rutinas de recorte de superficie.

### Recorte de polígonos tridimensionales

Los paquetes gráficos normalmente tratan sólo con escenas que contienen «objetos gráficos». Se trata de objetos cuyos contornos se describen mediante ecuaciones lineales, por lo que cada objeto está compuesto por un conjunto de polígonos de superficie. Por tanto, para recortar los objetos de una escena tridimensional, aplicamos las rutinas de recorte a las superficies del polígono. La Figura 7.59, por ejemplo, resalta las secciones de superficie de una pirámide que hay que recortar, mientras que las líneas discontinuas muestran las secciones de las superficies de los polígonos que se encuentran dentro del volumen de visualización.



**FIGURA 7.59.** Recorte de un objeto tridimensional. Las secciones de la superficie que caen fuera de los planos de recorte del volumen de visualización se eliminan de la descripción del objeto, pudiendo ser necesario construir nuevas caras de la superficie.

Podemos primero comprobar el poliedro para su aceptación o rechazo triviales utilizando una caja de contorno, una esfera circunscrita o alguna otra medida de los límites de sus coordenadas. Si los límites de las coordenadas del objetos se encuentran dentro de los límites de recorte, conservamos el objeto completo. Si los límites de coordenadas se encuentran completamente fuera de alguno de los límites de recorte, eliminamos el objeto completo.

Cuando no podemos guardar o eliminar el objeto completo, se puede procesar la lista de vértices del conjunto de polígonos que definen las superficies del objeto. Aplicando métodos similares a los del recorte de polígonos bidimensionales, podemos recortar las aristas para obtener nuevas listas de vértices para las superficies del objeto. Puede que también tengamos que crear algunas nuevas de listas de vértices para superficies adicionales que resulten de las operaciones de recorte. Asimismo, las tablas de polígonos deberán ser actualizadas para agregar cualesquiera nuevas superficies de polígonos y para revisar la conectividad y la información de aristas compartidas de las superficies.

Para simplificar el recorte de poliedros generales, las superficies poligonales se suelen dividir en secciones triangulares y describirlas mediante bandas de triángulos. Entonces, podemos recortar las bandas de triángulos utilizando la técnica de Sutherland-Hodgman expuesta en la Sección 3.15. Cada una de las bandas de triángulos se procesa por turnos con respecto a los seis planos de recorte para obtener la lista final de vértices de la banda.

Para polígonos cóncavos, podemos aplicar métodos de división (Sección 3.15) para obtener, por ejemplo, un conjunto de triángulos y luego recortar los triángulos. Alternativamente, podríamos recortar los polígonos tridimensionales cóncavos utilizando el algoritmo de Weiler-Atherton descrito en la Sección 6.8.

## Recorte de curvas tridimensionales

Como en el reparto de poliedros, primero comprobamos si los límites de coordenadas de un objeto curvo, como por ejemplo una esfera o una superficie de tipo spline, se encuentran completamente dentro del volumen de visualización. A continuación tratamos de determinar si el objeto cae completamente fuera de algunos de los seis planos de recorte.

Si estos test triviales de rechazo-aceptación fallan, localizamos las intersecciones con los planos de recorte. Para ello, resolvemos el conjunto de ecuaciones formado por las ecuaciones de la superficie y la ecuación del plano de recorte. Por esta razón, la mayoría de los paquetes gráficos no incluyen rutinas de recorte para objetos curvos. En su lugar, las superficies curvas se aproximan mediante un conjunto de parches poligonales y luego se recortan los objetos utilizando las rutinas de recorte de polígonos. Cuando se aplican procedimientos de representación de superficies a los parches poligonales, estos pueden proporcionar una imagen muy realista de una superficie curva.

## Planos de recorte arbitrarios

También es posible, en algunos paquetes gráficos, recortar una escena tridimensional utilizando planos adicionales que pueden especificarse con cualquier orientación espacial. Esta opción resulta útil en diversos tipos de aplicaciones. Por ejemplo, puede que queramos aislar o recortar un objeto con forma irregular, o que queramos eliminar parte de una escena con un ángulo oblicuo para obtener algún tipo de efecto especial, o que queramos cortar una sección de un objeto según un eje seleccionado para mostrar una vista de sección transversal de su interior.

Los planos de recorte opcional pueden especificarse junto con la descripción de la escena, de modo que las operaciones de recorte se lleven a cabo antes de la transformación de proyección. Sin embargo, esto quiere decir también que las rutinas de recorte deberán implementarse en software.

Puede especificarse un plano de recorte mediante los parámetros del plano  $A, B, C$  y  $D$ . El plano divide entonces el espacio tridimensional en dos partes, de modo que todas las partes de una escena que caigan en un lado del plano se eliminan. Suponiendo que haya que eliminar los objetos situados detrás del plano, todo punto en el espacio  $(x, y, z)$  que satisfaga la siguiente desigualdad será eliminado de la escena:

$$Ax + By + Cz + D < 0 \quad (7.52)$$

Como ejemplo, si el conjunto de parámetros del plano tiene los valores  $(A, B, C, D) = (1.0, 0.0, 0.0, 8.0)$ , entonces todo punto que satisface  $x + 8.0 < 0.0$  (o  $x < -8.0$ ) será eliminado de la escena.

Para recortar un segmento de línea, primero podemos comprobar sus dos extremos para ver si la línea se encuentra completamente detrás del plano de recorte o completamente delante suyo. Podemos representar la desigualdad 7.52 en forma vectorial utilizando el vector normal al plano  $\mathbf{N} = (A, B, C)$ . Entonces, para un segmento de línea con extremos  $\mathbf{P}_1$  y  $\mathbf{P}_2$ , recortaremos la línea completa si ambos extremos satisfacen,

$$\mathbf{N} \cdot \mathbf{P}_k + D < 0, \quad k = 1, 2 \quad (7.53)$$

Y conservaremos la línea completa si ambos extremos satisfacen,

$$\mathbf{N} \cdot \mathbf{P}_k + D \geq 0, \quad k = 1, 2 \quad (7.54)$$

En caso contrario, los extremos se encuentran en lados opuestos del plano de recorte, como ilustra la Figura 7.60, y deberemos calcular el punto de intersección con la línea.

Para calcular el punto de intersección de la línea con el plano de recorte, podemos emplear la siguiente representación paramétrica del segmento de línea:

$$\mathbf{P} = \mathbf{P}_1 + (\mathbf{P}_2 - \mathbf{P}_1)u, \quad 0 \leq u \leq 1 \quad (7.55)$$

El punto  $\mathbf{P}$  se hallará sobre el plano de recorte si satisface la ecuación del plano,

$$\mathbf{N} \cdot \mathbf{P} + D = 0 \quad (7.56)$$

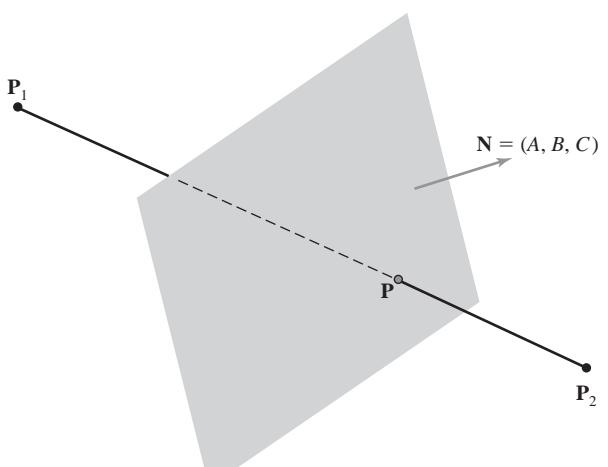
Sustituyendo el valor de  $\mathbf{P}$  dado en 7.55, tendremos:

$$\mathbf{N} \cdot [\mathbf{P}_1 + (\mathbf{P}_2 - \mathbf{P}_1)u] + D = 0 \quad (7.57)$$

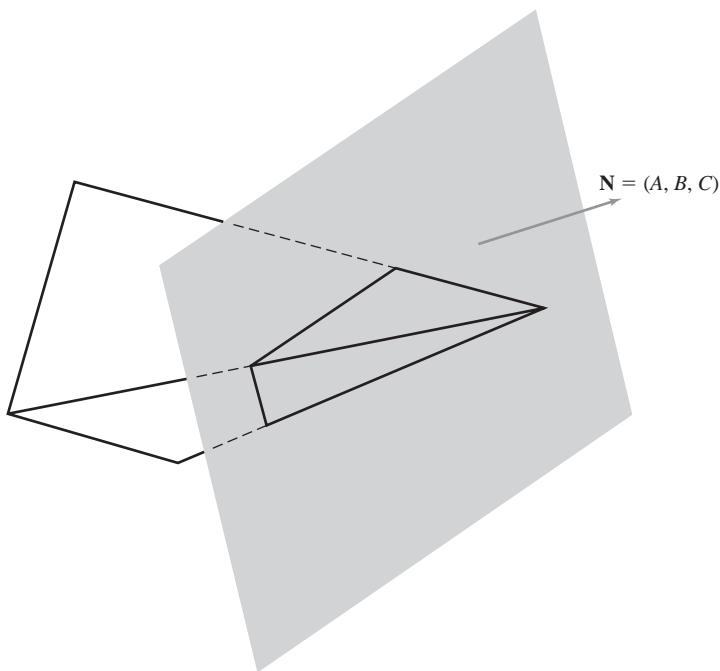
Despejando el parámetro  $u$  en esta ecuación, se obtiene:

$$u = \frac{-D - \mathbf{N} \cdot \mathbf{P}_1}{\mathbf{N} \cdot (\mathbf{P}_2 - \mathbf{P}_1)} \quad (7.58)$$

Entonces sustituimos este valor de  $u$  en la representación paramétrica vectorial de la línea (Ecuación 7.55) para obtener los valores de las coordenadas  $x$ ,  $y$  y  $z$  del punto de intersección. Para el ejemplo de la Figura



**FIGURA 7.60.** Recorte de un segmento de línea mediante un plano con vector normal  $\mathbf{N}$ .



**FIGURA 7.61.** Recorte de las superficies de una pirámide según un plano con vector normal  $\mathbf{N}$ . Las superficies delante del plano se conservan, mientras que las superficies de la pirámide situadas detrás del plano se eliminan.

7.60, se recortaría el segmento de línea que va de  $\mathbf{P}_1$  a  $\mathbf{P}$  y se conservaría la sección de la línea que va de  $\mathbf{P}$  a  $\mathbf{P}_2$ .

Para el caso de poliedros, como por ejemplo la pirámide de la Figura 7.61, aplicamos procedimientos de recorte similares. Primero vemos si el objeto está completamente detrás o completamente delante del plano de recorte. En caso contrario, procesamos la lista de vértices de cada superficie poligonal. Aplicamos métodos de recorte de línea a cada arista sucesiva del polígono, como en el caso de recorte para volúmenes de visualización, con el fin de producir las nuevas listas de vértices de la superficie. Pero en este caso, sólo tenemos que tomar en consideración un único plano de recorte.

Recortar un objeto curvo de acuerdo con un único plano de recorte es más fácil que recortar el objeto según los seis planos de un volumen de visualización, pero seguirá siendo necesario resolver un conjunto de ecuaciones no lineales para localizar las intersecciones, a menos que aproximemos los contornos de la curva mediante secciones lineales.

## 7.12 PLANOS DE RECORTE OPCIONALES EN OpenGL

---

Además de los seis planos de recorte que encierran un volumen de visualización, OpenGL permite la especificación de planos de recorte adicionales en una escena. A diferencia de los planos de recorte del volumen de visualización, que son perpendiculares a alguno de los ejes de coordenadas, estos planos adicionales pueden tener cualquier orientación.

Para especificar un plano de recorte opcional y activar el recorte de acuerdo con dicho plano, se utilizan las instrucciones:

```
glClipPlane (id, planeParameters);
 glEnable (id);
```

El parámetro *id* se utiliza como identificador para un plano de recorte. A este parámetro se le asignan los valores `GL_CLIP_PLANE0`, `GL_CLIP_PLANE1`, etc., hasta un máximo definido por cada implementación. El

plano se define entonces utilizando la matriz de cuatro elementos `planeParameters`, cuyos elementos son los valores de coma flotante y doble precisión de los cuatro parámetros de la ecuación del plano  $A, B, C$  y  $D$ . Para desactivar un plano de recorte activo al que se le haya asignado el identificador `id` se utiliza la instrucción:

```
glDisable (id);
```

Los parámetros del plano  $A, B, C$  y  $D$  se transforman a coordenadas de visualización y se utilizan para comprobar las posiciones en coordenadas de visualización dentro de una escena. Los subsiguientes cambios en los parámetros de visualización o de transformación geométrica no afectan a los parámetros del plano almacenados. Por tanto, si especificamos planos de recorte opcionales antes de especificar las transformaciones geométricas o de visualización, los parámetros del plano almacenados coincidirán con los parámetros que se hayan introducido. Asimismo, puesto que las rutinas de recorte para estos planos se aplican en coordenadas de visualización y no en el espacio de coordenadas normalizadas, el rendimiento de un programa puede degradarse cuando se activan los planos opcionales de recorte.

Todos los puntos que se encuentre «detrás» de un plano de recorte OpenGL activado se eliminarán. Así, una posición  $(x, y, z)$  en coordenadas de visualización será recortada si satisface la condición 7.52.

Hay disponibles seis planos de recorte opcionales en cualquier implementación OpenGL, aunque puede que alguna implementación concreta proporcione más. Podemos ver cuántos planos de recorte opcionales pueden emplearse en una implementación OpenGL concreta utilizando la instrucción,

```
glGetIntegerv (GL_MAX_CLIP_PLANES, numPlanes);
```

El parámetro `numPlanes` es el nombre de una matriz de enteros a la que hay que asignar un valor entero igual al número de planos de recorte opcionales que podemos utilizar. El comportamiento predeterminado para la función `glClipPlane` es que se asigna un valor 0 a los parámetros  $A, B, C$  y  $D$  de todos los planos de recorte opcionales. Asimismo, inicialmente, todos los planos de recorte opcionales están desactivados.

## 7.13 RESUMEN

---

Los procedimientos de visualización para las escenas tridimensionales siguen el enfoque general utilizado en visualización bidimensional. Primero creamos una escena en coordenadas universales, bien a partir de las definiciones de los objetos en coordenadas de modelado o directamente en coordenadas universales. Después, establecemos un sistema de referencias de coordenadas de visualización y transferimos las descripciones de los objetos de coordenadas universales a coordenadas de visualización. A continuación, las descripciones de los objetos se procesan a través de varias rutinas para obtener las coordenadas de dispositivo.

Sin embargo, a diferencia de la visualización bidimensional, el caso de la visualización tridimensional requiere rutinas de proyección para transformar las descripciones de los objetos a un plano de visualización antes de la transformación a coordenadas de dispositivo. Asimismo, las operaciones de visualización tridimensional requieren más parámetros espaciales. Podemos utilizar la analogía de la cámara para describir los parámetros de visualización tridimensionales. Se establece un sistema de referencia de coordenadas de visualización con un punto de vista de referencia (la posición de la cámara), un vector normal al plano de visualización **N** (la dirección del objetivo de la cámara) y un vector vertical **V** (la dirección que apunta hacia arriba en la cámara). Entonces, la posición del plano de visualización se establece a lo largo del eje de visualización  $z$  y las descripciones de los objetos se proyectan sobre este plano. Pueden utilizarse métodos de proyección paralela o proyección en perspectiva para transferir las descripciones de los objetos al plano de visualización.

Las proyecciones paralelas pueden ser ortográficas u oblicuas y pueden especificarse mediante un vector de proyección. Las proyecciones paralelas ortográficas que muestran más de una cara de un objeto se denominan proyecciones axonométricas. Obtenemos una isométrica de un objeto mediante una proyección axono-

métrica que acorte todos los ejes principales según un mismo factor. Las proyecciones oblicuas más comúnmente utilizadas son la perspectiva caballera y la perspectiva cabinet. Las proyecciones en perspectiva de los objetos se obtienen mediante líneas de proyección que se cruzan en el punto de referencia de proyección. Las proyecciones paralelas mantienen las proyecciones de los objetos, mientras que las proyecciones en perspectiva reducen el tamaño de los objetos distantes. Las proyecciones en perspectiva hacen que las líneas paralelas parezcan converger en un punto de fuga, supuesto que las líneas no sean paralelas al plano de visualización. Pueden generarse diagramas de ingeniería y arquitectura con proyecciones en perspectiva de un punto, dos puntos o tres puntos, dependiendo del número de ejes principales que intersecten el plano de visualización. Obtenemos una proyección en perspectiva oblicua cuando la línea que une el punto de referencia de proyección con el centro de la ventana de recorte no es perpendicular al plano de visualización.

Los objetos de una escena tridimensional pueden recortarse de acuerdo con un volumen de visualización, con el fin de eliminar las secciones no deseadas de la escena. La parte superior, la inferior y las laterales del volumen de visualización se forman con planos paralelos a las líneas de proyección y que pasan a través de los lados de la ventana de recorte. Los planos próximo y lejano (también denominados anterior y posterior) se utilizan para crear un volumen de visualización cerrado. Para una proyección paralela, el volumen de visualización es un paralelepípedo. Para una proyección en perspectiva, el volumen de visualización es un frustum. En cualquiera de los dos casos, podemos convertir el volumen de visualización en un cubo normalizado con límites en 0 y 1 para cada coordenada o en -1 y 1. Una serie de eficientes algoritmos de recorte procesan los objetos de la escena de acuerdo con los planos que limitan el volumen de visualización normalizado. El recorte se suele llevar a cabo en los paquetes gráficos en coordenadas homogéneas de cuatro dimensiones, después de las transformaciones de proyección y de normalización del volumen de visualización. Entonces, las coordenadas homogéneas se convierten a coordenadas de proyección cartesianas tridimensionales. También pueden utilizarse planos de recorte adicionales con una orientación arbitraria, con el fin de eliminar partes seleccionadas de una escena o de conseguir efectos especiales.

La biblioteca OpenGL Utility incluye una función de visualización tridimensional para especificar los parámetros de visualización (véase la Tabla 7.1). Esta biblioteca también incluye una función para establecer una transformación de producción de perspectiva simétrica. Hay disponibles otras tres funciones de visualización en la biblioteca básica OpenGL para especificar una proyección ortográfica, una proyección en perspectiva general y planos de recorte opcionales. La Tabla 7.1 resume las funciones de visualización OpenGL analizadas en este capítulo. Además, la tabla incluye algunas otras funciones relacionadas con la visualización.

**TABLA 7.1. RESUMEN DE FUNCIONES DE VISUALIZACIÓN TRIDIMENSIONALES DE OpenGL**

Función	Descripción
gluLookAt	Especifica los parámetros de visualización tridimensional
glOrtho	Especifica los parámetros de una ventana de recorte y de los planos de recorte próximo y lejano para una proyección ortogonal.
gluPerspective	Especifica el ángulo del campo visual de otros parámetros para una proyección en perspectiva simétrica.
glFrustum	Especifica los parámetros para una ventana de recorte y para los planos de recorte próximo y lejano para una proyección en perspectiva (simétrica u oblicua).
glClipPlane	Especifica los parámetros para un plano de recorte opcional.

## REFERENCIAS

---

Puede encontrar un análisis de los algoritmos de visualización tridimensional y de recorte en Weiler y Atherton (1977), Weiler (1980), Cyrus y Beck (1978) y Liang y Barsky (1984). Los algoritmos de recorte en coordenadas homogéneas se describen en Blinn y Newell (1978), Riesenfeld (1981) y Blinn (1993, 1996 y 1998). También puede encontrar un análisis de diversas técnicas de programación para visualización tridimensional en Glassner (1990), Arvo (1991), Kirk (1992), Heckbert (1994) y Paeth (1995).

Puede encontrar el listado completo de funciones de visualización tridimensionales OpenGL en Shreiner (2000). Si está interesado en consultar ejemplos de programación OpenGL que utilicen visualización tridimensional, le recomendamos Woo, Neider, Davis y Shreiner (1999). También puede encontrar ejemplos de programación adicionales en el sitio web tutorial de Nate Robins: <http://www.cs.utah.edu/~narobins/opengl.html>.

## EJERCICIOS

---

- 7.1 Escriba un procedimiento para especificar la matriz que transforme los puntos en coordenadas universales a coordenadas de visualización tridimensionales, dados  $\mathbf{P}_0$ ,  $\mathbf{N}$  y  $\mathbf{V}$ . El vector vertical puede encontrarse en cualquier dirección que no sea paralela a  $\mathbf{N}$ .
- 7.2 Escriba un procedimiento para transformar los vértices de un poliedro a coordenadas de proyección utilizando una proyección paralela con cualquier vector de proyección especificado.
- 7.3 Escriba un procedimiento para obtener diferentes vistas de proyección paralela de un poliedro aplicando primero una rotación especificada.
- 7.4 Escriba un procedimiento para realizar una proyección en perspectiva de un punto de un objeto.
- 7.5 Escriba un procedimiento para realizar una proyección en perspectiva de dos puntos de un objeto.
- 7.6 Desarrolle una rutina para realizar una proyección en perspectiva de tres puntos de un objeto.
- 7.7 Escriba una rutina para convertir un frustum de proyección en perspectiva en un paralelepípedo rectangular.
- 7.8 Modifique el algoritmo de recorte de líneas bidimensional de Cohen-Sutherland para recortar líneas tridimensionales según el volumen de visualización simétrico normalizado.
- 7.9 Modifique el algoritmo de recorte de líneas bidimensional de Liang-Barsky para recortar líneas tridimensionales según un paralelepípedo regular especificado.
- 7.10 Modifique el algoritmo de recorte de líneas bidimensional de Liang-Barsky para recortar un poliedro de acuerdo con un paralelepípedo regular especificado.
- 7.11 Escriba una rutina para realizar el recorte de líneas en coordenadas homogéneas.
- 7.12 Desarrolle un algoritmo para recortar un poliedro según un frustum definido. Compare las operaciones necesarias en este algoritmo con las que harían falta en un algoritmo que recortara el poliedro de acuerdo con un paralelepípedo regular.
- 7.13 Amplíe el algoritmo de recorte de polígonos de Sutherland-Hodgman para recortar un poliedro convexo de acuerdo con un volumen de visualización simétrico normalizado.
- 7.14 Escriba una rutina para implementar el ejercicio anterior.
- 7.15 Escriba una rutina para realizar el recorte de poliedros en coordenadas homogéneas.
- 7.16 Modifique el ejemplo de programa de la Sección 7.10 para permitir a un usuario especificar una vista para la parte frontal o posterior del cuadrado.
- 7.17 Modifique el ejemplo de programa de la Sección 7.10 para permitir que el usuario introduzca los parámetros de visualización en perspectiva.

- 7.18 Modifique el ejemplo de programa de la Sección 7.10 para generar una vista de cualquier poliedro de entrada.
- 7.19 Modifique el programa del ejercicio anterior para generar una vista del poliedro utilizando una proyección ortográfica.
- 7.20 Modifique el programa del ejercicio anterior para generar una vista del poliedro utilizando una proyección paralela oblicua.

# Representaciones de objetos tridimensionales



Una escena de una habitación generada por computador que contiene objetos modelados con varias representaciones tridimensionales. (Cortesía de Autodesk, Inc.).

- |             |   |             |   |
|-------------|---|-------------|---|
| <b>8.1</b>  | Poliedros   | <b>8.16</b> | Conversión entre representaciones mediante <i>splines</i>     |
| <b>8.2</b>  | Funciones para poliedros en OpenGL                                | <b>8.17</b> | Visualización de curvas y superficies mediante <i>splines</i> |
| <b>8.3</b>  | Superficies curvadas  | <b>8.18</b> | Funciones de <i>splines</i> de aproximación en OpenGL         |
| <b>8.4</b>  | Superficies cuádricas   | <b>8.19</b> | Representaciones de barrido                                   |
| <b>8.5</b>  | Supercuádricas  | <b>8.20</b> | Métodos de geometría de sólidos constructiva                  |
| <b>8.6</b>  | Funciones OpenGL para superficies cuádricas y superficies cúbicas | <b>8.21</b> | Árboles octales   |
| <b>8.7</b>  | Objetos sin forma   | <b>8.22</b> | Árboles BSP   |
| <b>8.8</b>  | Representaciones mediante <i>splines</i>                          | <b>8.23</b> | Métodos de geometría fractal                                  |
| <b>8.9</b>  | Métodos de interpolación mediante <i>splines</i> cúbicos          | <b>8.24</b> | Gramáticas de formas y otros métodos procedimentales          |
| <b>8.10</b> | Curvas mediante <i>splines</i> de Bézier                          | <b>8.25</b> | Sistemas de partículas  |
| <b>8.11</b> | Superficies de Bézier   | <b>8.26</b> | Modelado basado en las características físicas                |
| <b>8.12</b> | Curvas mediante <i>splines</i> B                                  | <b>8.27</b> | Visualización de conjuntos de datos                           |
| <b>8.13</b> | Superficies mediante <i>splines</i> B                             | <b>8.28</b> | Resumen   |
| <b>8.14</b> | <i>Splines</i> beta   |             |   |
| <b>8.15</b> | <i>Splines</i> racionales   |             |   |

Las escenas gráficas pueden contener muchas clases diferentes de objetos y superficies de materiales: árboles, flores, nubes, rocas, agua, ladrillos, tableros de madera, goma, papel, mármol, acero, cristal, plástico y tela, por mencionar unos pocos. Por tanto, no puede sorprender que no haya un único método que podamos utilizar para describir objetos que incluyan todas las características de estos materiales diferentes.

Las superficies de polígonos y de cuádricas proporcionan descripciones precisas de objetos euclídeos simples, tales como los poliedros y los elipsoides; las superficies mediante *splines* y las técnicas de la geometría sólida constructiva son útiles para diseñar alas de avión, ruedas dentadas y otras estructuras de ingeniería con superficies curvadas; los métodos procedimentales, tales como las construcciones fractales y los sistemas de partículas, nos permiten modelar las características del terreno, nubes, prados de césped y otros objetos naturales; los métodos de modelado basados en la Física, que utilizan sistemas de fuerzas de interacción, se pueden utilizar para describir el comportamiento no rígido de una prenda de ropa o una porción de gelatina; las codificaciones mediante árboles octales se utilizan para representar características internas de objetos, tales como las obtenidas a partir de imágenes CT médicas; y las visualizaciones mediante superficies de nivel, los sombreados de volúmenes y otras técnicas de visualización se aplican a conjuntos de datos discretos tridimensionales para obtener representaciones visuales de los datos.

Las técnicas de representación de objetos sólidos se dividen a menudo en dos grandes categorías, aunque no todas las representaciones pertenecen con claridad a una de estas dos categorías. Las **representaciones por límites** describen un objeto tridimensional como un conjunto de superficies que separan el interior del objeto de su entorno. Los ejemplos habituales de representaciones por límites son las facetas de polígonos y parches con *splines*. Las **representaciones de particionamiento del espacio** se utilizan para describir propiedades internas, particionando la región del espacio que contiene un objeto en un conjunto de sólidos pequeños, contiguos y que no se superponen (habitualmente cubos). Una descripción habitual de particionamiento del espacio

cio de un objeto tridimensional es una representación mediante un árbol octal. En este capítulo consideramos las características de las distintas técnicas de representación y cómo se utilizan en aplicaciones gráficas por computadora.

## 8.1 POLIEDROS

---

La representación por límites de un objeto gráfico tridimensional que se utiliza más habitualmente consiste en un conjunto de polígonos que encierra el interior del objeto. Muchos sistemas gráficos almacenan todas las descripciones de los objetos como conjuntos de polígonos. Esto simplifica y acelera el sombreado de las superficies y la visualización de los objetos, ya que todas las superficies se describen con ecuaciones lineales. Por esta razón, las descripciones de polígonos se denominan a menudo *objetos gráficos estándar*. En algunos casos, una representación poligonal es la única disponible, pero muchos paquetes también permiten describir las superficies de los objetos con otras técnicas, tales como superficies mediante *splines*, que habitualmente se convierten en representaciones poligonales para su procesamiento en la *pipeline* de visualización.

Para describir un objeto como un conjunto de facetas poligonales, damos la lista de coordenadas de los vértices de cada polígono sobre la superficie del objeto. Las coordenadas de los vértices y la información de las aristas de las partes de la superficie se almacenan a continuación en tablas (Sección 3.15), junto con otra información tal como el vector normal a la superficie para cada polígono. Algunos paquetes gráficos proporcionan subrutinas para la generación de mallas de polígonos como un conjunto de triángulos o de cuadriláteros. Esto nos permite describir una gran parte de la superficie límite de un objeto, o incluso toda la superficie, con una única orden. Algunos paquetes también proporcionan subrutinas para mostrar formas comunes, tales como un cubo, una esfera, o un cilindro, representados con superficies de polígonos. Los sistemas gráficos sofisticados utilizan sombreadores (renderers) rápidos de polígonos implementados en hardware que tienen la capacidad de mostrar un millón o más de polígonos (habitualmente triángulos) sombreados por segundo, en los que se incluye la aplicación de texturas a su superficie y de efectos especiales de iluminación.

## 8.2 FUNCIONES PARA POLIEDROS EN OpenGL

---

Disponemos de dos métodos para especificar polígonos en un programa con OpenGL. Utilizando las primitivas para polígonos estudiadas en la Sección 3.16, podemos generar una gran variedad de formas poliédricas y mallas de superficie. Además, podemos utilizar las funciones de GLUT para mostrar los cinco poliedros regulares.

### Funciones de áreas de relleno de polígonos en OpenGL

Un conjunto de parches de polígonos de una parte de la superficie de un objeto, o una descripción completa de un poliedro, se puede proporcionar utilizando las constantes de primitivas de OpenGL `GL_POLYGON`, `GL_TRIANGLES`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`, `GL_QUADS`, y `GL_QUAD_STRIP`. Por ejemplo, podríamos teselar la superficie lateral (axial) de un cilindro utilizando una tira de cuadriláteros. De forma similar, todas las caras de un paralelogramo se podrían describir con un conjunto de rectángulos, y todas las caras de una pirámide triangular se podrían especificar utilizando un conjunto de superficies triangulares conectadas.

### Funciones para poliedros regulares de GLUT

Algunas formas estándar, los cinco poliedros regulares, están predefinidos mediante subrutinas en la biblioteca GLUT. Estos poliedros, también llamados sólidos platónicos, se distinguen por el hecho de que todas las caras de cualquier poliedro regular son polígonos regulares idénticos. Por tanto, todas las aristas de un polie-

dro regular son iguales, todos los ángulos entre las aristas son iguales y todos los ángulos entre las caras son iguales. A los poliedros se les asigna un nombre de acuerdo con el número de caras en cada uno de los sólidos. Los cinco poliedros regulares son el tetraedro regular (o pirámide triangular, con 4 caras), el hexaedro regular (o cubo, con 6 caras), el octaedro regular (8 caras), el dodecaedro regular (12 caras) y el icosaedro regular (20 caras).

GLUT proporciona diez funciones GLUT para generar estos sólidos: cinco de las funciones producen objetos de malla de alambre y cinco muestran las facetas de los poliedros como áreas de relleno sombreadas. Las características de la superficie mostrada de las áreas de relleno se determinan mediante las propiedades del material y las condiciones de iluminación que se establecen para una escena. Cada poliedro regular se describe en coordenadas de modelado, de modo que cada uno esté centrado en el origen del sistema de coordenadas universales.

Obtenemos la pirámide triangular, regular y de cuatro caras utilizando cualquiera de las dos funciones siguientes:

```
glutWireTetrahedron ( );
o
glutSolidTetrahedron ( );
```

Este poliedro se genera con su centro en el origen del sistema de coordenadas universales y con un radio (distancia desde el centro del tetraedro a cualquier vértice) igual a  $\sqrt{3}$ .

El hexaedro regular de seis caras (cubo) se visualiza con:

```
glutWireCube (edgeLength);
o
glutSolidCube (edgeLength);
```

Al parámetro `edgeLength` se le puede asignar cualquier valor en punto flotante de doble precisión y positivo, y el cubo está centrado en el origen de coordenadas.

Para visualizar el octaedro regular de ocho caras, invocamos cualquiera de los siguientes comandos:

```
glutWireOctahedron ( );
o
glutSolidOctahedron ( );
```

Este poliedro tiene caras triangulares equiláteras, y el radio (distancia desde el centro del octaedro situado en el origen de coordenadas hasta cualquier vértice) es 1.0.

El dodecaedro regular de doce caras, centrado en el origen del sistema de coordenadas universales se genera con:

```
glutWireDodecahedron ( );
o
glutSolidDodecahedron ( );
```

Cada cara de este poliedro es un pentágono.

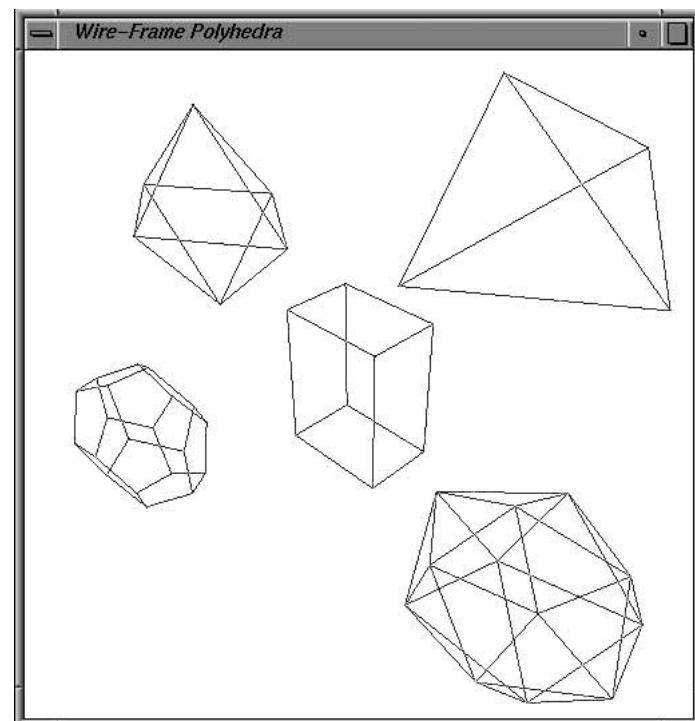
Y las siguientes funciones generan el icosaedro regular de veinte caras.

```
glutWireIcosahedron ( );
o
glutSolidIcosahedron ( );
```

El radio (distancia desde el centro del poliedro, situado en el origen de coordenadas, hasta cualquier vértice) predeterminado del icosaedro es 1.0 y cada cara es un triángulo equilátero.

## Ejemplo de programa de poliedros con GLUT

Utilizando las funciones de GLUT para los sólidos platónicos, el siguiente programa genera una visualización en perspectiva, en malla de alambre y transformada de estos poliedros. Los cinco sólidos se encuentran dentro de una ventana de visualización (Figura 8.1).



**FIGURA 8.1.** Una vista en perspectiva de los cinco poliedros de GLUT, cambiados de escala y posicionados dentro de una ventana de visualización por el procedimiento `displayWirePolyhedra`.

```
#include <GL/glut.h>

GLsizei winWidth = 500, winHeight = 500;      //Tamaño inicial de la ventana de
                                                //visualización.

void init (void)
{
    glClearColor (1.0, 1.0, 1.0, 0.0); // Ventana de visualización en blanco.
}

void displayWirePolyhedra (void)
{
    glClear (GL_COLOR_BUFFER_BIT); // Borra la ventana de visualización.

    glColor3f (0.0, 0.0, 1.0);      // Establece el color de las líneas en azul.

    /* Establece la transformación de visionado. */
    gluLookAt (5.0, 5.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);

    /* Cambia de escala un cubo y lo muestra como paralelepípedo alámbrico. */
    glScalef (1.5, 2.0, 1.0);
}
```

```
glutWireCube (1.0);

/* Cambia de escala, traslada, y muestra un dodecaedro alámbrico. */
glScalef (0.8, 0.5, 0.8);
glTranslatef (-6.0, -5.0, 0.0);
glutWireDodecahedron ( );

/* Traslada y muestra un tetraedro en modelo alámbrico. */
glTranslatef (8.6, 8.6, 2.0);
glutWireTetrahedron ( );

/* Traslada y visualiza un octaedro en modelo alámbrico. */
glTranslatef (-3.0, -1.0, 0.0);
glutWireOctahedron ( );

/* Cambia de escala, traslada y muestra un icosaedro en modelo alámbrico. */
glScalef (0.8, 0.8, 1.0);
glTranslatef (4.3, -2.0, 0.5);
glutWireIcosahedron ( );

glFlush ( );
}

void winReshapeFcn (GLint newWidth, GLint newHeight)
{
    glViewport (0, 0, newWidth, newHeight);

    glMatrixMode (GL_PROJECTION);
    glFrustum (-1.0, 1.0, -1.0, 1.0, 2.0, 20.0);

    glMatrixMode (GL_MODELVIEW);

    glClear (GL_COLOR_BUFFER_BIT);
}

void main (int argc, char** argv)
{
    glutInit (&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    glutInitWindowPosition (100, 100);
    glutInitWindowSize (winWidth, winHeight);
    glutCreateWindow ("Poliedros en modelo alámbrico");

    init ();
    glutDisplayFunc (displayWirePolyhedra);
    glutReshapeFunc (winReshapeFcn);

    glutMainLoop ( );
}
```

## 8.3 SUPERFICIES CURVADAS

Las ecuaciones de los objetos con límites curvos se pueden expresar en forma paramétrica o en forma no paramétrica. El Apéndice A proporciona un resumen y una comparación de las representaciones paramétricas y no paramétricas. Entre los múltiples objetos que son útiles a menudo en las aplicaciones gráficas se pueden incluir las superficies cuádricas, las supercuádricas, las funciones polinómicas y exponenciales, y las superficies mediante *splines*. Estas descripciones de objetos de entrada se teselan habitualmente para producir aproximaciones de las superficies con mallas de polígonos.

## 8.4 SUPERFICIES CUÁDRICAS

Una clase de objetos que se utiliza frecuentemente es la de las *superficies cuádricas*, que se describen con ecuaciones de segundo grado (cuádricas). Entre ellas se incluyen las esferas, los elipsoides, los toros, los paraboloides y los hiperboloides. Las superficies cuádricas, particularmente las esferas y los elipsoides, son elementos comunes en las escenas gráficas, y las subrutinas para generar estas superficies están disponibles a menudo en los paquetes gráficos. También, las superficies cuadráticas se pueden producir con representaciones mediante *splines* racionales.

### Esfera

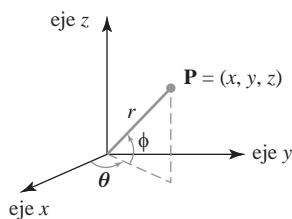
En coordenadas cartesianas, una superficie esférica de radio  $r$  centrada en el origen de coordenadas se define como el conjunto de puntos  $(x, y, z)$  que satisface la ecuación:

$$x^2 + y^2 + z^2 = r^2 \quad (8.1)$$

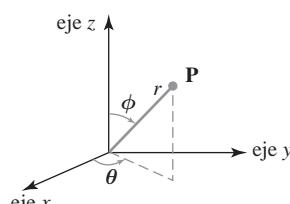
También podemos describir la superficie esférica de forma paramétrica, utilizando los ángulos de la latitud y la longitud (Figura 8.2):

$$\begin{aligned} x &= r \cos \phi \cos \theta, & -\pi/2 \leq \phi \leq \pi/2 \\ y &= r \cos \phi \sin \theta, & -\pi \leq \theta \leq \pi \\ z &= r \sin \phi \end{aligned} \quad (8.2)$$

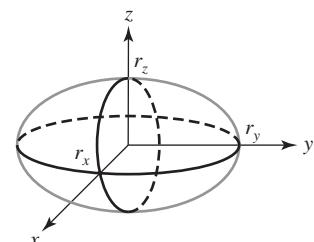
La representación paramétrica de las Ecuaciones 8.2 proporciona un rango simétrico para los parámetros angulares  $\theta$  y  $\phi$ . Como alternativa, podríamos escribir las ecuaciones paramétricas utilizando coordenadas esféricas estándar, en las que el ángulo  $\phi$  se especifica como colatitud (Figura 8.3). Entonces,  $\phi$  se define dentro del rango  $0 \leq \phi \leq \pi$  y  $\theta$  se toma a menudo en el rango  $0 \leq \theta \leq 2\pi$ . Podríamos también establecer la representación utilizando parámetros  $u$  y  $v$  definidos sobre el rango que varía entre 0 y 1, haciendo las sustituciones  $\phi = \pi u$  y  $\theta = 2\pi v$ .



**FIGURA 8.2.** Coordenadas paramétricas  $(r, \theta, \phi)$  de la superficie de una esfera de radio  $r$ .



**FIGURA 8.3.** Parámetros de las coordenadas esféricas  $(r, \theta, \phi)$ , utilizando la colatitud para el ángulo  $\phi$ .



**FIGURA 8.4.** Un elipsoide con radios  $r_x$ ,  $r_y$  y  $r_z$ , centrado en el origen de coordenadas.

## Elipsoide

Una superficie elipsoidal se puede describir como una ampliación de la superficie esférica, en la que el radio en tres direcciones perpendiculares entre sí puede tener valores diferentes (Figura 8.4). La representación cartesiana de los puntos de la superficie de un elipsoide centrado en el origen es:

$$\left(\frac{x}{r_x}\right)^2 + \left(\frac{y}{r_y}\right)^2 + \left(\frac{z}{r_z}\right)^2 = 1 \quad (8.3)$$

Una representación paramétrica de un elipsoide en función del ángulo de la latitud  $\phi$  y del ángulo de la longitud  $\theta$  de la Figura 8.2 es:

$$\begin{aligned} x &= r_x \cos \phi \cos \theta, & -\pi/2 \leq \phi \leq \pi/2 \\ y &= r_y \cos \phi \sin \theta, & -\pi \leq \theta \leq \pi \\ z &= r_z \sin \phi \end{aligned} \quad (8.4)$$

## Toro

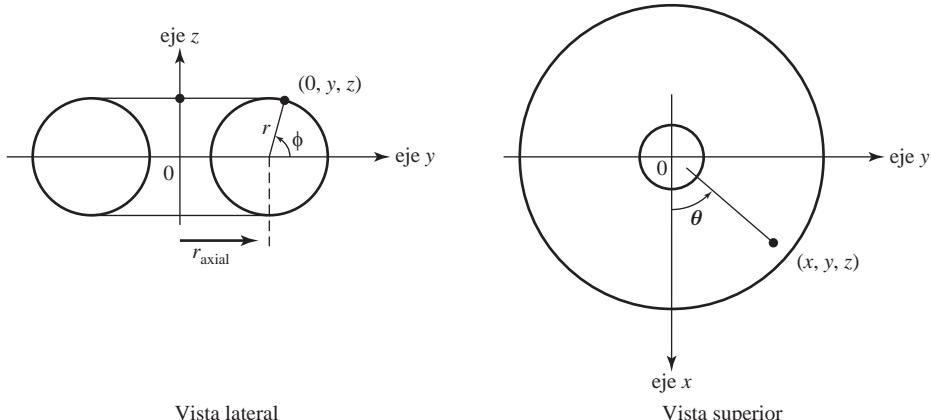
Un objeto con forma de donut se denomina toro. Muy a menudo se describe como la superficie generada al hacer girar un círculo o una elipse alrededor de un eje coplanario que es externo a la cónica. Los parámetros de definición de un toro son entonces la distancia del centro de la cónica al eje de rotación y las dimensiones de la cónica. En la Figura 8.5 se muestra un toro generado por la rotación de un círculo de radio  $r$  en el plano  $yz$  alrededor del eje  $z$ . Con el centro del círculo en el eje  $y$ , el radio axial,  $r_{\text{axial}}$ , del toro resultante es igual a la distancia en la dirección del eje  $y$  al centro del círculo desde el eje  $z$  (eje de rotación). El radio de la sección recta del toro es el radio del círculo generatriz.

La ecuación del círculo de la sección recta que se muestra en la vista lateral de la Figura 8.5 es:

$$(y - r_{\text{axial}})^2 + z^2 = r^2$$

Al hacer girar este círculo alrededor del eje  $z$  se genera el toro cuya superficie se describe con la ecuación cartesiana:

$$\left(\sqrt{x^2 + y^2} - r_{\text{axial}}\right)^2 + z^2 = r^2 \quad (8.5)$$



**FIGURA 8.5.** Un toro centrado en el origen de coordenadas con una sección recta circular y con el eje del toro según el eje  $z$ .

Y las ecuaciones paramétricas correspondientes del toro con una sección recta circular son:

$$\begin{aligned}x &= (r_{\text{axial}} + r \cos \phi) \cos \theta, & -\pi \leq \phi \leq \pi \\y &= (r_{\text{axial}} + r \cos \phi) \sin \theta, & -\pi \leq \theta \leq \pi \\z &= r \sin \phi\end{aligned}\quad (8.6)$$

También podríamos generar un toro haciendo girar una elipse, en lugar de un círculo, alrededor del eje  $z$ . En el caso de una elipse situada en el plano  $yz$  con semidiámetro principal  $r_y$  y semidiámetro secundario referenciados como  $r_z$ , podemos escribir la ecuación de la elipse como:

$$\left(\frac{y - r_{\text{axial}}}{r_y}\right)^2 + \left(\frac{z}{r_z}\right)^2 = 1$$

en la que  $r_{\text{axial}}$  es la distancia según el eje  $y$  desde el eje de rotación  $z$  al centro de la elipse. De este modo, se genera un toro que se puede describir con la ecuación cartesiana,

$$\left(\frac{\sqrt{x^2 + y^2} - r_{\text{axial}}}{r_y}\right)^2 + \left(\frac{z}{r_z}\right)^2 = 1 \quad (8.7)$$

La representación paramétrica correspondiente del toro con una sección recta elíptica es

$$\begin{aligned}x &= (r_{\text{axial}} + r_y \cos \phi) \cos \theta, & -\pi \leq \phi \leq \pi \\y &= (r_{\text{axial}} + r_y \cos \phi) \sin \theta, & -\pi \leq \theta \leq \pi \\z &= r_z \sin \phi\end{aligned}\quad (8.8)$$

Son posibles otras variaciones de las ecuaciones anteriores del toro. Por ejemplo, podríamos generar una superficie toroidal haciendo girar un círculo o una elipse siguiendo una trayectoria elíptica alrededor del eje de rotación.

## 8.5 SUPERCUÁDRICAS

---

Esta clase de objetos es una generalización de las cuádricas. Las **supercuádricas** se crean incorporando parámetros adicionales a las ecuaciones de las cuádricas, para proporcionar una mayor flexibilidad en el ajuste de las formas de los objetos. Se añade un parámetro adicional a las ecuaciones de una curva y se utilizan dos parámetros adicionales en las ecuaciones de las superficies.

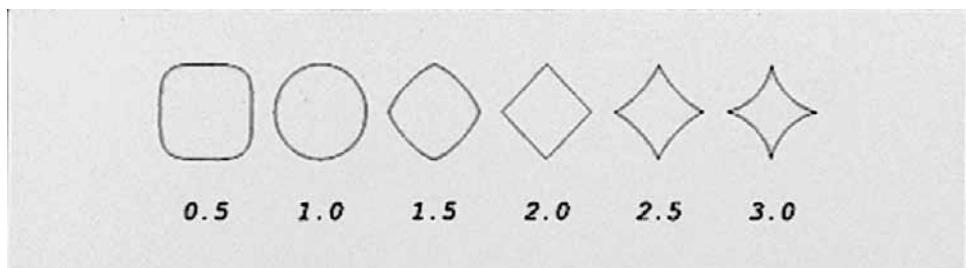
### Superelipse

Obtenemos la representación cartesiana de una superelipse a partir de la ecuación de una elipse permitiendo que el exponente de los términos  $x$  e  $y$  sea variable. Un modo de hacer esto es escribir la ecuación cartesiana de la superelipse en la forma:

$$\left(\frac{x}{r_x}\right)^{2/s} + \left(\frac{y}{r_y}\right)^{2/s} = 1 \quad (8.9)$$

en la que al parámetro  $s$  se le puede asignar cualquier valor real. Cuando  $s=1$ , tenemos una elipse ordinaria.

Las ecuaciones paramétricas correspondientes a la superelipse de la Ecuación 8.9 se pueden expresar como



**FIGURA 8.6.** Superelipses dibujadas con valores del parámetro  $s$  variando desde 0.5 a 3.0 y con  $r_x = r_y$ .

$$\begin{aligned} x &= r_x \cos^s \theta, & -\pi \leq \theta \leq \pi \\ y &= r_y \sin^s \theta \end{aligned} \quad (8.10)$$

La Figura 8.6 muestra las formas de superelipses que se pueden generar utilizando varios valores del parámetro  $s$ .

### Superelipsoide

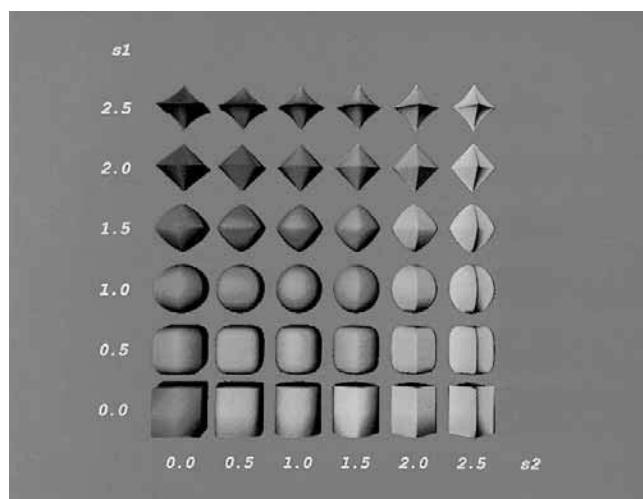
Una representación cartesiana de un superelipsoide se obtiene a partir de la ecuación de un elipsoide incorporando dos parámetros exponenciales:

$$\left[ \left( \frac{x}{r_x} \right)^{2/s_2} + \left( \frac{y}{r_y} \right)^{2/s_2} \right]^{s_2/s_1} + \left( \frac{z}{r_z} \right)^{2/s_1} = 1 \quad (8.11)$$

Con  $s_1 = s_2 = 1$ , tenemos un elipsoide ordinario.

Podemos a continuación escribir la representación paramétrica correspondiente al superelipsoide de la Ecuación 8.11 como,

$$\begin{aligned} x &= r_x \cos^{s_1} \phi \cos^{s_2} \theta, & -\pi/2 \leq \phi \leq \pi/2 \\ y &= r_y \cos^{s_1} \phi \sin^{s_2} \theta, & -\pi \leq \theta \leq \pi \\ z &= r_z \sin^{s_1} \phi \end{aligned} \quad (8.12)$$



**FIGURA 8.7.** Superelipsoides dibujados con valores de los parámetros  $s_1$  y  $s_2$  variando desde 0.0 a 2.5 y con  $r_x = r_y = r_z$ .

La Figura 8.7 muestra las formas de los superelipsoides que se pueden generar utilizando varios valores de los parámetros  $s_1$  y  $s_2$ . Estas y otras formas de supercuádricas se pueden combinar para crear estructuras más complejas, tales como descripciones de muebles, tornillos roscados y otros artículos de ferretería.

## 8.6 FUNCIONES OpenGL PARA SUPERFICIES CUÁDRICAS Y SUPERFICIES CÚBICAS

---

Una esfera y un gran número de otros objetos de superficies cuádricas tridimensionales se pueden representar utilizando funciones que están incluidas en el conjunto de herramientas de OpenGL Utility Toolkit (GLUT) y en OpenGL Utility (GLU). Además, GLUT posee una función para mostrar la forma de una tetera que está definida con parches de superficie bicúbicos. Las funciones de GLUT, que son fácilmente incorporables a un programa de aplicación, tienen cada una dos versiones. Una versión de cada función muestra una superficie alámbrica y la otra versión muestra la superficie como un conjunto de parches poligonales coloreados. Con las funciones de GLUT podemos mostrar una esfera, un cono, un toro o una tetera. Las funciones de GLU para superficies cuádricas son un poquito más farragosas de configurar, pero proporcionan más opciones. Con las funciones de GLU, podemos representar una esfera, un cilindro, un cilindro con tapas, un cono, una corona circular (o disco hueco) y un sector de corona circular (o disco).

### Funciones para superficies cuádricas de GLUT

Generamos una esfera con GLUT con cualquiera de las dos funciones siguientes:

```
glutWireSphere (r, nLongitudes, nLatitudes);
o
glutSolidSphere (r, nLongitudes, nLatitudes);
```

en las que el radio de la esfera se determina con el número en punto flotante de doble precisión que asignemos al parámetro *r*. Los parámetros *nLongitudes* y *nLatitudes* se utilizan para seleccionar el número entero de líneas de longitud y de latitud que se utilizarán para aproximar la superficie esférica mediante una malla de cuadriláteros. Las aristas de los parches de superficie de cuadriláteros son aproximaciones de líneas rectas de las líneas de longitud y de latitud. La esfera se define con coordenadas de modelado, centrada en el origen de coordenadas universal con su eje polar según la dirección del eje *z*.

Un cono con GLUT se obtiene con:

```
glutWireCone (rBase, height, nLongitudes, nLatitudes);
o
glutSolidCone (rBase, height, nLongitudes, nLatitudes);
```

Establecemos los valores en punto flotante de doble precisión para el radio de la base del cono y la altura del cono, utilizando los parámetros *rbase* y *height*, respectivamente. Como en el caso de la esfera, a los parámetros *nLongitudes* y *nLatitudes* se les asignan valores enteros que especifican el número de líneas ortogonales de superficie en la aproximación mediante una malla de cuadriláteros. Una línea de longitud de un cono es un segmento de línea recta sobre la superficie del cono, desde el vértice hacia la base, que se encuentre en un plano que contenga al eje del cono. Cada línea de longitud se visualiza como un conjunto de segmentos de línea recta alrededor de la circunferencia de un círculo, en la superficie del cono que es paralelo a la base del cono y que se encuentre en un plano perpendicular al eje del cono. El cono se describe con coordenadas de modelado, con el centro de la base en el origen de coordenadas universal y con el eje del cono en la dirección del eje *z*.

Las visualizaciones alámbricas o con superficie sombreada de un toro con sección recta circular se generan mediante:

```

glutWireTorus  (rCrossSection, rAxial, nConcentrics, nRadialSlices);

o

glutSolidTorus  (rCrossSection, rAxial, nConcentrics, nRadialSlices);

```

El toro que se obtiene con estas subrutinas de GLUT se puede describir como la superficie generada mediante la rotación de un círculo de radio `rCrossSection` alrededor del eje `z` coplanario, en la que la distancia del centro del círculo al eje `z` es `rAxial` (Sección 8.4). Seleccionamos un tamaño del toro utilizando valores en punto flotante de doble precisión para estos radios en las funciones de GLUT. Y el tamaño de los cuadriláteros de la malla de aproximación de la superficie del toro se establece con los valores enteros de los parámetros `nConcentrics` y `nRadialSlices`. El parámetro `nConcentrics` especifica el número de círculos concéntricos (con centro en el eje `z`) que se deben utilizar en la superficie del toro y el parámetro `nRadialSlices` especifica el número de secciones radiales de la superficie del toro. Estos dos parámetros hacen referencia al número de líneas de la cuadricula ortogonal sobre la superficie del toro, que se visualizan como segmentos de línea recta (los límites de los cuadriláteros) entre las intersecciones. El toro mostrado está centrado en el origen de coordenadas universal y tiene su eje en la dirección del eje `z` universal.

### Función de GLUT para la generación de una tetera con una superficie cúbica

Durante el desarrollo inicial de los métodos de los gráficos por computadora, se construyeron tablas de datos de mallas poligonales que describen varios objetos tridimensionales que se pudieron utilizar para probar las técnicas de sombreado. Entre estos objetos se incluyen las superficies de un automóvil Volkswagen y una tetera, que fueron desarrollados en la universidad de UTA. El conjunto de datos de la tetera de UTA, como fue construido por Martin Newell en 1975, contiene 306 vértices, que definen 32 parches de superficies de Bézier bicúbicas (Sección 8.11). Ya que la determinación de las coordenadas de la superficie de un objeto complejo requiere un tiempo, estos conjuntos de datos, sobre todo la malla de la superficie de la tetera, llegaron a ser profusamente utilizados.

Podemos visualizar la tetera, como una malla de cerca de mil parches de superficies bicúbicas, utilizando cualquiera de las dos funciones siguientes de GLUT:

```

glutWireTeapot  (size);

o

glutSolidTeapot  (size);

```

La superficie de la tetera se genera utilizando funciones de curvas de Bézier de OpenGL (Sección 8.11). El parámetro `size` establece el valor en punto flotante de doble precisión del radio máximo de la vasija de la tetera. La tetera está centrada en el origen de coordenadas UNIVERSAL y su eje vertical lo tiene en la dirección del eje `y`.

### Funciones para la generación de superficies cuádricas de GLU

Para generar una superficie cuádrica utilizando funciones de GLU, necesitamos asignar un nombre a la cuádrica, activar el sombreador de cuádricas de GLU y especificar los valores de los parámetros de la superficie. Además, podemos establecer otros parámetros para controlar la apariencia de una superficie cuádrica con GLU.

Las siguientes líneas de código muestran la secuencia básica de llamadas a funciones para mostrar una esfera alámbrica centrada en origen de coordenadas universal.

```

GLUquadricObj  *sphere1;

Sphere1 = gluNewQuadric  ( );

```

```
gluQuadricDrawStyle (sphere1, GLU_LINE);
gluSphere (sphere1, r, nLongitudes, nLatitudes);
```

En la primera línea de código se define un nombre para el objeto de tipo cuádrica. En este ejemplo, hemos elegido el nombre `sphere1`. Este nombre se utiliza después en otras funciones de GLU para hacer referencia a esta superficie cuádrica particular. A continuación, se activa el sombreador de cuádricas con la función `gluNewQuadric`, entonces se selecciona el modo de visualización `GLU_LINE` para `sphere1` con el comando `gluQuadricDrawStyle`. Por tanto, la esfera se muestra en su modelo alámbrico con un segmento de línea recta entre cada par de vértices de la superficie. Al parámetro `r` se le asigna un valor de doble precisión para usarlo como radio de la esfera. La esfera se divide en un conjunto de caras poligonales mediante líneas de longitud y de latitud equiespaciadas. Especificamos el número entero de líneas de longitud y de líneas de latitud como valores de los parámetros `nLongitudes` y `nLatitudes`.

Hay disponibles otros tres modos de visualización de superficies cuádricas con GLU. Utilizando la constante simbólica `GLU_POINT` en `gluQuadricDrawStyle`, visualizamos una superficie cuádrica como un dibujo de puntos. En el caso de la esfera, se visualiza un punto en cada vértice de la superficie determinado por la intersección de una línea de longitud y una línea de latitud. Otra opción es la constante simbólica `GLU_SILHOUTTE`. Ésta produce una visualización alámbrica eliminando las aristas comunes entre dos caras poligonales coplanares. Con la constante simbólica `GLU_FILL`, visualizamos los parches de polígonos como áreas de relleno sombreadas.

Utilizando la misma secuencia básica de comandos generamos visualizaciones de las otras primitivas para superficies cuádricas con GLU. Para producir una vista de un cono, un cilindro, o cilindro con tapas, reemplazamos la función `gluSphere` por

```
gluCylinder (quadricName, rBase, rTop, height, nLongitudes, nLatitudes) ;
```

La base de este objeto se encuentra en el plano  $xy$  ( $z=0$ ) y su eje es el eje  $z$ . Utilizando el parámetro `rBase` asignamos un valor de doble precisión al radio de la base de esta superficie cuádrica y utilizando el parámetro `rTop` al radio de la tapa superior. Si `stop = 0.0`, obtenemos un cono; si `rTop = rBase` obtenemos un cilindro. En caso contrario, se obtiene un cilindro con tapas. Al parámetro `height` se le asigna un valor de doble precisión de la altura y la superficie queda dividida en un número de líneas verticales y horizontales equiespaciadas que viene determinado por los valores enteros asignados a los parámetros `nLongitudes` y `nLatitudes`.

Una corona circular plana o disco sólido se visualiza en el plano  $xy$  ( $z=0$ ) y centrado en el origen de coordenadas universal con:

```
gluDisk (ringName, rInner, rOuter, nRadii, nRings);
```

Con los parámetros `rInner` y `rOuter` establecemos los valores de doble precisión del radio interior y del radio exterior. Si `rInner = 0`, el disco está completamente lleno. De lo contrario, se visualiza con un agujero concéntrico en el centro del disco. La superficie del disco está dividida en un conjunto de facetas mediante los parámetros enteros `nRadii` y `nRings`, que especifican el número de rodajas radiales que hay que utilizar en la teselación y el número de anillos circulares concéntricos, respectivamente. La orientación del anillo se define con respecto al eje  $z$ , la cara frontal del anillo está orientada en la dirección del semieje positivo  $z$  y la cara posterior en la dirección del semieje negativo  $z$ .

Podemos especificar una parte de una corona circular con la siguiente función de GLU.

```
gluPartialDisk (ringName, rInner, rOuter, nRadii, nRings, startAngle, sweepAngle);
```

El parámetro de doble precisión `startAngle` hace referencia al ángulo en grados en el plano  $xy$  medido en el sentido de las agujas del reloj desde el semieje positivo  $y$ . De forma similar, el parámetro `sweepAngle` hace referencia a la distancia angular en grados desde el ángulo `startAngle`. Por tanto, una parte de una corona circular plana se visualiza desde el ángulo `startAngle` hasta `startAngle + sweepAngle`. Por

ejemplo, si `startAngle = 0.0` y `sweepAngle = 90.0`, entonces se muestra la parte de la corona circular situada en el primer cuadrante del plano *xy*.

La memoria reservada para cualquier superficie cuádrica creada con GLU se puede liberar además de eliminar la superficie con,

```
gluDeleteQuadric (quadricName);
```

También, podemos definir las direcciones de la cara frontal y la cara posterior de cualquier superficie cuádrica con la función de orientación:

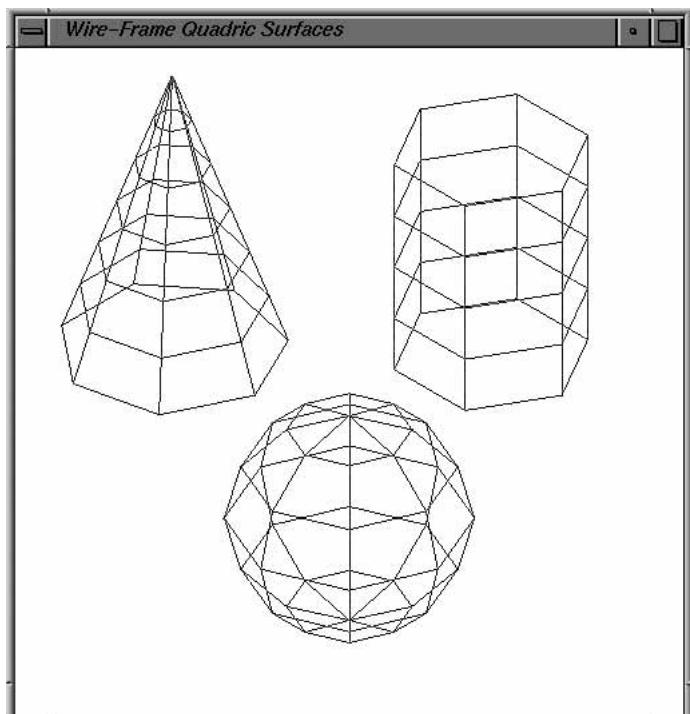
```
gluQuadricOrientation (quadricName, normalVectorDirection);
```

Al parámetro `normalVectorDirection` se le asigna `GLU_OUTSIDE` o `GLU_INSIDE` para indicar una dirección de los vectores normales a la superficie, donde «*outside*» indica la **dirección de la cara frontal** e «*inside*» indica la **dirección de la cara posterior**. El valor predeterminado es `GLU_OUTSIDE`. La dirección predeterminada de la cara frontal de una corona circular plana es la del semieje positivo *z* («por encima» del disco).

Otra opción es la generación de los vectores normales a la superficie.

```
gluQuadricNormals (quadricName, generationMode) ;
```

Al parámetro `generationMode` se le asigna una constante simbólica para indicar cómo se deberían generar los vectores normales a la superficie. La constante predeterminada es `GLU_NONE`, que significa que no hay que generar normales a la superficie y, habitualmente, no se aplican condiciones de iluminación a la superficie de la cuádrica. En el caso de sombreado plano de la superficie (un color constante en cada superficie), utilizamos la constante simbólica `GLU_FLAT`. Esta produce una normal a la superficie para cada cara poligonal. Cuando hay que aplicar otras condiciones de iluminación y sombreado, utilizamos la constante `GLU_SMOOTH`, que genera un vector normal para cada vértice de la superficie.



**FIGURA 8.8.** Visualización de una esfera creada con GLUT, un cono creado con GLUT y un cilindro creado con GLU, posicionados dentro de una ventana de visualización con el procedimiento `wireQuadSurfs`.

Entre otras opciones para las superficies cuádricas con GLU se incluye la modificación de parámetros de la textura de la superficie. Podemos designar una función que se ha de invocar cuando ocurre un error durante la generación de una superficie cuádrica:

```
gluQuadricCallback (quadricName, GLU_ERROR, function);
```

### Ejemplo de programa que utiliza las funciones de creación de superficies cuádricas de GLUT y GLU

Se visualizan tres objetos con superficies cuádricas (una esfera, un cono y un cilindro) en su modelo alámbrico con el siguiente programa de ejemplo. Establecemos la dirección de la vista como el semieje positivo  $z$  para que el eje de todos los objetos visualizados sea vertical. Los tres objetos se posicionan en diferentes localizaciones dentro de una única ventana de visualización, como se muestra en la Figura 8.8.

---

```
#include <GL/glut.h>

GLsizei winWidth = 500, winHeight = 500; // Tamaño inicial de la ventana de
// visualización.

void init (void)
{
    glClearColor (1.0, 1.0, 1.0, 0.0);           // Establece el color de la ventana
// de visualización.

}

void wireQuadSurfs (void)
{
    glClear (GL_COLOR_BUFFER_BIT);               // Borra la ventana de visualizacion.

    glColor3f (0.0, 0.0, 1.0);                  // Establece el color de las líneas
// en azul.

/* Establece los parametros de visualización con el eje z universal como
 * dirección vista arriba. */
    gluLookAt (2.0, 2.0, 2.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0);

/* Posiciona y muestra una esfera alámbrica de GLUT. */
    glPushMatrix ();
    glTranslatef (1.0, 1.0, 0.0);
    glutWireSphere (0.75, 8, 6);
    glPopMatrix ();

/* Posiciona y muestra un cono alámbrico de GLUT. */
    glPushMatrix ();
    glTranslatef (1.0, -0.5, 0.5);
    glutWireCone (0.7, 2.0, 7, 6);
    glPopMatrix ();
```

```

/* Posiciona y muestra un cilindro alámbrico de GLUT. */
GLUquadricObj *cylinder; // Establece el nombre del objeto de cuádrica de
                         GLU.

glPushMatrix ();
glTranslatef (0.0, 1.2, 0.8);
cylinder = gluNewQuadric ();
gluQuadricDrawStyle (cylinder, GLU_LINE);
gluCylinder (cylinder, 0.6, 0.6, 1.5, 6, 4);
glPopMatrix ();

glFlush ();
}

void winReshapeFcn (GLint newWidth, GLint newHeight)
{
    glViewport (0, 0, newWidth, newHeight);

    glMatrixMode (GL_PROJECTION);
    glOrtho (-2.0, 2.0, -2.0, 2.0, 0.0, 5.0);

    glMatrixMode (GL_MODELVIEW);

    glClear (GL_COLOR_BUFFER_BIT);
}

void main (int argc, char** argv)
{
    glutInit (&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    glutInitWindowPosition (100, 100);
    glutInitWindowSize (winWidth, winHeight);
    glutCreateWindow ("Superficies Cuádricas con modelo alámbrico");

    init ();
    glutDisplayFunc (wireQuadSurfs);
    glutReshapeFunc (winReshapeFcn);

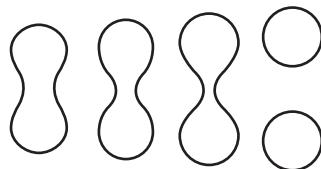
    glutMainLoop ();
}

```

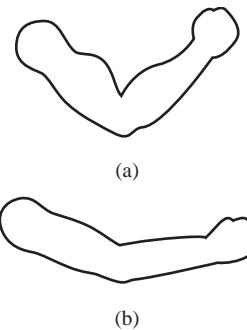
## 8.7 OBJETOS SIN FORMA (BLOBBY)

---

Se han desarrollado varias técnicas para modelar objetos no rígidos en aplicaciones de gráficos por computadora. En la Sección 8.26 se estudian métodos para visualizar las características de materiales tales como la ropa y la goma. Pero otros objetos, como las estructuras moleculares, los líquidos y las gotitas de agua, objetos que se funden y las formas de los músculos de los animales y de los humanos muestran un cierto grado de fluidez. Estos objetos cambian las características de su superficie con algunos movimientos o cuando se aproximan a otros objetos, y poseen superficies curvadas que no se pueden representar fácilmente con formas estándar. Generalmente, esta clase de objetos se denominan **objetos sin forma (blobby)**.



**FIGURA 8.9.** Enlace molecular. A medida que dos moléculas se separan una de la otra, las formas de la superficie se estiran, se rompen en dos y finalmente se contraen para formar esferas.



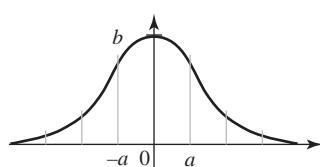
**FIGURA 8.10.** Formas de músculos «sin forma» de un brazo humano.

Una forma molecular, por ejemplo, se puede describir como esférica cuando está aislada, pero esta forma cambia cuando la molécula se aproxima a otra molécula. Esto es debido al hecho de que la forma de la nube de densidad de electrones se distorsiona con la presencia de otra molécula, para que tenga lugar un efecto de unión entre las dos moléculas. La Figura 8.9 muestra los efectos de estiramiento, de ruptura en dos y de contracción de las formas moleculares cuando dos moléculas se separan. Estas características no se pueden describir adecuadamente con simples formas esféricas o elípticas. De forma similar, la Figura 8.10 muestra formas musculares de un brazo humano, que presenta características similares.

Se han desarrollado varios modelos para representar objetos sin forma como funciones de distribución sobre regiones del espacio. Habitualmente, las formas de la superficie se describen de forma que el volumen del objeto permanece constante durante cualquier movimiento o interacción. Un método de modelado de objetos sin forma consiste en utilizar una combinación de funciones de densidad gaussianas, o bultos gaussianos (Figura 8.11). De esta manera, una función de la superficie se define como:

$$f(x, y, z) = \sum_k b_k e^{-a_k r_k^2} - T = 0 \quad (8.13)$$

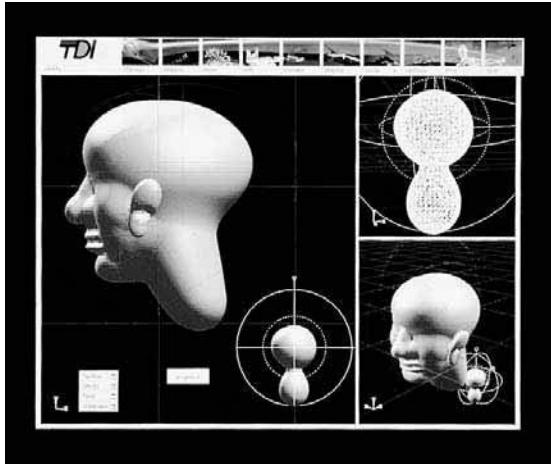
donde  $r_k^2 = x_k^2 + y_k^2 + z_k^2$ , el parámetro  $T$  es algún umbral especificado, y los parámetros  $a_k$  y  $b_k$  se utilizan para ajustar la cantidad de «sin forma» de las componentes individuales de la superficie. Los valores negativos de  $b_k$  se pueden utilizar para producir hendiduras en lugar de abultamientos. La Figura 8.12 muestra la estructura de la superficie de un objeto compuesto modelado con cuatro funciones de densidad gaussianas. En el nivel del umbral, se utilizan técnicas numéricas de búsqueda de raíces para localizar los valores de las coordenadas de la intersección. Las secciones rectas de los objetos individuales se modelan como círculos o elipses. Si las dos secciones rectas están próximas entre sí, se fusionan para dar lugar a una forma sin forma como se muestra en la Figura 8.9, cuya estructura depende de la separación de los dos objetos.



**FIGURA 8.11.** Una función de densidad gaussiana tridimensional centrada en el valor 0, con altura  $b$  y desviación estándar  $a$ .



**FIGURA 8.12.** Un objeto compuesto sin forma formado por cuatro abultamientos gaussianos.



**FIGURA 8.13.** Una distribución de pantalla, utilizada en los paquetes Blob Modeler y Blob Animador, para modelar objetos con metabolas. (Cortesía de Thomson Digital Image.)

Otros métodos para generar objetos sin forma utilizan funciones de densidad que decrecen hasta 0 en un intervalo finito, en lugar de exponencialmente. El modelo de metabolas describe los objetos compuestos como combinaciones de funciones de densidad cuádricas de la forma:

$$f(r) = \begin{cases} b\left(1 - \frac{3r^2}{d^2}\right), & \text{si } 0 < r \leq \frac{d}{3} \\ \frac{3}{2}b\left(1 - \frac{r}{d}\right)^2, & \text{si } \frac{d}{3} < r \leq d \\ 0, & \text{si } r > d \end{cases} \quad (8.14)$$

Y el modelo de **objeto suave** utiliza la función:

$$f(r) = \begin{cases} 1 - \frac{22r^2}{9d^2} + \frac{17r^4}{9d^4} - \frac{4r^6}{9d^6}, & \text{si } 0 < r \leq d \\ 0, & \text{si } r > d \end{cases} \quad (8.15)$$

Algunos paquetes de diseño y dibujo ahora proporcionan modelado de funciones sin forma para manejar aplicaciones que no se pueden modelar adecuadamente con otras representaciones. La Figura 8.13 muestra una interfaz de usuario de un modelador de objetos sin forma que utiliza metabolas.

## 8.8 REPRESENTACIONES CON SPLINES

En el dibujo de bocetos, un *spline* es una banda flexible que se utiliza para producir una curva suave que pasa por unos puntos concretos. Se distribuyen varios pesos pequeños a lo largo de la banda para mantenerla en su posición sobre la mesa de dibujo mientras se traza la curva. El término *curva con spline* en principio hacía referencia a una curva dibujada de este modo. Podemos describir matemáticamente tal curva con una función creada por tramos de polinomios cúbicos, cuya primera y segunda derivadas son continuas en las diferentes partes de la curva. En los gráficos por computadora, el término **curva con spline** ahora se refiere a cualquier curva compuesta formada por partes polinómicas que satisfacen condiciones de continuidad específicas en los límites de las mismas. Una **superficie con splines** se puede describir con dos conjuntos de curvas ortogonales con *splines*. Existen varias clases diferentes de especificaciones de *splines* que se utilizan en



**FIGURA 8.14.** Conjunto de seis puntos de control interpolados con secciones polinómicas continuas por tramos.



**FIGURA 8.15.** Conjunto de seis puntos de control aproximados con secciones polinómicas continuas por tramos.

aplicaciones de gráficos por computadora. Cada especificación individual simplemente hace referencia a un tipo particular de polinomio con ciertas condiciones específicas en los límites.

Los *splines* se utilizan para diseñar formas de curvas y de superficies, para digitalizar dibujos y para especificar trayectorias de animación de objetos o la posición de la cámara en una escena. Entre las aplicaciones habituales de CAD con *splines* se incluyen el diseño de la carrocería de un automóvil, las superficies de aviones o naves espaciales, los cascos de las embarcaciones y los electrodomésticos.

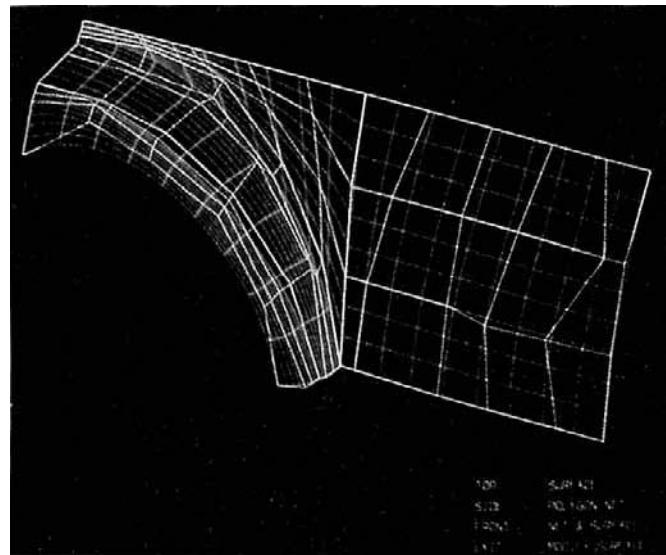
### Splines de interpolación y de aproximación

Especificamos una curva con *spline* proporcionando un conjunto de coordenadas de puntos, llamados **puntos de control**, que marcan la forma general de la curva. Estas coordenadas se ajustan mediante funciones polinómicas paramétricas y continuas por tramos de uno de estos dos modos. Cuando las partes polinómicas se ajustan para que todos los puntos de control estén conectados, como en la Figura 8.14, se dice que la curva resultante **interpola** el conjunto de puntos de control. En cambio, cuando la curva de polinomios generada se dibuja para que algunos, o todos, los puntos de control no estén en la trayectoria de la curva, se dice que la curva **aproxima** el conjunto de puntos de control (Figura 8.15). Se utilizan métodos similares para construir superficies de interpolación o de aproximación con *splines*.

Los métodos de interpolación se utilizan habitualmente para digitalizar dibujos o para especificar trayectorias de animación. Los métodos de aproximación se utilizan fundamentalmente como herramientas de diseño para crear formas de objetos. La Figura 8.16 muestra la visualización por pantalla de una superficie de aproximación con *splines*, de una aplicación de diseño. Líneas rectas conectan los puntos de control situados por encima de la superficie.

Una curva o una superficie con *splines* se define, modifica y manipula con operaciones sobre los puntos de control. Seleccionando interactivamente posiciones espaciales para los puntos de control, un diseñador puede establecer una forma inicial. Después de que se visualiza el polinomio de ajuste para un conjunto concreto de puntos de control, el diseñador puede volver a posicionar algunos o todos los puntos de control para reestructurar la forma del objeto. Las transformaciones geométricas (traslación, rotación y cambio de escala) se aplican al objeto transformando los puntos de control. Además, los paquetes de CAD insertan puntos de control adicionales para ayudar al diseñador en el ajuste de las formas de los objetos.

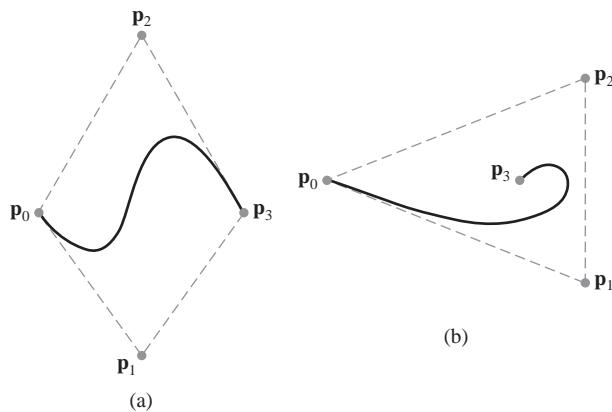
Un conjunto de puntos de control define un límite de una región del espacio que se denomina **armazón (hull) convexo**. Una manera de imaginarse la forma de un armazón convexo para una curva bidimensional consiste en imaginar una banda de goma estirada alrededor de los puntos de control para que cada punto de control esté sobre el perímetro de ese límite o dentro de éste (Figura 8.17). Por tanto, el armazón convexo de una curva bidimensional con *spline* es un polígono convexo. En el espacio tridimensional, el armazón convexo de un conjunto de puntos de control de *splines* forma un poliedro convexo. El armazón convexo



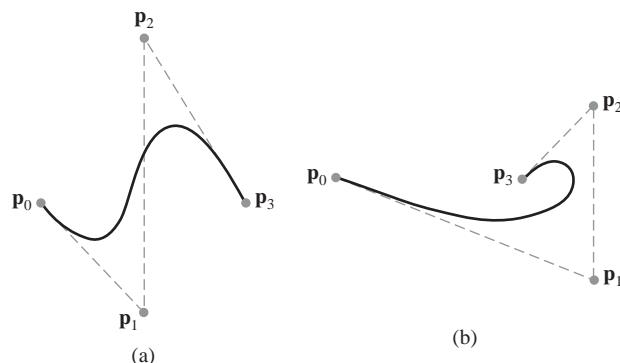
**FIGURA 8.16.** Una superficie de aproximación con *splines* de una aplicación de CAD de diseño de automóviles. Los contornos de la superficie se dibujan con partes de curvas polinómicas y los puntos de control de la superficie están conectados mediante segmentos de línea recta. (Cortesía de Evans & Sutherland.)

proporciona una medida de la desviación de una curva o una superficie de la región del espacio próxima a los puntos de control. En la mayoría de los casos, un *spline* está confinado dentro de su armazón convexo, lo cual garantiza que la forma de objeto sigue los puntos de control sin oscilaciones erráticas. También, el armazón convexo proporciona una medida de las amplitudes de las coordenadas de una curva o una superficie diseñadas, por lo que es útil en subrutinas de recorte y de visualización.

Una polilínea que conecta la secuencia de puntos de control de una curva de aproximación con *splines* se muestra habitualmente, para recordar al diseñador las posiciones de los puntos de control y su ordenación. Este conjunto de segmentos de línea conectados se denomina **grafo de control** de la curva. A menudo, el grafo de control se denomina «polígono de control» o «polígono característico», aunque el grafo de control es una polilínea y no un polígono. La Figura 8.18 muestra la forma del grafo de control de las secuencias de puntos de control de la Figura 8.17. En una superficie con *splines*, dos conjuntos de polilíneas que conectan los puntos de control forman las aristas de las caras poligonales, de una malla de cuadriláteros del grafo de control de la superficie, como se muestra en la Figura 8.16.



**FIGURA 8.17.** Formas de los armazones convexas (líneas discontinuas) de dos conjuntos de puntos de control en el plano  $xy$ .



**FIGURA 8.18.** Formas de los grafos de control (líneas discontinuas) de dos conjuntos de puntos de control en el plano  $xy$ .

## Condiciones de continuidad paramétricas

Para garantizar una transición suave de una parte de un *spline* paramétrico por tramos a la siguiente, podemos imponer varias **condiciones de continuidad** en los puntos de conexión. Si cada parte de la curva con *spline* se describe con un conjunto de funciones de coordenadas paramétricas de la forma,

$$x = x(u), \quad y = y(u), \quad z = z(u), \quad u_1 \leq u \leq u_2 \quad (8.16)$$

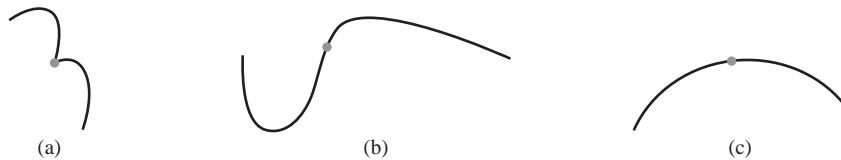
establecemos la **continuidad paramétrica** haciendo coincidir las derivadas paramétricas de las partes continuas de la curva en sus extremos comunes.

La **continuidad paramétrica de orden cero**, representada como continuidad  $C^0$ , significa que las curvas se encuentran. Es decir, los valores de  $x$ ,  $y$  y  $z$  evaluados en  $u_2$  en la primera parte de la curva son iguales, respectivamente, a los valores de  $x$ ,  $y$  y  $z$  evaluados en  $u_1$  en la siguiente parte de la curva. La **continuidad paramétrica de primer orden**, referenciada como continuidad  $C^1$ , significa que las primeras derivadas (líneas tangentes) de las funciones de las coordenadas de las Ecuaciones 8.16 de dos partes sucesivas de la curva son iguales en su punto de unión. La **continuidad paramétrica de segundo orden**, o continuidad  $C^2$ , significa que tanto la primera como la segunda derivada paramétrica de las dos partes de la curva son iguales en la intersección. Las condiciones de continuidad paramétrica de orden superior se definen de forma similar. La Figura 8.19 muestra ejemplos de continuidad  $C^0$ ,  $C^1$  y  $C^2$ .

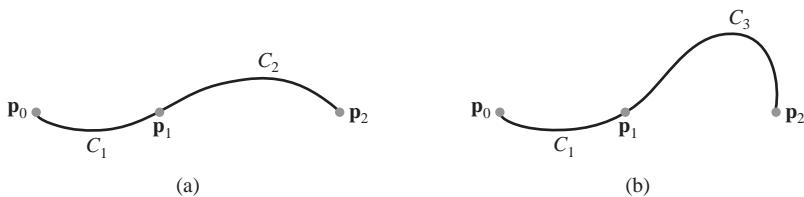
Con continuidad paramétrica de segundo orden, las tasas de cambio de los vectores tangente de las partes con conexión son iguales en su intersección. Por tanto, la transición de las líneas tangentes es suave de una parte de la curva a la siguiente (Figura 8.19(c)). Pero con continuidad paramétrica de primer orden, la tasa de cambio de los vectores tangentes de las dos partes pueden ser bastante diferentes (Figura 8.19(b)), de modo que las formas generales de las dos partes adyacentes pueden cambiar abruptamente. La continuidad paramétrica de primer orden es a menudo suficiente para la digitalización de dibujos y para algunas aplicaciones de diseño, mientras que la continuidad de segundo orden es útil para establecer las trayectorias de animación de movimiento de una cámara y para muchos requisitos de CAD de precisión. Una cámara desplazándose según la trayectoria curva de la Figura 8.19(b) con incrementos iguales en el parámetro  $u$  experimentaría un cambio abrupto en la aceleración en la frontera de las dos partes, produciendo una discontinuidad en la secuencia de movimientos. Pero si la cámara estuviese desplazándose según la trayectoria de la Figura 8.19(c), la secuencia de cuadros del movimiento sufriría una transición suave en la frontera.

## Condiciones de continuidad geométrica

Otro método para unir dos partes sucesivas de una curva consiste en especificar condiciones de **continuidad geométrica**. En este caso, requerimos sólo que las derivadas paramétricas de las dos partes sean proporcionales entre sí en su frontera común, en lugar de requerir igualdad.



**FIGURA 8.19.** Construcción por partes de una curva uniendo dos segmentos de curva utilizando órdenes diferentes de continuidad: (a) sólo continuidad de orden cero, (b) continuidad de primer orden y (c) continuidad de segundo orden.



**FIGURA 8.20.** Tres puntos de control ajustados mediante dos partes de curva unidas con (a) continuidad paramétrica y con (b) continuidad geométrica, en la que el vector tangente de la curva C<sub>3</sub> en el punto P<sub>1</sub> tiene una mayor magnitud que el vector tangente de la curva C<sub>1</sub> en P<sub>1</sub>.

La **continuidad geométrica de orden cero**, descrita como continuidad G<sup>0</sup>, es la misma que la continuidad paramétrica de orden cero. Es decir, dos partes sucesivas de la curva deben tener las mismas coordenadas en el punto frontera. La **continuidad geométrica de primer orden**, o continuidad G<sup>1</sup>, significa que las primeras derivadas paramétricas son proporcionales en la intersección de dos partes sucesivas. Si hacemos referencia a la posición paramétrica en la curva como  $\mathbf{P}(u)$ , la dirección del vector tangente  $\mathbf{P}'(u)$ , pero no necesariamente su magnitud, será la misma en dos partes sucesivas de la curva en su punto común con continuidad G<sup>1</sup>. La **continuidad geométrica de segundo orden**, o continuidad G<sup>2</sup>, significa que tanto la primera como las segundas derivadas paramétricas de las dos partes de la curva son proporcionales en su frontera. Con continuidad G<sup>2</sup>, las curvaturas de dos partes de la curva coincidirán en el punto de unión.

Una curva generada con condiciones de continuidad geométrica es similar a una generada con continuidad paramétrica, pero con ligeras diferencias en la forma de la curva. La Figura 8.20 proporciona una comparación de las continuidades geométrica y paramétrica. Con continuidad geométrica, la curva se desplaza hacia la parte con mayor magnitud en su vector tangente.

## Especificaciones de *splines*

Existen tres métodos equivalentes para especificar una representación de *spline* concreta, dado el grado del polinomio y los puntos de control: (1) Podemos establecer el conjunto de condiciones en la frontera que se imponen en el *spline*; o (2) podemos establecer la matriz que caracteriza el *spline*; o (3) podemos establecer el conjunto de *funciones de combinación* (o *funciones base*) que determinan cómo las restricciones especificadas en la curva se combinan para calcular los puntos de la trayectoria de la curva.

Para mostrar estas tres especificaciones equivalentes, suponga que tenemos la siguiente representación polinómica cúbica paramétrica para la coordenada  $x$  a lo largo de la trayectoria de una parte de una curva con *spline*:

$$x(u) = a_x u^3 + b_x u^2 + c_x u + d_x, \quad 0 \leq u \leq 1 \quad (8.17)$$

Las condiciones de la frontera de esta curva se pueden establecer en los puntos extremos  $x(0)$  y  $x(1)$  y en las primeras derivadas paramétricas en los puntos extremos:  $x'(0)$  y  $x'(1)$ . Estas cuatro condiciones de la frontera son suficientes para determinar los valores de los cuatro coeficientes  $a_x$ ,  $b_x$ ,  $c_x$  y  $d_x$ .

A partir de las condiciones de la frontera, podemos obtener la matriz que caracteriza esta curva con *splines* reescribiendo en primer lugar la Ecuación 8.17 como un producto matricial:

$$x(u) = [u^3 \quad u^2 \quad u \quad 1] \begin{bmatrix} a_x \\ b_x \\ c_x \\ d_x \end{bmatrix} = \mathbf{U} \cdot \mathbf{C} \quad (8.18)$$

en el que  $\mathbf{U}$  es la matriz fila con las potencias del parámetro  $u$  y  $\mathbf{C}$  es la matriz columna de coeficientes. Utilizando la Ecuación 8.18, podemos escribir las condiciones en la frontera en forma matricial y resolver para la matriz de coeficientes  $\mathbf{C}$  de este modo,

$$\mathbf{C} = \mathbf{M}_{\text{spline}} \cdot \mathbf{M}_{\text{geom}} \quad (8.19)$$

donde  $\mathbf{M}_{\text{geom}}$  es una matriz columna de cuatro elementos que contiene los valores de las restricciones geométricas (condiciones en la frontera) del *spline*, y  $\mathbf{M}_{\text{spline}}$  es la matriz de tamaño 4 por 4 que transforma los valores de las restricciones geométricas en los coeficientes del polinomio y proporciona una caracterización de la curva con *splines*. La matriz  $\mathbf{M}_{\text{geom}}$  contiene las coordenadas de los puntos de control y otras restricciones geométricas que se hayan especificado. Por tanto, podemos sustituir la matriz  $\mathbf{C}$  de la Ecuación 8.18 para obtener:

$$x(u) = \mathbf{U} \cdot \mathbf{M}_{\text{spline}} \cdot \mathbf{M}_{\text{geom}} \quad (8.20)$$

La matriz  $\mathbf{M}_{\text{spline}}$ , que caracteriza una representación de un *spline*, a veces llamada *matriz base*, es particularmente útil para transformar una representación de un *spline* a otra.

Finalmente, podemos desarrollar la Ecuación 8.20 para obtener una representación polinomial de la coordenada  $x$  en términos de los parámetros de las restricciones geométricas  $g_k$ , como las coordenadas de los puntos de control y la pendiente de la curva en los puntos de control:

$$x(u) = \sum_{k=0}^3 g_k \cdot \text{BF}_k(u) \quad (8.21)$$

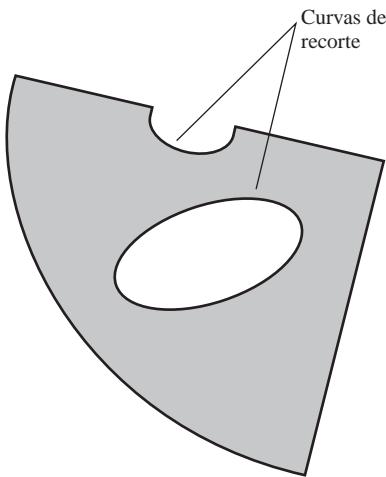
Los polinomios  $\text{BF}_k(u)$ , con  $k = 0, 1, 2, 3$ , se denominan **funciones de combinación** o **funciones base** porque combinan (funden) los valores de las restricciones geométricas para obtener las coordenadas a lo largo de la curva. En las secciones siguientes, exploraremos las características de varias curvas y superficies con *splines* que son útiles en aplicaciones de gráficos por computadora, incluyendo la especificación de sus representaciones con matriz y con función de fundido.

## Superficies con *splines*

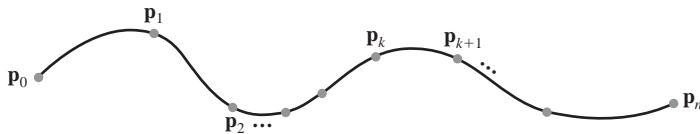
El procedimiento habitual para definir una superficie con *splines* consiste en especificar dos conjuntos de curvas ortogonales con *splines*, utilizando una malla de puntos de control sobre una región del espacio. Si hacemos referencia a los puntos de control como  $\mathbf{p}_{ku,kv}$ , cualquier punto de la superficie con *splines* se puede calcular como el producto cartesiano de las funciones de combinación de la curva con *splines*:

$$\mathbf{P}(u, v) = \sum_{k_u, k_v} \mathbf{p}_{k_u, k_v} \text{BF}_{k_u}(u) \text{BF}_{k_v}(v) \quad (8.22)$$

Los parámetros de superficie  $u$  y  $v$  a menudo varían dentro del rango de 0 a 1, pero este rango depende del tipo de curvas con *splines* que utilicemos. Un método para designar los puntos de control tridimensionales consiste en seleccionar valores de la altura por encima de una malla bidimensional de puntos sobre un plano de tierra.



**FIGURA 8.21.** Modificación de una parte de una superficie utilizando curvas de recorte.



**FIGURA 8.22.** Interpolación con *splines* cúbicos continuos por tramos de  $n + 1$  puntos de control.

### Recorte de superficies con *splines*

En aplicaciones CAD, un diseño de una superficie puede requerir algunas características que no se implementan fácilmente simplemente ajustando los puntos de control. Por ejemplo, una parte de una superficie con *splines* puede necesitar un recorte para encajar dos piezas del diseño, o se puede necesitar un agujero para que un conducto pueda pasar a través de la superficie. Para estas aplicaciones, los paquetes gráficos a menudo proporcionan funciones para generar **curvas de recorte** que se pueden utilizar para extraer partes de una superficie con *splines*, como se muestra en la Figura 8.21. Las curvas de recorte se definen habitualmente con coordenadas paramétricas  $uv$  de la superficie y, a menudo, se deben especificar como curvas cerradas.

## 8.9 MÉTODOS DE INTERPOLACIÓN CON *SPLINES* CÚBICOS

Esta clase de *splines* se usa muy a menudo para establecer trayectorias para el movimiento de objetos o para proporcionar una representación de un objeto o dibujo existente, pero los *splines* de interpolación también se utilizan a veces para diseñar las formas de objetos. Los polinomios cúbicos ofrecen un compromiso razonable entre flexibilidad y velocidad de computación. Comparados con polinomios de mayor grado, los *splines* cúbicos requieren menos cálculos y espacio de almacenamiento, y son más estables. Comparados con los polinomios cuadráticos y los segmentos de línea recta, los polinomios cúbicos son más flexibles para modelado de formas de objetos.

Dado un conjunto de puntos de control, los *splines* de interpolación cúbicos se obtienen ajustando los puntos de entrada con una curva polinómica por tramos cúbica que pasa a través de cada punto de control. Suponga que tenemos  $n + 1$  puntos de control especificados con las coordenadas:

$$\mathbf{p}_k = (x_k, y_k, z_k), \quad k = 0, 1, 2, \dots, n$$

En la Figura 8.22 se muestra un ajuste de interpolación cúbica de estos puntos. Podemos describir el polinomio cúbico paramétrico que hay que ajustar entre cada par de puntos de control con el siguiente conjunto de ecuaciones:

$$\begin{aligned}x(u) &= a_x u^3 + b_x u^2 + c_x u + d_x \\y(u) &= a_y u^3 + b_y u^2 + c_y u + d_y, \quad (0 \leq u \leq 1) \\z(u) &= a_z u^3 + b_z u^2 + c_z u + d_z\end{aligned}\tag{8.23}$$

Para cada una de estas tres ecuaciones, necesitamos determinar los valores de los cuatro coeficientes  $a$ ,  $b$ ,  $c$  y  $d$  de la representación polinómica de cada una de las  $n$  partes de la curva, entre los  $n + 1$  puntos de control. Hacemos esto estableciendo suficientes condiciones en los límites en los puntos de control entre partes de la curva, para que podamos obtener los valores numéricos de todos los coeficientes. En las siguientes secciones, estudiaremos métodos habituales para establecer las condiciones en los límites de los *splines* cúbicos de interpolación.

### Splines cúbicos naturales

Una de las primeras curvas con *splines* que se desarrolló para aplicaciones gráficas es el **spline cúbico natural**. Esta curva de interpolación es una representación matemática del *spline* original de generación de borradores. Formulamos un *spline* cúbico natural imponiendo que dos partes adyacentes de la curva posean la misma primera y segunda derivadas paramétricas en su límite común. Por tanto, los *splines* cúbicos naturales tienen continuidad  $C^2$ .

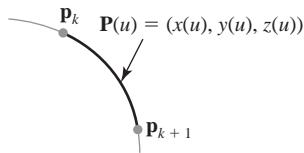
Si tenemos  $n + 1$  puntos de control, como en la Figura 8.22, entonces tenemos  $n$  partes de la curva con un total de  $4n$  coeficientes de polinomio que hay que determinar. En cada uno de los  $n - 1$  puntos de control interiores tenemos cuatro condiciones en el límite: las dos partes de la curva a cada lado de un punto de control deben tener las mismas primera y segunda derivadas paramétricas en dicho punto de control, y cada curva debe pasar por ese punto de control. Esto nos proporciona  $4n - 4$  ecuaciones que hay que satisfacer con  $4n$  coeficientes de polinomio. Obtenemos una ecuación adicional a partir del primer punto de control  $\mathbf{p}_0$ , la posición de comienzo de la curva y otra condición a partir del punto de control  $\mathbf{p}_n$ , que debe ser el último punto de la curva. Sin embargo, todavía necesitamos dos condiciones más para ser capaces de determinar los valores de todos los coeficientes. Un método para obtener las dos condiciones adicionales consiste en establecer las segundas derivadas en  $\mathbf{p}_0$  y  $\mathbf{p}_n$  en el valor 0. Otra técnica es añadir dos puntos de control extra (llamados *puntos ficticios*), uno en cada extremo de la secuencia original de puntos de control. Es decir, añadimos un punto de control etiquetado como  $\mathbf{p}_{-1}$  en el comienzo de la curva y un punto de control etiquetado como  $\mathbf{p}_{n+1}$  en el final. Entonces todos los puntos de control originales son puntos interiores y tenemos las  $4n$  condiciones necesarias en el límite.

Aunque los *splines* cúbicos naturales son un modelo matemático del *spline* de generación de borradores, tienen una desventaja principal. Si la posición de cualquiera de los puntos de control se modifica, la curva entera se ve afectada. Por tanto, los *splines* naturales cúbicos no permiten «control local», por lo que no podemos reestructurar parte de la curva sin especificar un conjunto totalmente nuevo de puntos de control. Por esta razón, se han desarrollado otras representaciones para un *spline* cúbico de interpolación.

### Interpolación de Hermite

Un ***spline* de Hermite** (denominado así en honor de un matemático francés llamado Charles Hermite) es un polinomio de interpolación cúbico por tramos con una tangente específica en cada punto de control. A diferencia de los *splines* cúbicos naturales, los *splines* de Hermite se pueden ajustar localmente porque cada parte de la curva sólo depende de las restricciones de sus puntos extremos.

Si  $\mathbf{P}(u)$  representa una función paramétrica cúbica de punto para la parte de la curva entre los puntos de control  $\mathbf{p}_k$  y  $\mathbf{p}_{k+1}$ , como se muestra en la Figura 8.23, entonces las condiciones en los límites que definen esta parte de curva de Hermite son:



**FIGURA 8.23.** Función paramétrica de punto  $\mathbf{P}(u)$  para una parte de curva de Hermite entre los puntos de control  $\mathbf{p}_k$  y  $\mathbf{p}_{k+1}$ .

$$\begin{aligned}\mathbf{P}(0) &= \mathbf{p}_k \\ \mathbf{P}(1) &= \mathbf{p}_{k+1} \\ \mathbf{P}'(0) &= \mathbf{D}\mathbf{p}_k \\ \mathbf{P}'(1) &= \mathbf{D}\mathbf{p}_{k+1}\end{aligned}\tag{8.24}$$

donde  $\mathbf{D}\mathbf{p}_k$  y  $\mathbf{D}\mathbf{p}_{k+1}$  especifican los valores de las derivadas paramétricas (pendientes de la curva) en los puntos de control  $\mathbf{p}_k$  y  $\mathbf{p}_{k+1}$ , respectivamente.

Podemos escribir el vector equivalente a las Ecuaciones 8.23 para esta parte de la curva de Hermite como:

$$\mathbf{P}(u) = \mathbf{a} u^3 + \mathbf{b} u^2 + \mathbf{c} u + \mathbf{d}, \quad 0 \leq u \leq 1\tag{8.25}$$

donde la componente  $x$  de  $\mathbf{P}(u)$  es  $x(u) = a_x u^3 + b_x u^2 + c_x u + d_x$ , y de forma similar para las componentes  $y$  y  $z$ . La matriz equivalente a la Ecuación 8.25 es:

$$\mathbf{P}(u) = [u^3 \quad u^2 \quad u \quad 1] \cdot \begin{bmatrix} \mathbf{a} \\ \mathbf{b} \\ \mathbf{c} \\ \mathbf{d} \end{bmatrix}\tag{8.26}$$

y la derivada de la función de punto se puede expresar como,

$$\mathbf{P}'(u) = [3u^2 \quad 2u \quad 1 \quad 0] \cdot \begin{bmatrix} \mathbf{a} \\ \mathbf{b} \\ \mathbf{c} \\ \mathbf{d} \end{bmatrix}\tag{8.27}$$

Sustituyendo los valores de los puntos extremos 0 y 1 en el parámetro  $u$  de las dos ecuaciones anteriores, podemos expresar las condiciones 8.24 de Hermite en los límites en forma matricial:

$$\begin{bmatrix} \mathbf{p}_k \\ \mathbf{p}_{k+1} \\ \mathbf{D}\mathbf{p}_k \\ \mathbf{D}\mathbf{p}_{k+1} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{a} \\ \mathbf{b} \\ \mathbf{c} \\ \mathbf{d} \end{bmatrix}\tag{8.28}$$

Resolviendo esta ecuación para los coeficientes de los polinomios, tenemos:

$$\begin{bmatrix} \mathbf{a} \\ \mathbf{b} \\ \mathbf{c} \\ \mathbf{d} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{bmatrix}^{-1} \cdot \begin{bmatrix} \mathbf{p}_k \\ \mathbf{p}_{k+1} \\ \mathbf{D}\mathbf{p}_k \\ \mathbf{D}\mathbf{p}_{k+1} \end{bmatrix} = \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{p}_k \\ \mathbf{p}_{k+1} \\ \mathbf{D}\mathbf{p}_k \\ \mathbf{D}\mathbf{p}_{k+1} \end{bmatrix} = \mathbf{M}_H \cdot \begin{bmatrix} \mathbf{p}_k \\ \mathbf{p}_{k+1} \\ \mathbf{D}\mathbf{p}_k \\ \mathbf{D}\mathbf{p}_{k+1} \end{bmatrix}\tag{8.29}$$

donde  $\mathbf{M}_H$ , la matriz de Hermite, es la inversa de la matriz de restricciones en los límites. La Ecuación 8.26 se puede escribir por tanto en función de las condiciones en el límite como:

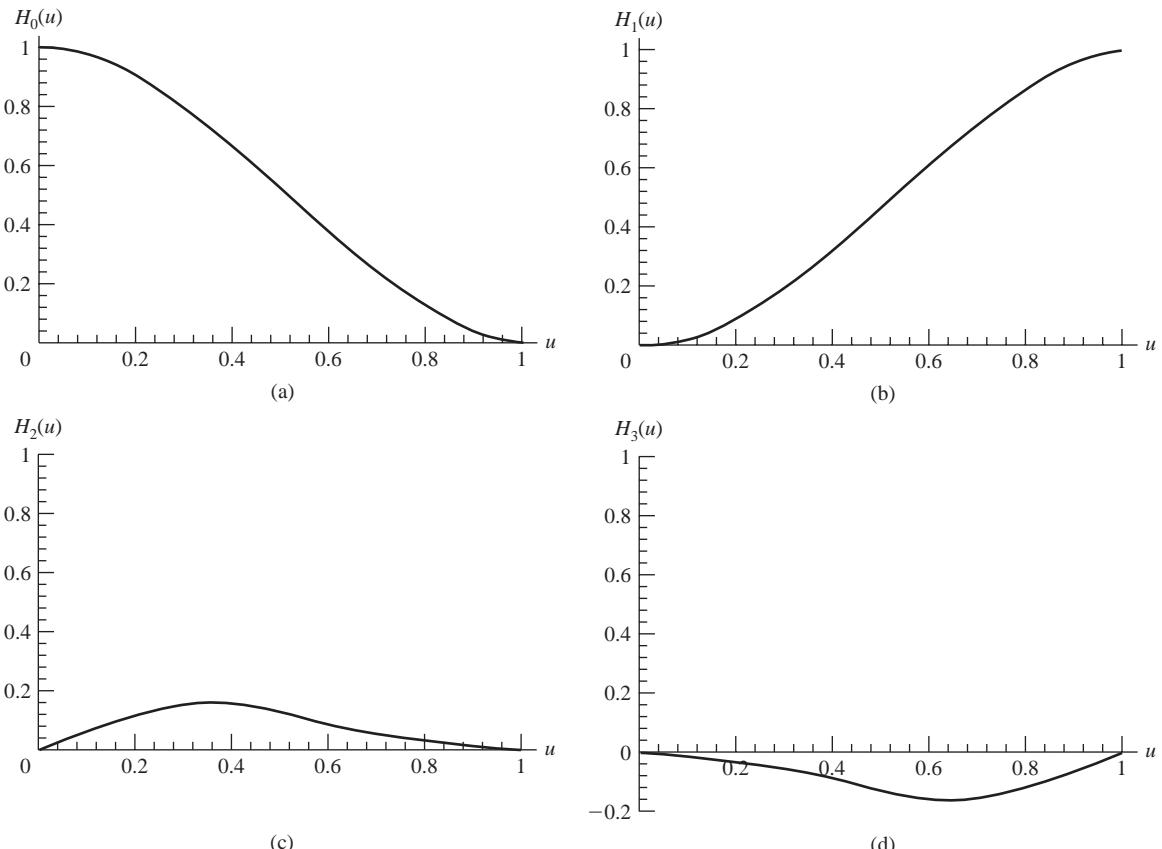
$$\mathbf{P}(u) = [u^3 \quad u^2 \quad u \quad 1] \cdot \mathbf{M}_H \cdot \begin{bmatrix} \mathbf{p}_k \\ \mathbf{p}_{k+1} \\ \mathbf{D}\mathbf{p}_k \\ \mathbf{D}\mathbf{p}_{k+1} \end{bmatrix} \quad (8.30)$$

Finalmente, podemos determinar las expresiones de las funciones de combinación de los polinomios de Hermite,  $H_k(u)$  con  $k = 0, 1, 2, 3$ , realizando las multiplicaciones de las matrices de la Ecuación 8.30 y agrupando los coeficientes de las restricciones en los límites para obtener la forma polinómica:

$$\begin{aligned} \mathbf{P}(u) &= \mathbf{p}_k(2u^3 - 3u^2 + 1) + \mathbf{p}_{k+1}(-2u^3 + 3u^2) + \mathbf{D}\mathbf{p}_k(u^3 - 2u^2 + u) \\ &\quad + \mathbf{D}\mathbf{p}_{k+1}(u^3 + u^2) \\ &= \mathbf{p}_k H_0(u) + \mathbf{p}_{k+1} H_1 + \mathbf{D}\mathbf{p}_k H_2 + \mathbf{D}\mathbf{p}_{k+1} H_3 \end{aligned} \quad (8.31)$$

La Figura 8.24 muestra la forma de las cuatro funciones de combinación de Hermite.

Los polinomios de Hermite pueden ser útiles en algunas aplicaciones de digitalización en las que puede que no sea demasiado difícil especificar o aproximar las pendientes de la curva. Pero en la mayoría de los problemas de gráficos por computadora es más útil generar curvas con *splines* que no requieran la introducción



**FIGURA 8.24.** Funciones de combinación de Hermite.

de los valores de las pendientes de la curva u otra información geométrica, además de las coordenadas de los puntos de control. Los *splines cardinales* y los *splines* de Kochanek-Bartels, estudiados en las dos secciones siguientes, son modificaciones de los *splines* de Hermite que no requieren la introducción de los valores de las derivadas de la curva en los puntos de control. Los procedimientos para estos *splines* calculan las derivadas paramétricas a partir de las coordenadas de los puntos de control.

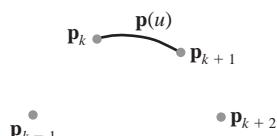
## Splines cardinales

Al igual que los *splines* de Hermite, los *splines cardinales* son polinomios de interpolación por tramos cúbicos con tangentes específicas en los puntos extremos en los límites de cada sección de la curva. La diferencia es que no introducimos los valores de las tangentes en los puntos extremos. En un *spline* cardinal, la pendiente en el punto de control se calcula a partir de las coordenadas de los dos puntos de control adyacentes.

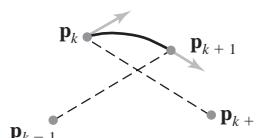
Una sección de un *spline* cardinal queda completamente determinada con cuatro puntos de control consecutivos. Los puntos de control centrales son los puntos extremos de la sección, y los otros dos puntos se utilizan en el cálculo de las pendientes en los puntos extremos. Si tomamos  $\mathbf{P}(u)$  como la representación de la función de punto paramétrica cúbica de la sección de curva entre los puntos de control  $\mathbf{p}_k$  y  $\mathbf{p}_{k+1}$ , como en la Figura 8.25, entonces los cuatro puntos de control desde  $\mathbf{p}_{k-1}$  hasta  $\mathbf{p}_{k+1}$  se utilizan para establecer las condiciones en los límites de la sección de *spline* cardinal de este modo:

$$\begin{aligned}\mathbf{P}(0) &= \mathbf{p}_k \\ \mathbf{P}(1) &= \mathbf{p}_{k+1} \\ \mathbf{P}'(0) &= \frac{1}{2}(1-t)(\mathbf{p}_{k+1} - \mathbf{p}_{k-1}) \\ \mathbf{P}'(1) &= \frac{1}{2}(1-t)(\mathbf{p}_{k+2} - \mathbf{p}_k)\end{aligned}\quad (8.32)$$

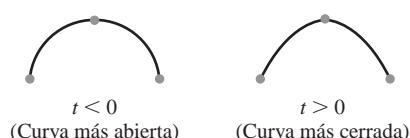
Por tanto, las pendientes en los puntos de control  $\mathbf{p}_k$  y  $\mathbf{p}_{k+1}$  se toman proporcionales, respectivamente, a las cuerdas  $\mathbf{p}_{k-1}\mathbf{p}_{k+1}$  y  $\mathbf{p}_k\mathbf{p}_{k+2}$  (Figura 8.26). El parámetro  $t$  se denomina parámetro de **tensión**, ya que controla cómo flojo o apretado se ajusta el *spline* cardinal a los puntos de control de entrada. La Figura 8.27 muestra la forma de una curva cardinal con un valor muy pequeño y con un valor muy grande de la tensión  $t$ . Cuando  $t = 0$ , esta clase de curvas se denominan *splines de Catmull-Rom*, o *splines de Overhauser*.



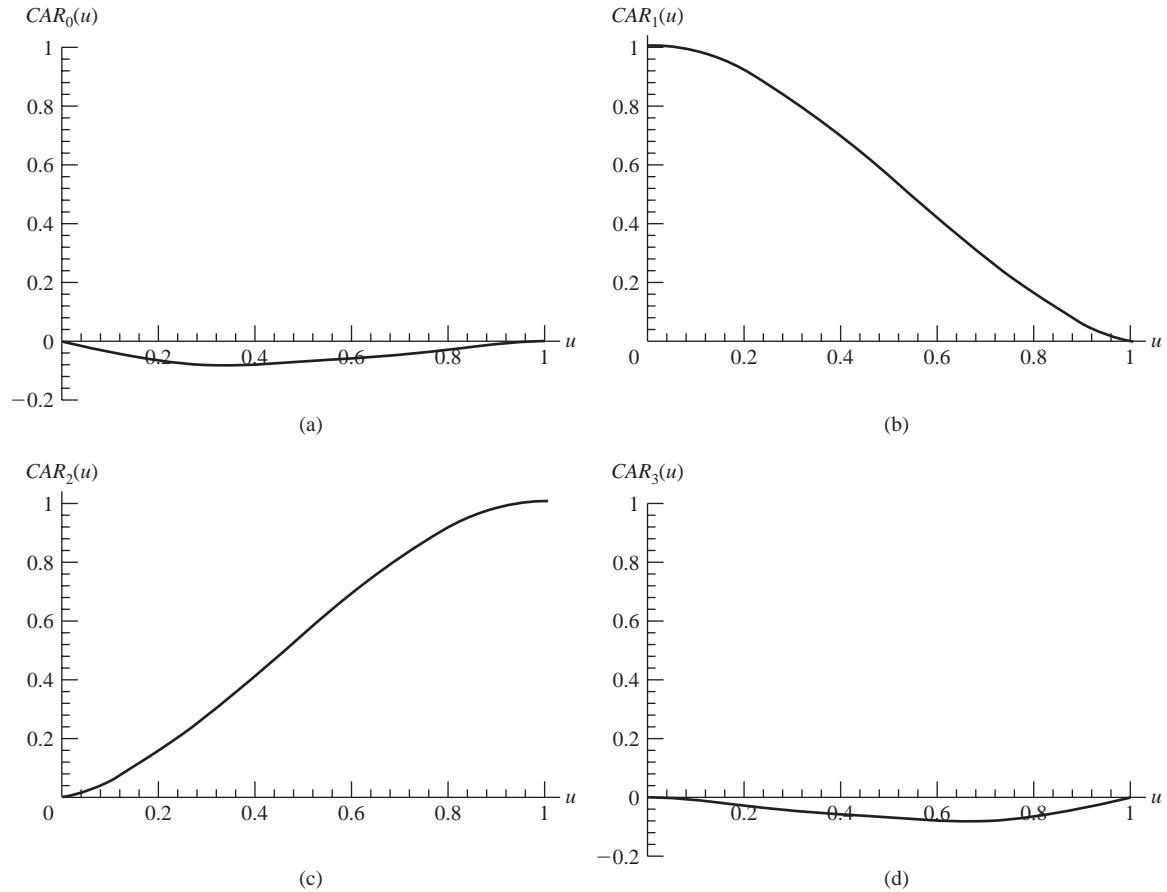
**FIGURA 8.25.** Función paramétrica de punto  $\mathbf{P}(u)$  de una sección de *spline* cardinal entre los puntos de control  $\mathbf{p}_k$  y  $\mathbf{p}_{k+1}$ .



**FIGURA 8.26.** Los vectores tangentes en los puntos extremos de una sección de *spline* cardinal son paralelos a las cuerdas formadas con los puntos de control vecinos (líneas discontinuas).



**FIGURA 8.27.** Efecto del parámetro de tensión sobre la forma de una sección de *spline* cardinal.



**FIGURA 8.28.** Funciones de combinación de *splines* cardinales con  $t = 0$  ( $s = 0.5$ ).

Utilizando métodos similares a los utilizados para los polinomios de Hermite, podemos convertir las condiciones 8.32 de los límites a su forma matricial,

$$\mathbf{P}(u) = [u^3 \quad u^2 \quad u \quad 1] \cdot \mathbf{M}_C \cdot \begin{bmatrix} \mathbf{p}_{k-1} \\ \mathbf{p}_k \\ \mathbf{p}_{k+1} \\ \mathbf{p}_{k+2} \end{bmatrix} \quad (8.33)$$

donde la matriz cardinal es:

$$\mathbf{M}_C = \begin{bmatrix} -s & 2-s & s-2 & s \\ 2s & s-3 & 3-2s & -s \\ -s & 0 & s & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \quad (8.34)$$

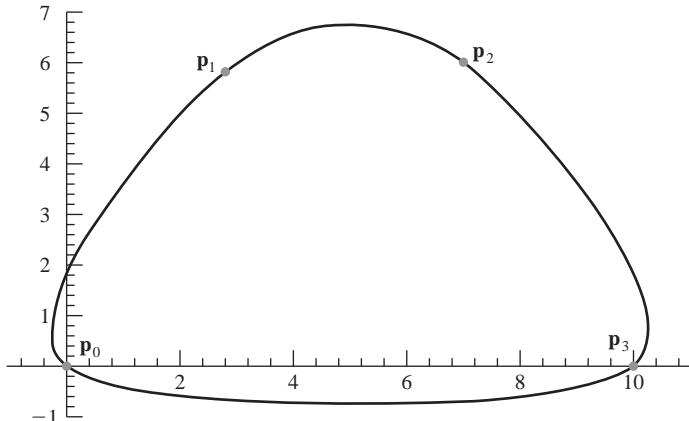
con  $s = (1-t)/2$ .

Desarrollando la Ecuación matricial 8.33 en forma polinómica, tenemos:

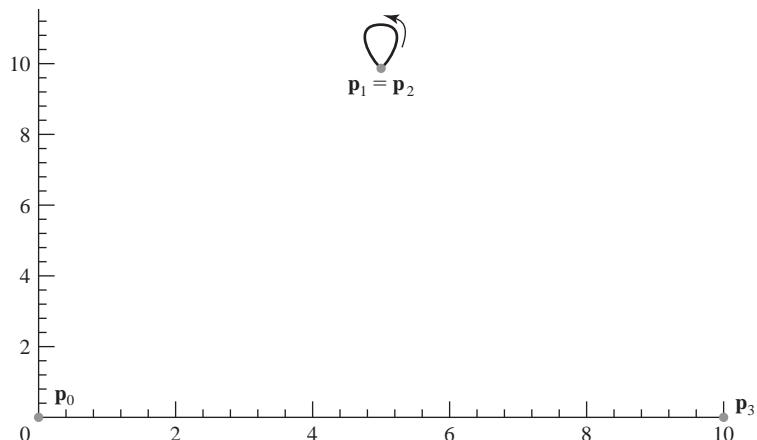
$$\begin{aligned}
 \mathbf{P}(u) &= \mathbf{p}_{k-1}(-s u^3 + 2s u^2 - s u) + \mathbf{p}_k[(2-s)u^3 + (s-3)u^2 + 1] \\
 &\quad + \mathbf{p}_{k+1}[(s-2)u^3 + (3-2s)u^2 + s u] + \mathbf{p}_{k+2}(s u^3 - s u^2) \\
 &= \mathbf{p}_{k-1} \text{CAR}_0(u) + \mathbf{p}_k \text{CAR}_1(u) + \mathbf{p}_{k+1} \text{CAR}_2(u) + \mathbf{p}_{k+2} \text{CAR}_3(u)
 \end{aligned} \tag{8.35}$$

donde los polinomios  $\text{CAR}_k(u)$  con  $k = 0, 1, 2, 3$  son las funciones de combinación de los *splines* cardinales. La Figura 8.28 proporciona un dibujo de las funciones base para los *splines* cardinales con  $t = 0$ .

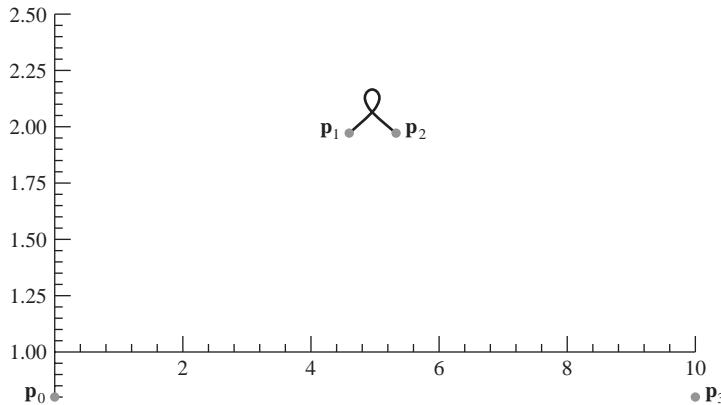
En las Figuras 8.29, 8.30 y 8.31 se proporcionan ejemplos de curvas producidas con las funciones de combinación de *splines* cardinales. En la Figura 8.29, se dibujan cuatro secciones de un *spline* cardinal para formar una curva cerrada. La primera parte de la curva se genera utilizando el conjunto de puntos de control  $\{\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3\}$ , la segunda curva se produce con el conjunto de puntos de control  $\{\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3, \mathbf{p}_0\}$ , la tercera curva tienen los puntos de control  $\{\mathbf{p}_2, \mathbf{p}_3, \mathbf{p}_0, \mathbf{p}_1\}$  y la última sección de la curva tiene los puntos de control  $\{\mathbf{p}_3, \mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2\}$ . En la Figura 8.30, se obtiene una curva cerrada con una única sección de *spline* cardinal cambiando la posición del tercer punto de control a la posición del segundo punto de control. En la Figura 8.31, se produce una sección de *spline* cardinal que se autointerseca estableciendo la posición del tercer punto de control muy cerca de la posición del segundo punto de control. La autointersección que resulta es debida a las restricciones en la pendiente de la curva en los puntos extremos  $\mathbf{p}_1$  y  $\mathbf{p}_2$ .



**FIGURA 8.29.** Una curva cerrada con cuatro secciones de *spline* cardinal, obtenida con una permutación circular de los puntos de control y con un parámetro de tensión  $t = 0$ .



**FIGURA 8.30.** Un bucle de *spline* cardinal producido con unos puntos extremos de la curva en la misma posición en coordenadas. Se asigna el valor 0 al parámetro de tensión.



**FIGURA 8.31.** Una parte de una curva con *spline* cardinal que se autointerseca producida con posiciones de los puntos extremos de la curva muy próximos en el espacio. El valor del parámetro de tensión se establece en el valor 0.

### Splines de Kochanek-Bartels

Estos polinomios de interpolación cúbicos son extensiones de los *splines* cardinales. Se introducen dos parámetros adicionales en las ecuaciones de las restricciones que definen los **splines de Kochanek-Bartels** para proporcionar mayor flexibilidad en el ajuste de las formas de las secciones de la curva.

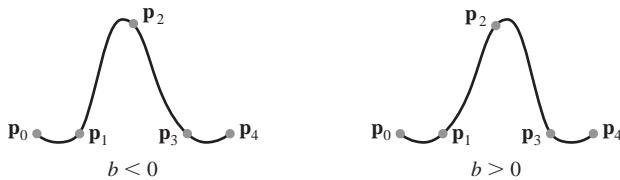
Dados cuatro puntos de control consecutivos, etiquetados como  $\mathbf{p}_{k-1}$ ,  $\mathbf{p}_k$ ,  $\mathbf{p}_{k+1}$  y  $\mathbf{p}_{k+2}$ , definimos las condiciones de los límites de una sección de curva de Kochanek-Bartels entre  $\mathbf{p}_k$  y  $\mathbf{p}_{k+1}$  del modo siguiente:

$$\begin{aligned}
 \mathbf{P}(0) &= \mathbf{p}_k \\
 \mathbf{P}(1) &= \mathbf{p}_{k+1} \\
 \mathbf{P}'(0)_{\text{in}} &= \frac{1}{2}(1-t)[(1+b)(1-c)(\mathbf{p}_k - \mathbf{p}_{k-1}) \\
 &\quad + (1-b)(1+c)(\mathbf{p}_{k+1} - \mathbf{p}_k)] \\
 \mathbf{P}'(1)_{\text{out}} &= \frac{1}{2}(1-t)[(1+b)(1+c)(\mathbf{p}_{k+1} - \mathbf{p}_k) \\
 &\quad + (1-b)(1-c)(\mathbf{p}_{k+2} - \mathbf{p}_{k+1})]
 \end{aligned} \tag{8.36}$$

donde  $t$  es el parámetro de tensión,  $b$  es el parámetro de desplazamiento (*bias*) y  $c$  es el parámetro de continuidad. En la formulación de Kochanek-Bartels, las derivadas paramétricas podrían no ser continuas en los límites de las secciones.

El parámetro de tensión  $t$  se interpreta del mismo modo que en la formulación del *spline* cardinal; es decir, controla lo suelto o apretado de las secciones de la curva. El parámetro de desplazamiento,  $b$ , se utiliza para ajustar la curvatura en cada extremo de una sección, para que las secciones de la curva se puedan desplazar hacia un extremo o hacia el otro (Figura 8.32). El parámetro  $c$  controla la continuidad del vector tangente en los límites de las secciones. Si a  $c$  se le asigna un valor distinto de cero, existe una discontinuidad en la pendiente de la curva en los límites de las secciones.

Los *splines* de Kochanek-Bartels se diseñaron para modelar trayectorias de animación. Concretamente, los cambios bruscos en el movimiento de un objeto se pueden simular con valores distintos de cero en el parámetro  $c$ . Estos cambios de movimiento se utilizan en dibujos animados, por ejemplo, cuando un personaje de dibujos animados se detiene rápidamente, cambia de dirección o colisiona con algún otro objeto.



**FIGURA 8.32.** Efecto del parámetro de desplazamiento sobre la forma de una sección de un *spline* de Kochanek-Bartels.

## 8.10 CURVAS CON SPLINES DE BÉZIER

Este método de aproximación con *spline* fue desarrollado por el ingeniero francés Pierre Bézier para su uso en el diseño de las carrocerías de automóviles Renault. Los *splines de Bézier* disponen de unas propiedades que los hacen especialmente útiles y convenientes para el diseño de curvas y superficies. Además, son fáciles de implementar. Por estas razones, los *splines de Bézier* están disponibles con mucha frecuencia en diversos sistemas CAD, en paquetes para gráficos generales y en paquetes heterogéneos de dibujo y pintura.

Por lo general, una parte de una curva de Bézier se puede ajustar a cualquier número de puntos de control, aunque algunos paquetes gráficos limitan el número de puntos de control a cuatro. El grado del polinomio de Bézier se determina con el número de puntos de control que hay que aproximar y con su posición relativa. Como en los *splines* de interpolación, podemos especificar la trayectoria de la curva de Bézier en las proximidades de los puntos de control utilizando funciones de combinación, una matriz de caracterización, o las condiciones en los límites. En las curvas generales de Bézier, sin restricciones en el número de puntos de control, la especificación de la función de fundido es la representación más conveniente.

### Ecuaciones de las curvas de Bézier

En primer lugar consideraremos el caso general con  $n + 1$  puntos de control, indicados como  $\mathbf{p}_k = (x_k, y_k, z_k)$ , donde  $k$  varía desde 0 a  $n$ . Estos puntos se combinan para producir el siguiente vector de posición  $\mathbf{P}(u)$ , que describe la trayectoria de una función de aproximación polinómica de Bézier entre  $\mathbf{p}_0$  y  $\mathbf{p}_n$ .

$$\mathbf{P}(u) = \sum_{k=0}^n \mathbf{p}_k \text{BEZ}_{k,n}(u), \quad 0 \leq u \leq 1 \quad (8.37)$$

Las funciones de combinación de Bézier  $\text{BEZ}_{k,n}(u)$  son los *polinomios de Bernstein*,

$$\text{BEZ}_{k,n}(u) = C(n, k)u^k(1 - u)^{n-k} \quad (8.38)$$

donde los parámetros  $C(n, k)$  son los coeficientes binomiales:

$$C(n, k) = \frac{n!}{k!(n-k)!} \quad (8.39)$$

La Ecuación vectorial 8.37 representa un sistema de tres ecuaciones paramétricas en las coordenadas individuales de la curva:

$$\begin{aligned} x(u) &= \sum_{k=0}^n x_k \text{BEZ}_{k,n}(u) \\ y(u) &= \sum_{k=0}^n y_k \text{BEZ}_{k,n}(u) \\ z(u) &= \sum_{k=0}^n z_k \text{BEZ}_{k,n}(u) \end{aligned} \quad (8.40)$$

En la mayoría de los casos, una curva de Bézier es un polinomio de un grado menor que el número de puntos de control designados: tres puntos generan una parábola, cuatro puntos una curva cúbica, y así sucesivamente. La Figura 8.33 muestra la apariencia de algunas curvas de Bézier para varias selecciones de puntos de control en el plano  $xy$  ( $z = 0$ ). Con ciertas posiciones de los puntos de control, sin embargo, obtenemos polinomios de Bézier degenerados. Por ejemplo, una curva de Bézier generada con tres puntos de control colineales es un segmento de línea recta. Y un conjunto de puntos de control que están todos en la misma posición producen una «curva» de Bézier que es un único punto.

Se pueden utilizar cálculos recursivos para obtener los valores sucesivos de los coeficientes binomiales del siguiente modo:

$$C(n, k) = \frac{n-k+1}{k} C(n, k-1) \quad (8.41)$$

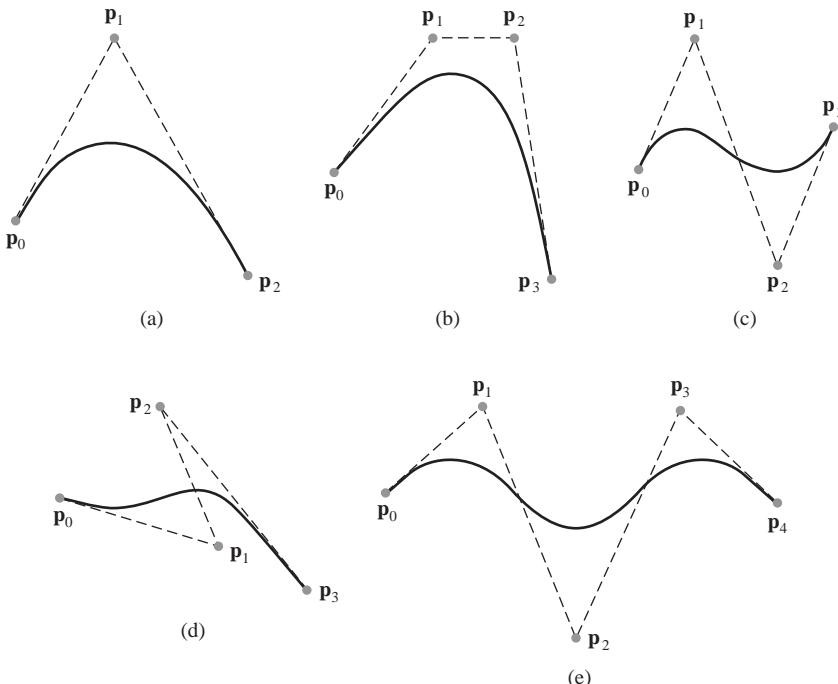
para  $n \geq k$ . También, las funciones de combinación de Bézier satisfacen la relación recursiva,

$$\text{BEZ}_{k,n}(u) = (1-u)\text{BEZ}_{k,n-1}(u) + u \text{ BEZ}_{k-1,n-1}(u), \quad n > k \geq 1 \quad (8.42)$$

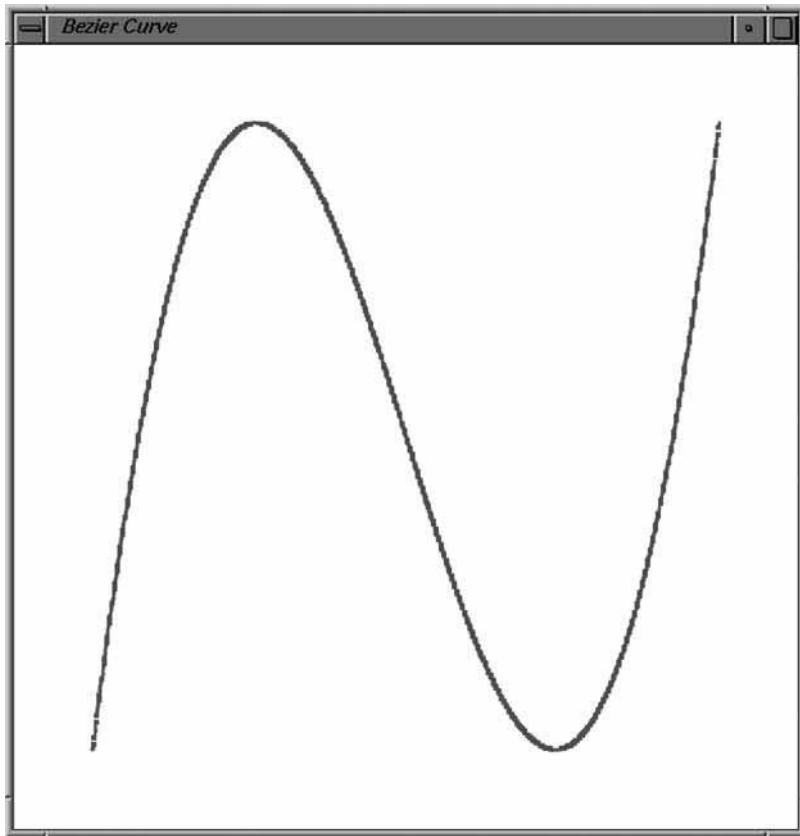
donde  $\text{BEZ}_{k,k} = u^k$  y  $\text{BEZ}_{0,k} = (1-u)^k$ .

### Ejemplo de un programa de generación de curvas de Bézier

En el siguiente programa se proporciona una implementación para el cálculo de las funciones de combinación de Bézier y la generación de una curva bidimensional con *splines* de Bézier cúbicos. Se definen cuatro puntos de control en el plano  $xy$ , y se dibujan 1000 posiciones de píxeles a lo largo de la trayectoria de la curva utilizando un grosor de pixel de 4. En el procedimiento `binomialCoeffs` se calculan los valores de los coeficientes binomiales y en el procedimiento `computeBezPt` se calculan las coordenadas a lo largo de la



**FIGURA 8.33.** Ejemplos de curvas de Bézier bidimensionales generadas con tres, cuatro y cinco puntos de control. Las líneas discontinuas conectan los puntos de control.



**FIGURA 8.34.** Una curva de Bézier visualizada mediante el programa de ejemplo.

trayectoria de la curva. Estos valores se pasan al procedimiento `bezier`, y se dibujan las posiciones de los píxeles utilizando las subrutinas de dibujo de puntos de OpenGL. De forma alternativa, podríamos haber aproximado la trayectoria de la curva con segmentos de línea recta, utilizando menos puntos. En la Sección 8.17 se estudian métodos más eficientes para generar las coordenadas a lo largo de la trayectoria de una curva con *splines*. En este ejemplo, los límites de las coordenadas universales se establecen para que se visualicen sólo los puntos de la curva dentro del visor (Figura 8.34). Si quisieramos también dibujar los puntos de control, el grafo de control, o el armazón convexo, necesitaríamos ampliar los límites de la ventana de recorte en coordenadas universales.

---

```
#include <GL/glut.h>
#include <stdlib.h>
#include <math.h>

/* Establece el tamaño inicial de la ventana de visualizacion. */
GLsizei winWidth = 600, winHeight = 600;

/* Establece el tamaño de la ventana de recorte en coordenadas universales. */
GLfloat xwcMin = -50.0, xwcMax = 50.0;
GLfloat ywcMin = -50.0, ywcMax = 50.0;
```

```

class wcPt3D {
    public:
        GLfloat x, y, z;
};

void init (void)
{
    /* Establece el color de la ventana de visualizacion en blanco. */
    glClearColor (1.0, 1.0, 1.0, 0.0);
}

void plotPoint (wcPt3D bezCurvePt)
{
    glBegin (GL_POINTS);
        glVertex2f (bezCurvePt.x, bezCurvePt.y);
    glEnd ();
}

/* Calcula los coeficientes binomiales C para un valor dado de n. */
void binomialCoeffs (GLint n, GLint * C)
{
    GLint k, j;

    for (k = 0; k <= n; k++) {
        /* Compute n!/(k!(n - k)!). */
        C [k] = 1;
        for (j = n; j >= k + 1; j--)
            C [k] *= j;
        for (j = n - k; j >= 2; j--)
            C [k] /= j;
    }
}

void computeBezPt (GLfloat u, wcPt3D * bezPt, GLint nCtrlPts,
                  wcPt3D * ctrlPts, GLint * C)
{
    GLint k, n = nCtrlPts - 1;
    GLfloat bezBlendFcn;

    bezPt->x = bezPt->y = bezPt->z = 0.0;

    /* Calcula las funciones de combinación y los puntos de control de
       combinación. */
    for (k = 0; k < nCtrlPts; k++) {
        bezBlendFcn = C [k] * pow (u, k) * pow (1 - u, n - k);
        bezPt->x += ctrlPts [k].x * bezBlendFcn;
        bezPt->y += ctrlPts [k].y * bezBlendFcn;
        bezPt->z += ctrlPts [k].z * bezBlendFcn;
    }
}

```

```

void bezier (wcPt3D * ctrlPts, GLint nCtrlPts, GLint nBezCurvePts)
{
    wcPt3D bezCurvePt;
    GLfloat u;
    GLint *C, k;

    /* Reserva espacio para los coeficientes binomiales */
    C = new GLint [nCtrlPts];

    binomialCoeffs (nCtrlPts - 1, C);
    for (k = 0; k <= nBezCurvePts; k++) {
        u = GLfloat (k) / GLfloat (nBezCurvePts);
        computeBezPt (u, &bezCurvePt, nCtrlPts, ctrlPts, C);
        plotPoint (bezCurvePt);
    }
    delete [ ] C;
}

void displayFcn (void)
{
    /* Establece un número de puntos de control de ejemplo y un número de
     * puntos de curva que se deben dibujar a lo largo de la curva de Bezier.
     */
    GLint nCtrlPts = 4, nBezCurvePts = 1000;

    wcPt3D ctrlPts [4] = { {-40.0, -40.0, 0.0}, {-10.0, 200.0, 0.0},
                           {10.0, -200.0, 0.0}, {40.0, 40.0, 0.0} };

    glClear (GL_COLOR_BUFFER_BIT);      // Borra la ventana de visualización.
    glPointSize (4);
    glColor3f (1.0, 0.0, 0.0);         // Establece el color de los puntos en rojo.

    bezier (ctrlPts, nCtrlPts, nBezCurvePts);
    glFlush ( );
}

void winReshapeFcn (GLint newWidth, GLint newHeight)
{
    /* Mantiene una relación de aspecto de valor 1.0. */
    glViewport (0, 0, newHeight, newHeight);

    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ( );

    gluOrtho2D (xwcMin, xwcMax, ywcMin, ywcMax);

    glClear (GL_COLOR_BUFFER_BIT);
}

void main (int argc, char** argv)
{

```

```

glutInit (&argc, argv);
glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
glutInitWindowPosition (50, 50);
glutInitWindowSize (winWidth, winHeight);
glutCreateWindow ("Curva de Bezier");

init ();
glutDisplayFunc (displayFcn);
glutReshapeFunc (winReshapeFcn);

glutMainLoop ();
}

```

## Propiedades de las curvas de Bézier

Una propiedad muy útil de una curva de Bézier es que la curva une el primer punto de control con el último. Por tanto, una característica básica de cualquier curva de Bézier es que,

$$\begin{aligned}\mathbf{P}(0) &= \mathbf{p}_0 \\ \mathbf{P}(1) &= \mathbf{p}_n\end{aligned}\quad (8.43)$$

Los valores de las primeras derivadas paramétricas de una curva de Bézier en los puntos extremos, se pueden calcular a partir de las coordenadas de los puntos de control del siguiente modo:

$$\begin{aligned}\mathbf{P}'(0) &= -n\mathbf{p}_0 + n\mathbf{p}_1 \\ \mathbf{P}'(1) &= -n\mathbf{p}_{n-1} + n\mathbf{p}_n\end{aligned}\quad (8.44)$$

En estas expresiones, vemos que la pendiente en el comienzo de la curva tiene la dirección de la línea que une los dos primeros puntos de control y la pendiente en el extremo final de la curva tiene la dirección de la línea que une los dos últimos puntos extremos. De forma similar, las segundas derivadas paramétricas de una curva de Bézier en sus puntos extremos se calcula como sigue:

$$\begin{aligned}\mathbf{P}''(0) &= n(n - 1)[(\mathbf{p}_2 - \mathbf{p}_1) - (\mathbf{p}_1 - \mathbf{p}_0)] \\ \mathbf{P}''(1) &= n(n - 1)[(\mathbf{p}_{n-2} - \mathbf{p}_{n-1}) - (\mathbf{p}_{n-1} - \mathbf{p}_n)]\end{aligned}\quad (8.45)$$

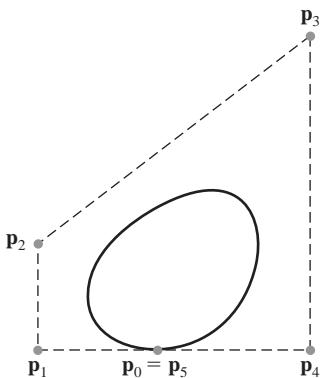
Otra propiedad importante de cualquier curva de Bézier es que se encuentra dentro del armazón convexo (polígono convexo) de los puntos de control. Esto se deriva del hecho de que las funciones de combinación de Bézier son todas positivas y su suma es siempre 1,

$$\sum_{k=0}^n \text{BEZ}_{k,n}(u) = 1 \quad (8.46)$$

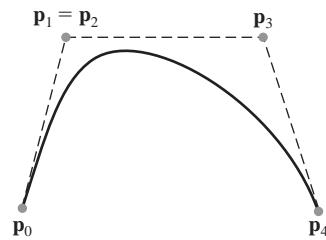
de forma que cualquier punto de la curva es simplemente la suma ponderada de los puntos de control. La propiedad del armazón convexo de una curva de Bézier garantiza que el polinomio sigue suavemente los puntos de control sin oscilaciones erráticas.

## Técnicas de diseño utilizando curvas de Bézier

Una curva de Bézier cerrada se genera cuando establecemos el último punto de control en la posición del primer punto de control, como en el ejemplo mostrado en la Figura 8.35. También, especificando múltiples



**FIGURA 8.35.** Una curva de Bézier cerrada generada especificando el primer y el último punto de control en la misma posición.



**FIGURA 8.36.** Se puede conseguir que una curva de Bézier pase más cerca de una posición dada asignando múltiples puntos de control a dicha posición.

puntos de control en una única posición proporciona más peso a dicha posición. En la Figura 8.36, una única posición de coordenadas se introduce como dos puntos de control y la curva resultante se desplaza hacia las proximidades de esta posición.

Podemos ajustar una curva de Bézier a cualquier número de puntos de control, pero esto requiere el cálculo de funciones polinómicas de grado superior. Cuando hay que generar curvas complicadas, éstas se pueden formar uniendo varias secciones de curvas de Bézier de menor grado. La generación de secciones de curvas de Bézier más pequeñas también proporciona un mejor control local sobre la forma de la curva. Ya que las curvas de Bézier tienen la importante propiedad de que la tangente a la curva en un punto extremo tiene la dirección de la línea que une aquel punto extremo con el punto de control adyacente. Por tanto, para obtener continuidad de primer orden entre las secciones de la curva, podemos seleccionar los puntos de control  $p_0'$  y  $p_1'$  para la siguiente sección de la curva de forma que estén sobre la misma línea que los puntos de control  $p_{n-1}$  y  $p_n$  de la parte anterior (Figura 8.37). Si la primera sección de la curva tiene  $n$  puntos de control y la siguiente sección tiene  $n'$  puntos de control, entonces hacemos coincidir las tangentes a la curva colocando el punto de control  $p_1'$  en la posición

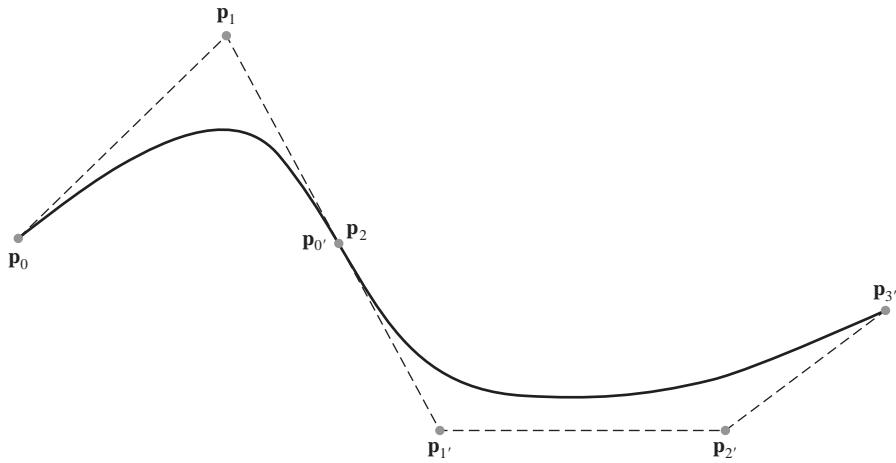
$$p_1' = p_n + \frac{n}{n'}(p_n - p_{n-1}) \quad (8.47)$$

Para simplificar la colocación de  $p_1'$ , podemos requerir sólo continuidad geométrica y colocar  $p_1'$  en cualquier sitio sobre la línea que une  $p_{n-1}$  y  $p_n$ .

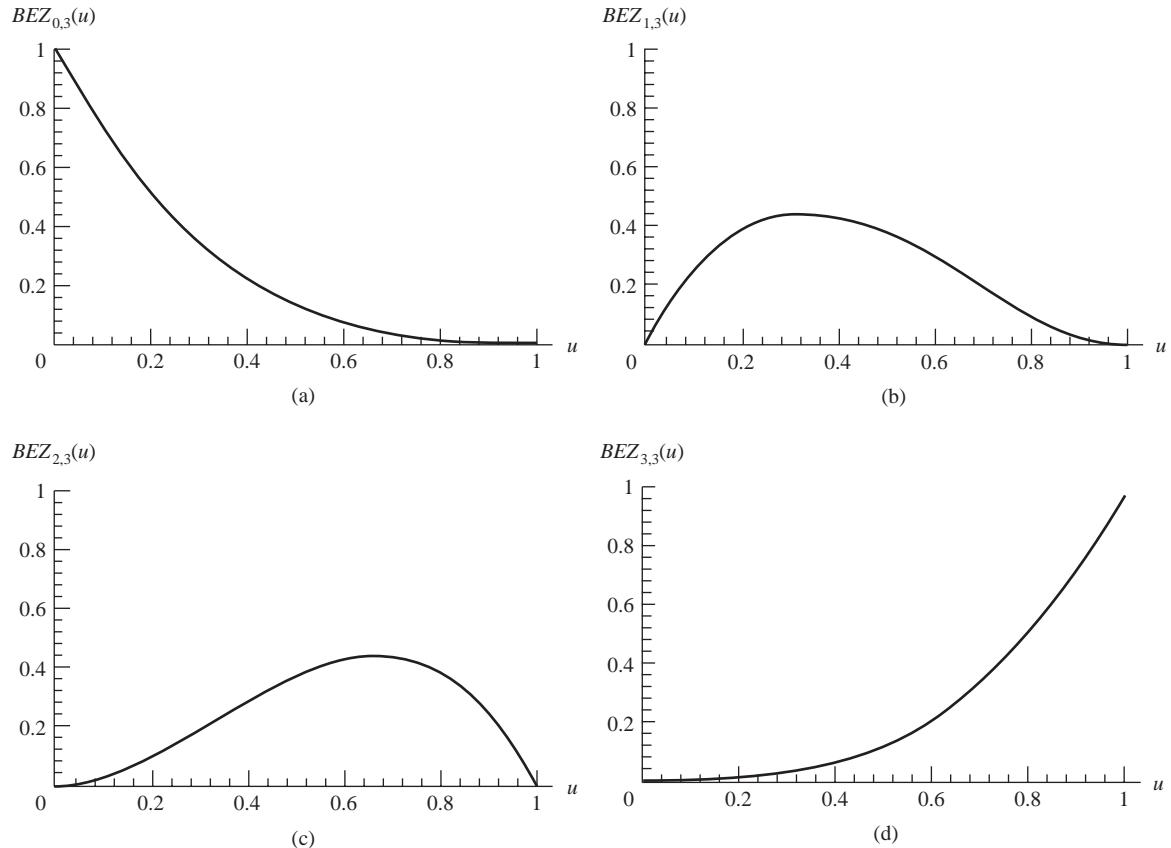
Obtenemos continuidad  $C^2$  utilizando las expresiones de las Ecuaciones 8.45 para emparejar las segundas derivadas paramétricas de las dos partes de Bézier adyacentes. Esto establece una posición para el punto de control  $p_2'$ , además de las posiciones fijadas para  $p_0'$  y  $p_1'$  que necesitamos para obtener continuidad  $C^0$  y  $C^1$ . Sin embargo, requerir continuidad de segundo orden para las secciones de la curva de Bézier puede ser una restricción innecesaria. Esto es particularmente cierto en curvas cúbicas, que tienen sólo cuatro puntos de control en cada parte. En este caso, la continuidad de segundo orden fija la posición de los tres primeros puntos de control y nos deja sólo un punto para que podamos ajustar la forma del segmento de curva.

## Curvas de Bézier cúbicas

Muchos paquetes gráficos proporcionan funciones para visualizar únicamente *splines* cúbicos. Esto permite una flexibilidad de diseño razonable en tanto que evita el incremento de cálculos que necesitan los polinomios de orden más elevado. Las curvas de Bézier cúbicas se generan con cuatro puntos de control. Las cuatro



**FIGURA 8.37.** Curva de aproximación por tramos formada por dos secciones de Bézier. La continuidad de orden cero y de primer orden se logra en las dos secciones de la curva estableciendo  $p_0' = p_2$  y definiendo  $p_1'$  sobre la línea formada por los puntos  $p_1$  y  $p_2$ .



**FIGURA 8.38.** Las cuatro funciones de combinación para curvas cúbicas ( $n = 3$ ).

funciones de combinación para curvas de Bézier cúbicas, obtenidas sustituyendo  $n = 3$  en la Ecuación 8.38, son:

$$\begin{aligned} \text{BEZ}_{0,3} &= (1 - u)^3 \\ \text{BEZ}_{1,3} &= 3u(1 - u)^2 \\ \text{BEZ}_{2,3} &= 3u^2(1 - u) \\ \text{BEZ}_{3,3} &= u^3 \end{aligned} \quad (8.48)$$

En la Figura 8.38 se proporcionan gráficas de las cuatro funciones cúbicas de fundido de Bézier. La forma de las funciones de combinación determina la influencia de los puntos de control en la forma de la curva para los valores del parámetro  $u$  en el rango que varía desde 0 a 1. Para  $u = 0$ , la única función de fundido distinta de cero es  $\text{BEZ}_{0,3}$ , que tiene el valor 1. Para  $u = 1$ , la única función de fundido distinta de cero es  $\text{BEZ}_{3,3}(1) = 1$ . Por tanto, una curva de Bézier cónica siempre comienza por el punto de control  $\mathbf{p}_0$  y termina en el punto de control  $\mathbf{p}_3$ . Las otras funciones,  $\text{BEZ}_{1,3}$  y  $\text{BEZ}_{2,3}$ , influyen en la forma de la curva para valores intermedios del parámetro  $u$ , de modo que la curva resultante tiende hacia los puntos  $\mathbf{p}_1$  y  $\mathbf{p}_2$ . La función de fundido  $\text{BEZ}_{1,3}$  tiene su máximo en  $u = 1/3$ , y  $\text{BEZ}_{2,3}$  en  $u = 2/3$ .

En la Figura 8.38 apreciamos que cada una de las cuatro funciones de combinación es distinta de cero sobre todo el rango del parámetro  $u$  entre los puntos extremos. Por tanto, las curvas de Bézier no permiten *control local* de la forma de la curva. Si reposicionamos cualquiera de los puntos de control, se ve afectada toda la curva.

En los puntos extremos de la curva de Bézier cónica, las primeras derivadas (pendientes) paramétricas son:

$$\mathbf{P}'(0) = 3(\mathbf{p}_1 - \mathbf{p}_0), \quad \mathbf{P}'(1) = 3(\mathbf{p}_3 - \mathbf{p}_2)$$

Y las segundas derivadas paramétricas son

$$\mathbf{P}''(0) = 6(\mathbf{p}_0 - 2\mathbf{p}_1 + \mathbf{p}_2), \quad \mathbf{P}''(1) = 6(\mathbf{p}_1 - 2\mathbf{p}_2 + \mathbf{p}_3)$$

Podemos construir curvas complejas con *splines* utilizando una serie de secciones de curvas cónicas de Bézier. Utilizando expresiones para las derivadas paramétricas, podemos igualar las tangentes a la curva para lograr continuidad  $C^1$  entre las secciones de la curva. Y podríamos utilizar las expresiones para las segundas derivadas para obtener continuidad  $C^2$ , aunque esto nos deja sin opciones en la colocación de los tres primeros puntos de control.

Se obtiene una formulación matemática para la función de curva de Bézier cónica, desarrollando las expresiones polinómicas de las funciones de combinación y reestructurando las ecuaciones como se indica a continuación,

$$\mathbf{P}(u) = [u^3 \quad u^2 \quad u \quad 1] \cdot \mathbf{M}_{\text{Bez}} \cdot \begin{bmatrix} \mathbf{p}_0 \\ \mathbf{p}_1 \\ \mathbf{p}_2 \\ \mathbf{p}_3 \end{bmatrix} \quad (8.49)$$

donde la **matriz de Bézier** es:

$$\mathbf{M}_{\text{Bez}} = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \quad (8.50)$$

También podríamos introducir parámetros adicionales que permitan ajustar la «tensión» y el «desplazamiento» de la curva como hicimos con los *splines* de interpolación. Pero los *splines* B que son más versátiles, así como los *splines*  $\beta$ , se proporcionan a menudo con esta capacidad.

## 8.11 SUPERFICIES DE BÉZIER

---

Dos familias de curvas de Bézier ortogonales se pueden utilizar para diseñar una superficie de un objeto. La función paramétrica del vector de una superficie de Bézier se crea con el producto cartesiano de las funciones de combinación de Bézier:

$$\mathbf{P}(u, v) = \sum_{j=0}^m \sum_{k=0}^n \mathbf{p}_{j,k} \text{BEZ}_{j,m}(v) \text{BEZ}_{k,n}(u) \quad (8.51)$$

donde  $\mathbf{p}_{j,k}$  especifica la localización de los  $(m + 1)$  por  $(n + 1)$  puntos de control.

La Figura 8.39 muestra dos dibujos de superficies de Bézier. Los puntos de control se unen mediante líneas discontinuas, y las líneas continuas muestran las curvas con  $u$  constante y  $v$  constante. Cada curva con  $u$  constante se dibuja variando  $v$  en el intervalo de 0 a 1, manteniendo en un valor fijo  $u$  dentro de este intervalo unitario. Las curvas con  $v$  constante se dibujan de forma similar.

Las superficies de Bézier tienen las mismas propiedades que las curvas de Bézier y proporcionan un método conveniente para aplicaciones interactivas de diseño. Para especificar las posiciones de los puntos de control con coordenadas tridimensionales, podríamos en primer lugar construir una cuadrícula rectangular en el plano  $xy$  «de tierra». A continuación, elegimos las alturas sobre el plano de tierra en las intersecciones de la cuadrícula como los valores de la coordenada  $z$  de los puntos de control. Los parches de superficie se pueden representar con polígonos y sombrear utilizando las técnicas de sombreado expuestas en el Capítulo 10.

La Figura 8.40 muestra una superficie formada por dos secciones de superficie de Bézier. Como en el caso de las curvas, se garantiza una transición suave de una sección a otra estableciendo tanto continuidad de orden cero como de primer orden en la línea límite. La continuidad de orden cero se obtiene haciendo coincidir los puntos de control en los límites. La continuidad de primer orden se obtiene seleccionando los puntos de control a lo largo de una línea recta que recorre el límite y manteniendo una relación constante entre los segmentos de línea colineales para cada conjunto de puntos de control específicos de los límites de las secciones.

## 8.12 CURVAS CON SPLINES B

---

Esta clase de *splines* es la más profusamente utilizada y las funciones de **splines B** están disponibles habitualmente en los sistemas CAD y en muchos paquetes de programación de gráficos. Al igual que los *splines* de Bézier, los **splines B** se generan aproximando un conjunto de puntos de control. Pero los **splines B** presentan dos ventajas frente a los *splines* de Bézier: (1) el grado de un polinomio de un *spline B* se puede establecer de forma independiente al número de puntos de control (con ciertas limitaciones), y (2) los *splines B* permiten control local sobre la forma de un *spline*. La desventaja es que los *splines B* son más complejos que los *splines* de Bézier.

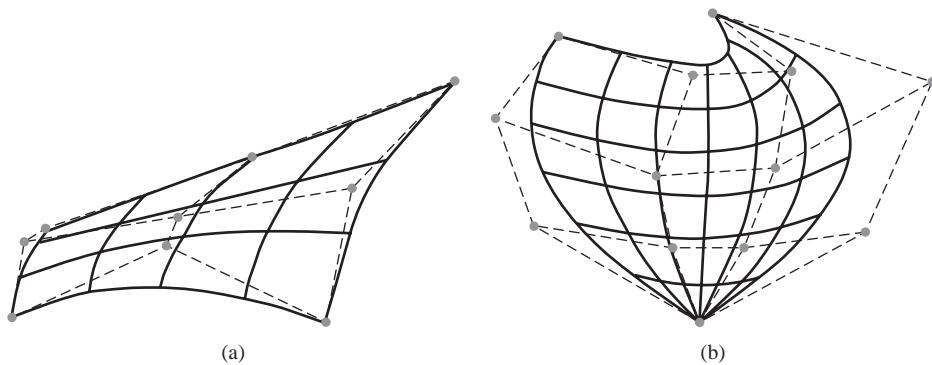
### Ecuaciones de una curva con *splines B*

Podemos escribir una expresión general para el cálculo de las coordenadas a lo largo de una curva con *splines B* utilizando la formulación con funciones de combinación del siguiente modo:

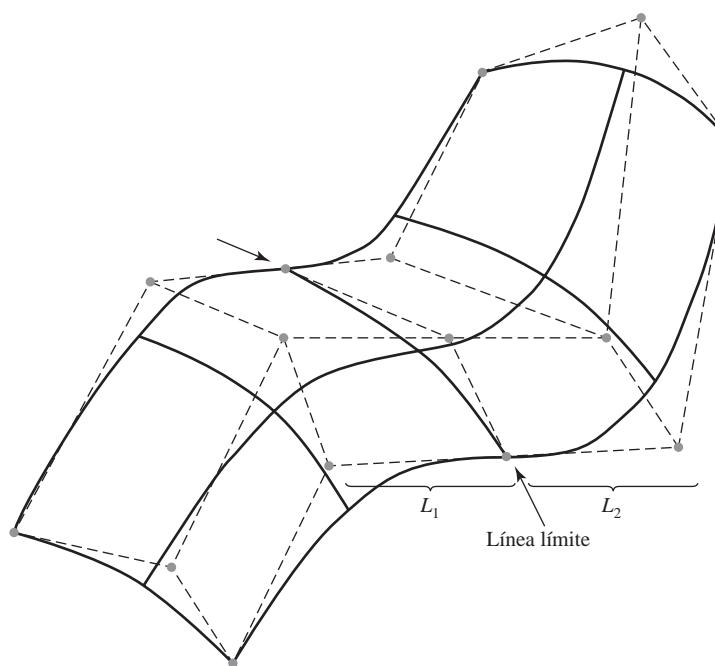
$$P(u) = \sum_{k=0}^n p_k B_{k,d}(u), \quad u_{\min} \leq u \leq u_{\max}, \quad 2 \leq d \leq n+1 \quad (8.52)$$

donde  $\mathbf{p}_k$  define un conjunto de entrada de  $n + 1$  puntos de control. Existen varias diferencias entre esta formulación con *splines B* y las expresiones de una curva con *splines* de Bézier. El rango de variación del parámetro  $u$  ahora depende de cómo elijamos los otros parámetros de los *splines B*. Y las funciones de combina-

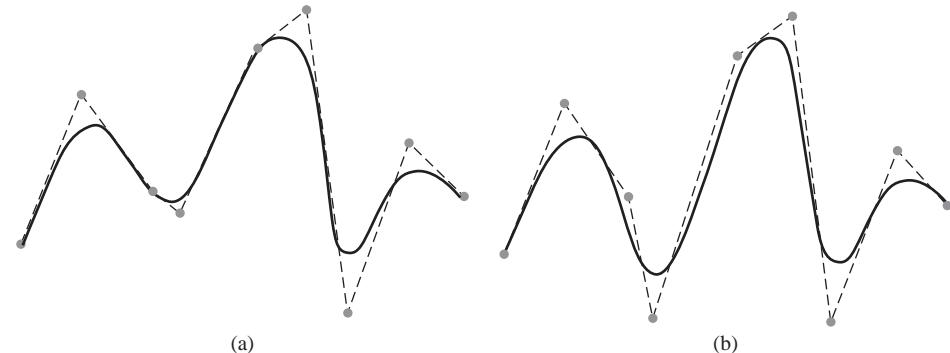
ción de los *splines B*  $B_{k,d}$  son polinomios de grado  $d - 1$ , donde  $d$  es el **parámetro de grado**. (A veces al parámetro  $d$  se le llama «orden» del polinomio, pero esto puede confundir, ya que el término orden se utiliza a menudo también para indicar simplemente el grado del polinomio, que es  $d - 1$ .) Al parámetro de grado  $d$  se le puede asignar cualquier valor entero dentro del rango que varía desde 2 hasta el número de puntos de control  $n + 1$ . Realmente, podríamos también establecer el valor del parámetro de grado en 1, pero entonces nuestra «curva» sería sólo una gráfica de puntos con los puntos de control. El control local en los *splines B* se logra definiendo las funciones de combinación sobre subintervalos del rango total de variación de  $u$ .



**FIGURA 8.39.** Superficies de Bézier en modelo alámbrico construidas con (a) nueve puntos de control dispuestos en una malla de tamaño 3 por 3 y (b) dieciséis puntos de control dispuestos en una malla de tamaño 4 por 4. Las líneas discontinuas unen los puntos de control.



**FIGURA 8.40.** Una superficie de Bézier compuesta construida con dos secciones de Bézier, unidas por la línea límite indicada. Las líneas discontinuas unen los puntos de control. La continuidad de primer orden se establece haciendo que la relación entre la longitud  $L_1$  y la longitud  $L_2$  sea constante para cada línea colineal de puntos de control en el límite entre las secciones de la superficie.



**FIGURA 8.41.** Modificación local de una curva con *splines B*. Cambiando uno de los puntos de control en (a) se genera la curva mostrada en (b), que sólo se modifica en la vecindad del punto de control alterado.

Las funciones de combinación para las curvas con *splines B* se definen mediante las fórmulas recursivas de Cox-deBoor:

$$B_{k,1}(u) = \begin{cases} 1 & \text{si } u_k \leq u \leq u_{k+1} \\ 0 & \text{traducir} \end{cases} \quad (8.53)$$

$$B_{k,d}(u) = \frac{u - u_k}{u_{k+d-1} - u_k} B_{k,d-1}(u) + \frac{u_{k+d} - u}{u_{k+d} - u_{k+1}} B_{k+1,d-1}(u)$$

donde cada función de fundido se define sobre  $d$  subintervalos del rango total de variación de  $u$ . Cada punto extremo de los subintervalos se denomina **nudo**, y el conjunto entero de los puntos extremos seleccionados de los subintervalos se denomina **vector de nudos**. Podemos elegir valores cualesquiera para los puntos extremos de los subintervalos con tal de que  $u_j \leq u_{j+1}$ . Los valores de  $u_{\min}$  y  $u_{\max}$  entonces dependen del número de puntos de control que seleccionemos, del valor que elijamos para el parámetro de grado  $d$ , y de cómo establezcamos los subintervalos (vector de nudos). Ya que es posible elegir los elementos del vector de nudos de modo que el resultado de la evaluación de algunos denominadores de los cálculos de Cox-deBoor sea nulo, esta formulación asume que a cualquier término cuya evaluación resulte 0/0 hay que asignarle el valor 0.

La Figura 8.41 muestra las características de control local de los *splines B*. Además del control local, los *splines B* nos permiten variar el número de puntos de control utilizados para diseñar una curva sin cambiar el grado del polinomio. Y podemos incrementar el número de valores del vector de nudos para ayudar al diseño de la curva. Cuando se hace esto, sin embargo, debemos añadir puntos de control ya que el tamaño del vector de nudos depende del parámetro  $n$ .

Las curvas con *splines B* tienen las siguientes propiedades:

- La curva polinómica tiene grado  $d - 1$  y continuidad  $C^{d-2}$  sobre el rango de variación de  $u$ .
- En los  $n + 1$  puntos de control, la curva se describe con  $n + 1$  funciones de combinación.
- Cada función de fundido  $B_{k,d}$  se define sobre  $d$  subintervalos del rango total de variación de  $u$ , comenzando por el nodo de valor  $u_k$ .
- El rango de variación del parámetro  $u$  se divide en  $n + d$  subintervalos con los  $n + d + 1$  valores especificados en el vector de nudos.
- Dado que los valores de los nudos están etiquetados como  $\{u_0, u_1, \dots, u_{n+d}\}$ , la curva con *splines B* resultante está definida únicamente en el intervalo que varía desde el valor de nodo  $u_{d-1}$  hasta el valor de nodo  $u_{n+1}$ . (Algunas funciones de combinación no están definidas fuera de este intervalo.)
- Cada sección de la curva con *splines* (entre dos valores de nodo sucesivos) se ve influenciada por  $d$  puntos de control.

- Cualquier punto de control puede afectar a la forma de a lo sumo  $d$  secciones de la curva.

Además, una curva con *splines B* se encuentra dentro del casco convexo de a lo sumo  $d + 1$  puntos de control, de modo que los *splines B* están ajustadamente ligados a las posiciones de entrada. Para algún valor de  $u$  del intervalo desde el valor de nudo  $u_{d-1}$  a  $u_{n+1}$ , la suma sobre todas las funciones base es 1:

$$\sum_{k=0}^n B_{k,d}(u) = 1 \quad (8.54)$$

Dadas las posiciones de los puntos de control y el valor del parámetro de grado  $d$ , entonces necesitamos especificar los valores de los nudos para obtener las funciones de combinación utilizando las relaciones de recurrencia 8.53. Hay tres clasificaciones generales de los vectores de nudos: uniforme, uniforme abierto y no uniforme. Los *splines B* se describen habitualmente según la clase seleccionada de vector de nudos.

### Curvas con *splines B* periódicos y uniformes

Cuando el espaciado entre valores de nudos es constante, la curva resultante se denomina **spline B uniforme**. Por ejemplo, podemos establecer un vector uniforme de nudos como el siguiente:

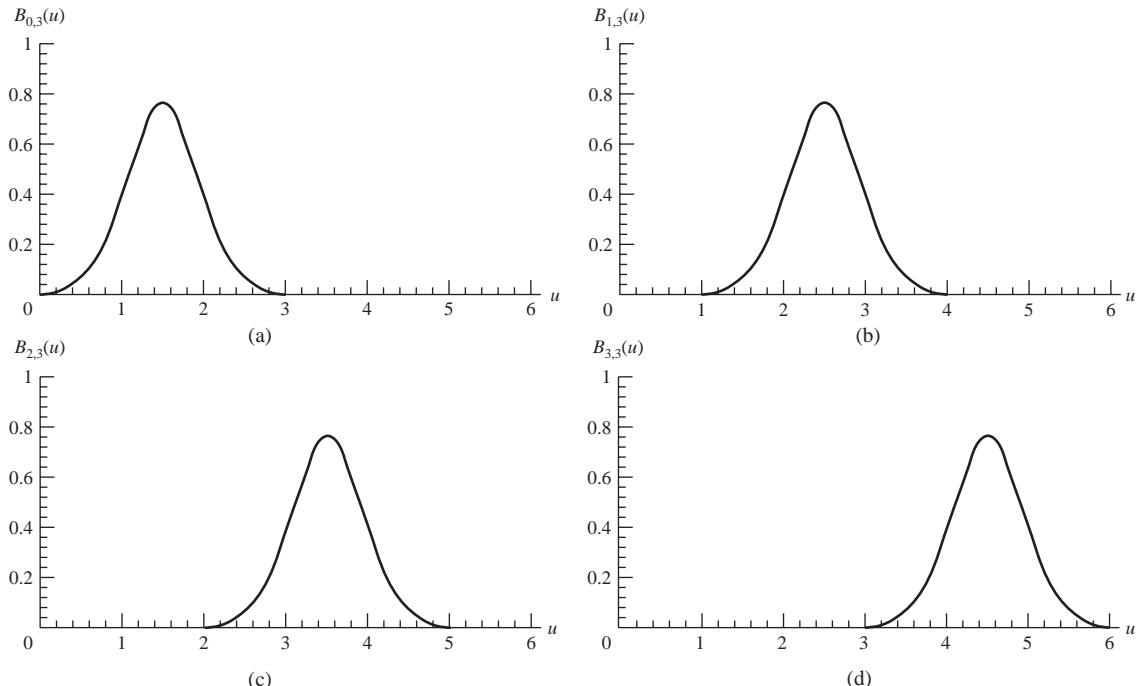
$$\{-1.5, -1.0, -0.5, 0.0, 0.5, 1.0, 1.5, 2.0\}$$

A menudo los valores de los nudos están normalizados en el rango que varía entre 0 y 1, como en,

$$\{0.0, 0.2, 0.4, 0.6, 0.8, 1.0\}$$

Es conveniente en muchas aplicaciones establecer valores uniformes de nudos con una separación de 1 y un valor inicial de 0. El siguiente vector de nudos es un ejemplo de este modo de especificación.

$$\{0, 1, 2, 3, 4, 5, 6, 7\}$$



**FIGURA 8.42.** Funciones de combinación de *splines B* periódicos para  $n = d = 3$  y un vector de nudos uniforme y entero.

Los *splines B* uniformes tienen funciones de combinación **periódicas**. Es decir, para valores dados de  $n$  y  $d$ , todas las funciones de combinación tienen la misma forma. Cada función sucesiva de fundido es simplemente una versión desplazada de la función anterior:

$$B_{k,d}(u) = B_{k+1,d}(u + \Delta u) = B_{k+2,d}(u + 2\Delta u) \quad (8.55)$$

donde  $\Delta u$  es el intervalo entre valores de nudo adyacentes. La Figura 8.42 muestra las funciones de combinación de *splines B* uniformes y cuadráticos generados en el siguiente ejemplo para una curva con cuatro puntos de control.

### Ejemplo 8.1 Splines B uniformes y cuadráticos

Para ilustrar la formulación de las funciones de combinación de *splines B* para un vector de nudos uniforme y entero, seleccionamos los valores de los parámetros  $n = d = 3$ . El vector de nudos debe tener  $n + d + 1 = 7$  valores de nudo:

$$\{0, 1, 2, 3, 4, 5, 6\}$$

y el rango de variación del parámetro  $u$  es de 0 a 6, con  $n + d = 6$  subintervalos.

Cada una de las cuatro funciones de combinación abarca  $d = 3$  subintervalos del rango total de variación de  $u$ . Utilizando las relaciones de recurrencia 8.53, obtenemos la primera función de fundido:

$$B_{0,3}(u) = \begin{cases} \frac{1}{2}u^2, & \text{para } 0 \leq u < 1 \\ \frac{1}{2}u(2-u) + \frac{1}{2}(u-1)(3-u), & \text{para } 1 \leq u < 2 \\ \frac{1}{2}(3-u)^2 & \text{para } 2 \leq u < 3 \end{cases}$$

Obtenemos la siguiente función periódica de fundido utilizando la relación 8.55, sustituyendo  $u$  por  $u - 1$  en  $B_{0,3}$  y desplazando las posiciones de comienzo hacia arriba una unidad:

$$B_{1,3}(u) = \begin{cases} \frac{1}{2}(u-1)^2, & \text{para } 1 \leq u < 2 \\ \frac{1}{2}(u-1)(3-u) + \frac{1}{2}(u-2)(4-u), & \text{para } 2 \leq u < 3 \\ \frac{1}{2}(4-u)^2 & \text{para } 3 \leq u < 4 \end{cases}$$

Similarmente, las dos restantes funciones periódicas se obtienen desplazando sucesivamente  $B_{1,3}$  hacia la derecha:

$$B_{2,3}(u) = \begin{cases} \frac{1}{2}(u-2)^2, & \text{para } 2 \leq u < 3 \\ \frac{1}{2}(u-2)(4-u) + \frac{1}{2}(u-3)(5-u), & \text{para } 3 \leq u < 4 \\ \frac{1}{2}(5-u)^2 & \text{para } 4 \leq u < 5 \end{cases}$$

$$B_{3,3}(u) = \begin{cases} \frac{1}{2}(u-3)^2, & \text{para } 3 \leq u < 4 \\ \frac{1}{2}(u-3)(5-u) + \frac{1}{2}(u-4)(6-u), & \text{para } 4 \leq u < 5 \\ \frac{1}{2}(6-u)^2 & \text{para } 5 \leq u < 6 \end{cases}$$

En la Figura 8.42 se proporciona una gráfica de las cuatro funciones periódicas y cuadráticas de fundido, que muestra la característica local de los *splines B*. El primer punto de control se multiplica por la función de fundido  $B_{0,3}(u)$ . Por tanto, el cambio de la posición del primer punto de control sólo afecta a la forma de la curva hasta  $u = 3$ . De forma similar, el último punto de control influye en la forma de la curva con *splines* en el intervalo donde  $B_{3,3}$  está definida.

La Figura 8.42 también muestra los límites de la curva con *splines B* para este ejemplo. Todas las funciones existen en el intervalo que varía desde  $u_{d-1} = 2$  hasta  $u_{n+1} = 4$ . Por debajo de 2 y por encima de 4, no todas las funciones de combinación existen. Este intervalo desde 2 a 4, es el rango de variación de la curva polinómica, y el intervalo en el que la Ecuación 8.54 es válida. Por tanto, la suma de todas las funciones de combinación vale 1 dentro de este intervalo. Fuera de este intervalo, no podemos sumar todas las funciones de combinación, ya que no todas ellas están definidas por debajo de 2 y por encima de 4.

Ya que el rango de variación de la curva polinómica que resulta es desde 2 a 4, podemos determinar el punto inicial y el punto final de la curva evaluando las funciones de combinación en estos puntos para obtener:

$$\mathbf{P}_{\text{ini}} = \frac{1}{2}(\mathbf{p}_0 + \mathbf{p}_1), \quad \mathbf{P}_{\text{fin}} = \frac{1}{2}(\mathbf{p}_2 + \mathbf{p}_3)$$

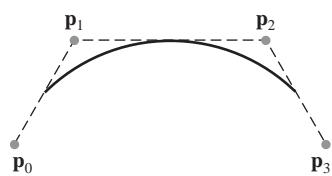
Por tanto, la curva comienza en el punto medio entre los dos primeros puntos de control y termina en el punto medio de los dos últimos puntos de control.

También podemos determinar las derivadas paramétricas en el punto inicial y en el punto final de la curva. Tomando las derivadas de las funciones de combinación y sustituyendo el parámetro  $u$  por su valor en los puntos extremos, encontramos que:

$$\mathbf{P}'_{\text{ini}} = \mathbf{p}_1 - \mathbf{p}_0, \quad \mathbf{P}'_{\text{fin}} = \mathbf{p}_3 - \mathbf{p}_2$$

La pendiente paramétrica de la curva en el punto inicial es paralela a la línea que une los dos primeros puntos de control, y la pendiente paramétrica en el punto final de la curva es paralela a la línea que une los dos últimos puntos de control.

En la Figura 8.43 se proporciona un dibujo de una curva con *splines B* cuadráticos y periódicos para cuatro puntos de control situados en el plano  $xy$ .



**FIGURA 8.43.** Un *spline B* cuadrático y periódico ajustado a cuatro puntos de control situados en el plano  $xy$ .

En el ejemplo anterior, observamos que la curva cuadrática comienza entre los dos primeros puntos de control y termina en un punto entre los dos últimos puntos de control. Este resultado es válido para los *splines B* cuadráticos y periódicos.

nes B cuadráticos y periódicos ajustados a cualquier número de puntos de control distintos. Por lo general, en los polinomios de orden más elevado, el punto de comienzo y el punto final son cada uno medias ponderadas de  $d - 1$  puntos de control. Podemos aproximar a cualquier punto de control cualquier curva con *splines* especificando dicho punto múltiples veces.

Las expresiones generales de las condiciones en los límites de los *splines* B periódicos se pueden obtener volviendo a parametrizar las funciones de combinación, de manera que el rango de variación del parámetro  $u$  se transforme en el intervalo que varía desde 0 a 1. Las condiciones de comienzo y fin se obtienen entonces para  $u = 0$  y  $u = 1$ .

### Curvas con *splines* B cúbicos y periódicos

Ya que los *splines* B cúbicos y periódicos se utilizan habitualmente en los paquetes gráficos, tendremos en cuenta la formulación de esta clase de *splines*. Los *splines* periódicos son particularmente útiles para generar ciertas curvas cerradas. Por ejemplo, la curva cerrada de la Figura 8.44 se puede generar por secciones especificando cíclicamente cuatro de los seis puntos de control de cada sección. También, si las posiciones en coordenadas de tres puntos de control son idénticas, la curva pasa a través de dicho punto.

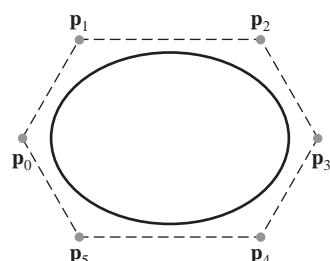
En curvas con *splines* B cúbicos,  $d = 4$  y cada función de fundido abarca cuatro subintervalos del rango total de variación de  $u$ . Si tenemos que ajustar la cúbica a cuatro puntos de control, entonces podríamos utilizar el vector entero de nudos:

$$\{0, 1, 2, 3, 4, 5, 6, 7\}$$

y las relaciones de recurrencia 8.53 para obtener las funciones periódicas de fundido, como hicimos en la última sección de los *splines* B cuadráticos y periódicos.

Para obtener las ecuaciones de la curva para un *spline* B periódico y cúbico, consideraremos una formulación alternativa comenzando por las condiciones en los límites y obteniendo las funciones de combinación normalizadas al intervalo  $0 \leq u \leq 1$ . Utilizando esta formulación, también podemos obtener fácilmente la matriz característica. Las condiciones en los límites para *splines* B cúbicos y periódicos con cuatro puntos de control, etiquetados como  $\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2$  y  $\mathbf{p}_3$  son:

$$\begin{aligned} \mathbf{P}(0) &= \frac{1}{6}(\mathbf{p}_0 + 4\mathbf{p}_1 + \mathbf{p}_2) \\ \mathbf{P}(1) &= \frac{1}{6}(\mathbf{p}_1 + 4\mathbf{p}_2 + \mathbf{p}_3) \\ \mathbf{P}'(0) &= \frac{1}{2}(\mathbf{p}_2 - \mathbf{p}_0) \\ \mathbf{P}'(1) &= \frac{1}{2}(\mathbf{p}_3 - \mathbf{p}_1) \end{aligned} \tag{8.56}$$



**FIGURA 8.44.** Un *spline* B cúbico por tramos, periódico y cerrado construido utilizando una especificación cíclica de cuatro puntos de control en cada sección de la curva.

Estas condiciones en los límites son similares a las de los *splines* cardinales: las secciones de la curva se definen con cuatro puntos de control y las derivadas paramétricas (pendientes) en el comienzo y en el final de cada sección de la curva son paralelas a las cuerdas que unen los puntos de control adyacentes. La sección de la curva con *splines* B comienza en una posición cercana a  $\mathbf{p}_1$  y termina en una posición cercana a  $\mathbf{p}_2$ .

Una formulación matricial de un *spline* B cúbico y periódico con cuatro puntos de control se puede escribir del siguiente modo:

$$\mathbf{P}(u) = [u^3 \quad u^2 \quad u \quad 1] \cdot \mathbf{M}_B \cdot \begin{bmatrix} \mathbf{p}_0 \\ \mathbf{p}_1 \\ \mathbf{p}_2 \\ \mathbf{p}_3 \end{bmatrix} \quad (8.57)$$

donde la matriz del *spline* B para polinomios cúbicos y periódicos es:

$$\mathbf{M}_B = \frac{1}{6} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix} \quad (8.58)$$

Esta matriz se puede obtener resolviendo los coeficientes en una expresión general polinomial cúbica, utilizando las cuatro condiciones específicas en los límites.

También podemos modificar las ecuaciones del *spline* B para incluir un parámetro de tensión  $t$  (como en los *splines* cardinales). La matriz de un *spline* B periódico y cúbico, en la que se incluye el parámetro de tensión  $t$ , es:

$$\mathbf{M}_{B_t} = \frac{1}{6} \begin{bmatrix} -t & 12-9t & 9t-12 & t \\ 3t & 12t-18 & 18-15t & 0 \\ -3t & 0 & 3t & 0 \\ t & 6-2t & t & 0 \end{bmatrix} \quad (8.59)$$

que coincide con  $M_B$  cuando  $t = 1$ .

Obtenemos las funciones de combinación de *splines* B periódicos y cúbicos en el rango de variación del parámetro desde 0 a 1 desarrollando la representación matricial hasta la forma polinómica. Por ejemplo, utilizando el valor de tensión  $t = 1$ , tenemos:

$$\begin{aligned} B_{0,3}(u) &= \frac{1}{6}(1-u)^3, & 0 \leq u \leq 1 \\ B_{1,3}(u) &= \frac{1}{6}(3u^3 - 6u^2 + 4) \\ B_{2,3}(u) &= \frac{1}{6}(-3u^3 + 3u^2 + 3u + 1) \\ B_{3,3}(u) &= \frac{1}{6}u^3 \end{aligned} \quad (8.60)$$

## Curvas con *splines* B abiertos y uniformes

Esta clase de *splines* B es una combinación de *splines* B uniformes y *splines* B no uniformes. A veces se trata como un tipo especial de *spline* B uniforme y otras se considera como un *spline* B no uniforme. En los *splines*

nes B **abiertos y uniformes**, o simplemente *splines B abiertos*, el espaciado de los nudos es uniforme excepto en los extremos, en los que los valores de los nudos se repiten  $d$  veces.

A continuación se muestran dos ejemplos de vectores de nudos abiertos, uniformes y enteros que comienzan por el valor 0.

$$\begin{aligned} \{0, 0, 1, 2, 3, 3\} &\quad \text{donde } d = 2 \text{ y } n = 3 \\ \{0, 0, 0, 0, 1, 2, 2, 2, 2\} &\quad \text{donde } d = 4 \text{ y } n = 4 \end{aligned} \tag{8.61}$$

Podemos normalizar estos vectores de nudos al intervalo que varía desde 0 a 1 del siguiente modo:

$$\begin{aligned} \{0, 0, 0.33, 0.67, 1, 1\} &\quad \text{donde } d = 2 \text{ y } n = 3 \\ \{0, 0, 0, 0, 0.5, 1, 1, 1, 1\} &\quad \text{donde } d = 4 \text{ y } n = 4 \end{aligned} \tag{8.62}$$

Para valores cualesquiera de los parámetros  $d$  y  $n$ , podemos generar un vector de nudos abierto y uniforme con valores enteros utilizando los cálculos:

$$u_j = \begin{cases} 0 & \text{para } 0 \leq j < d \\ j - d + 1 & \text{para } d \leq j \leq n \\ n - d + 2 & \text{para } j > n \end{cases} \tag{8.63}$$

para valores de  $j$  dentro del rango de variación de 0 a  $n + d$ . Con esta asignación, a los primeros  $d$  nudos se les asigna el valor 0 y los últimos  $d$  nudos tienen el valor  $n - d + 2$ .

Los *splines B* uniformes tienen características que son muy similares a las de los *splines de Bézier*. De hecho, cuando  $d = n + 1$  (el grado del polinomio es  $n$ ), los *splines B* son idénticos a los *splines de Bézier*, y todos los valores de los nudos son 0 o 1. Por ejemplo, para un *spline B* abierto y cúbico ( $d = 4$ ) y cuatro puntos de control, el vector de nudos es:

$$\{0, 0, 0, 0, 1, 1, 1, 1\}$$

La curva polinómica de un *spline B* abierto une los primeros con los últimos puntos de control. También, la pendiente paramétrica de la curva en el primer punto de control es paralela a la línea recta formada por los dos primeros puntos de control, y la pendiente paramétrica en el último punto de control es paralela a la línea definida por los dos últimos puntos de control. Por tanto, las restricciones geométricas para hacer coincidir las secciones de la curva son las mismas que las de las curvas de Bézier.

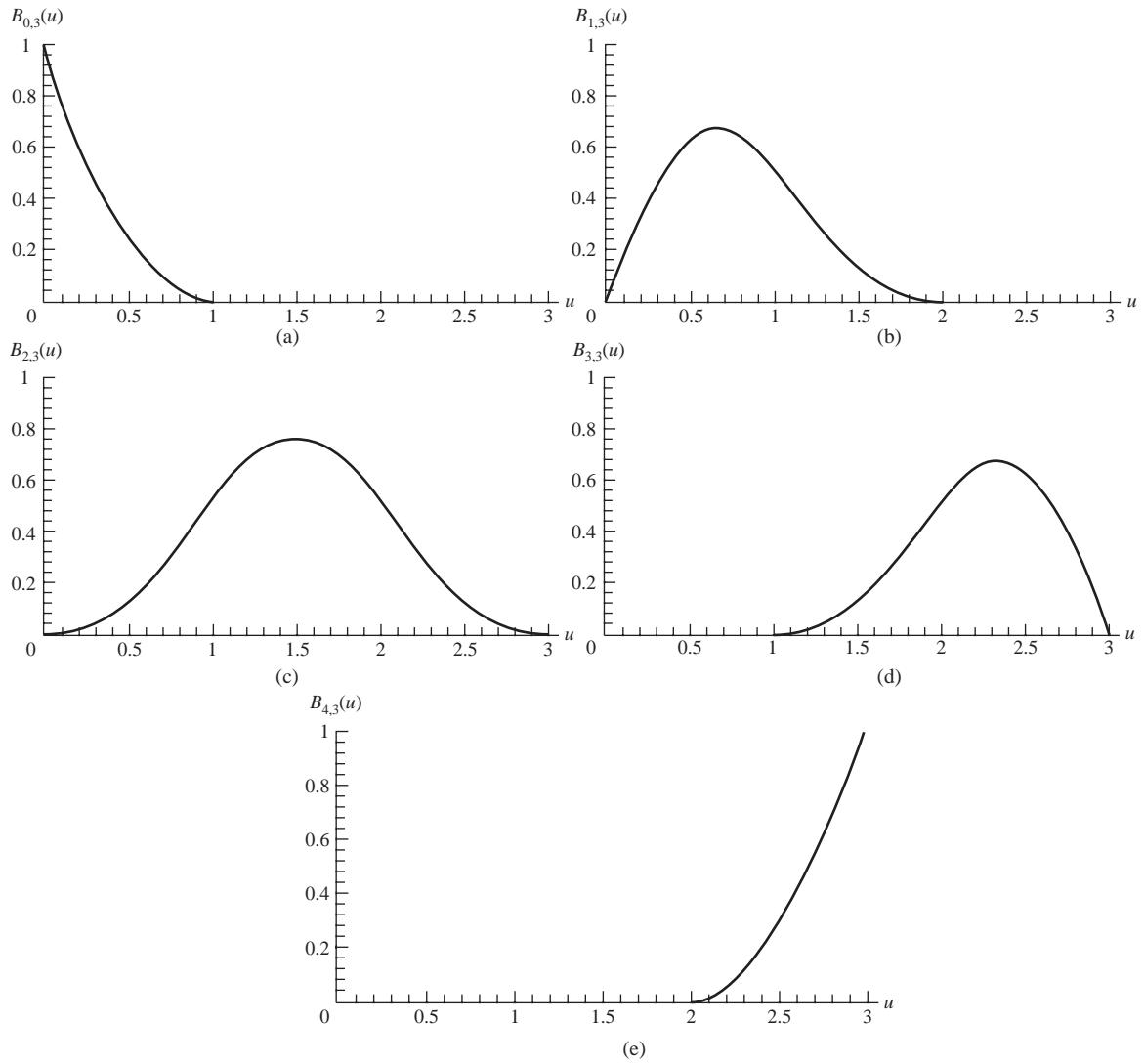
Como en las curvas de Bézier, especificar múltiples puntos de control en la misma posición en coordenadas desplaza cualquier curva con *splines B* más cerca de dicha posición. Ya que los *splines B* abiertos comienzan por el primer punto de control y terminan en el último punto de control, se puede generar una curva cerrada estableciendo el primer punto de control y el último en la misma posición de coordenadas.

### Ejemplo 8.2 Splines B abiertos, uniformes y cuadráticos

A partir de las condiciones 8.63 con  $d = 3$  y  $n = 4$  (cinco puntos de control), obtenemos los ocho valores siguientes del vector de nudos:

$$\{u_0, u_1, u_2, u_3, u_4, u_5, u_6, u_7\} = \{0, 0, 0, 1, 2, 3, 3, 3\}$$

El rango total de variación de  $u$  se divide en siete subintervalos, y cada una de las cinco funciones de combinación  $B_{k,3}$  están definidas sobre tres subintervalos, comenzando por el nudo  $u_k$ . Por tanto,  $B_{0,3}$  está definido desde  $u_0 = 0$  hasta  $u_3 = 1$ ,  $B_{1,3}$  está definido desde  $u_1 = 0$  hasta  $u_4 = 2$  y  $B_{4,3}$  está definido desde  $u_4 = 2$  hasta  $u_7 = 3$ . Las expresiones explícitas polinómicas de las funciones de combinación se obtienen a partir de las relaciones de recurrencia 8.53 y son las siguientes:



**FIGURA 8.45.** Funciones de combinación de *splines B* abiertos y uniformes con  $n = 4$  y  $d = 3$ .

$$B_{0,3}(u) = (1-u)^2 \quad 0 \leq u < 1$$

$$B_{1,3}(u) = \begin{cases} \frac{1}{2}u(4-3u) & 0 \leq u < 1 \\ \frac{1}{2}(2-u)^2 & 1 \leq u < 2 \end{cases}$$

$$B_{2,3}(u) = \begin{cases} \frac{1}{2}u^2 & 0 \leq u < 1 \\ \frac{1}{2}u(2-u) + \frac{1}{2}u(u-1)(3-u) & 1 \leq u < 2 \\ \frac{1}{2}(3-u)^2 & 2 \leq u < 3 \end{cases}$$

$$B_{3,3}(u) = \begin{cases} \frac{1}{2}(u-1)^2 & 1 \leq u < 2 \\ \frac{1}{2}(3-u)(3u-5) & 2 \leq u < 3 \end{cases}$$

$$B_{4,3}(u) = (u-2)^2 \quad 2 \leq u < 3$$

La Figura 8.45 muestra la forma de estas cinco funciones de combinación. Se observan de nuevo las características locales de los *splines* B. La función de fundido  $B_{0,3}$  es distinta de cero sólo en el subintervalo que varía de 0 a 1, de modo que el primer punto de control sólo influye en la curva en este intervalo. De forma similar, la función  $B_{4,3}$  es 0 fuera del intervalo que varía de 2 a 3, y la posición del último punto de control no afecta a la forma del comienzo ni a las partes medias de la curva.

Las formulaciones matriciales de los *splines* B abiertos no se generan tan cómodamente como las de los *splines* B periódicos y uniformes. Esto es debido a la multiplicidad de los valores de los nodos en el comienzo y final del vector de nudos.

### Curvas con *splines* B no uniformes

En esta clase de *splines*, podemos especificar cualesquiera valores e intervalos en el vector de nudos. En los *splines* B **no uniformes**, podemos elegir múltiples valores de nudos internos y espaciados desiguales entre los valores de los nudos. Algunos ejemplos son:

$$\begin{aligned} &\{0, 1, 2, 3, 3, 4\} \\ &\{0, 2, 2, 3, 3, 6\} \\ &\{0, 0, 0, 1, 1, 3, 3, 3\} \\ &\{0, 0.2, 0.6, 0.9, 1.0\} \end{aligned}$$

Los *splines* B no uniformes proporcionan una mayor flexibilidad en el control de la forma de la curva. Con los intervalos desigualmente espaciados en el vector de nudos, obtenemos formas diferentes en las funciones de combinación en intervalos diferentes, que se pueden utilizar para diseñar las características de los *splines*. Al incrementar la multiplicidad de los nudos, podemos producir variaciones sutiles en la trayectoria de la curva e introducir discontinuidades. Los valores múltiples de los nudos también reducen la continuidad en una unidad con cada repetición de valor concreto.

Obtenemos las funciones de combinación de un *spline* B no uniforme utilizando métodos similares a los estudiados para los *splines* B uniformes y abiertos. Dado un conjunto de  $n + 1$  puntos de control, establecemos el grado del polinomio y seleccionamos los valores de los nudos. A continuación, utilizando las relaciones de recurrencia, podríamos obtener el conjunto de funciones de combinación o evaluar directamente los puntos de la curva para su visualización. Los paquetes gráficos a menudo restringen los intervalos de los nudos a 0 o 1 para reducir los cálculos. Un conjunto de matrices características se puede almacenar entonces y utilizar para calcular los valores a lo largo de la curva con *splines* sin evaluar las relaciones de recurrencia en cada punto de la curva que hay que dibujar.

## 8.13 SUPERFICIES CON SPLINES B

La formulación de una superficie con *splines* B es similar a la de la superficie con *splines* de Bézier. Podemos obtener una función vectorial del punto sobre una superficie con *splines* B utilizando el producto cartesiano de las funciones de combinación de los *splines* B de este modo:



**FIGURA 8.46.** Un prototipo de helicóptero, diseñado y modelado por Daniel Langlois de SOFTIMAGE, Inc., Montreal, Quebec, Canadá, utilizando 180.000 parches de superficie con *splines B*. La escena se sombreó utilizando trazado de rayos, mapas de abultamientos y mapas de reflexión. (Cortesía de Silicon Graphics, Inc.)

$$\mathbf{P}(u, v) = \sum_{k_u=0}^{n_u} \sum_{k_v=0}^{n_v} \mathbf{p}_{k_u, k_v} B_{k_u, d_u}(u) B_{k_v, d_v}(v) \quad (8.64)$$

donde los valores del vector  $\mathbf{p}_{k_u, k_v}$  especifican las posiciones de los  $(n_u+1)$  por  $(n_v+1)$  puntos de control.

Las superficies con *splines B* exhiben las mismas propiedades que sus curvas componentes con *splines B*. Una superficie se puede construir a partir de valores seleccionados para los parámetros de grado  $d_u$  y  $d_v$ , que establecen los grados de los polinomios ortogonales de la superficie en  $d_u - 1$  y  $d_v - 1$ . Para cada parámetro  $u$  y  $v$  de la superficie, también podemos seleccionar valores para los vectores de nudos, que determinan el rango del parámetro en las funciones de combinación. La Figura 8.46 muestra un objeto modelado con superficies con *splines B*.

## 8.14 SPLINES BETA

Los *splines beta* son una generalización de los *splines B*, también denominados *splines β*, que se formulan imponiendo condiciones de continuidad geométrica en la primera y segunda derivadas paramétricas. Los parámetros de continuidad en los *splines beta* se denominan *parámetros beta*.

### Condiciones de continuidad de los *splines beta*

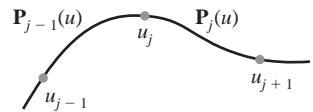
En un vector específico de nudos, nombramos las secciones del *spline* de izquierda a derecha de un nudo particular  $u_j$  con los vectores de posición  $\mathbf{P}_{j-1}(u)$  y  $\mathbf{P}_j(u)$  (Figura 8.47). La continuidad de orden cero (*continuidad posicional*),  $G^0$ , en  $u_j$  se obtiene requiriendo que:

$$\mathbf{P}_{j-1}(u_j) = \mathbf{P}_j(u_j) \quad (8.65)$$

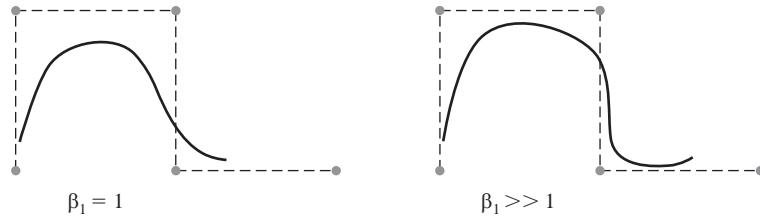
La continuidad de primer orden (*continuidad de tangente unitaria*),  $G^1$ , se obtiene requiriendo que los vectores tangente sean proporcionales:

$$\beta_1 \mathbf{P}'_{j-1}(u_j) = \mathbf{P}'_j(u_j), \quad \beta_1 > 0 \quad (8.66)$$

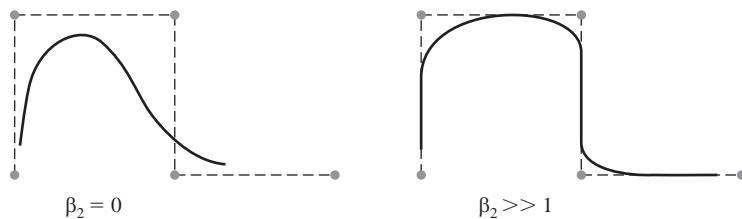
En este punto, las primeras derivadas paramétricas son proporcionales y los vectores unitarios tangentes son continuos a través del nudo.



**FIGURA 8.47.** Vectores de posición a lo largo de las secciones de la curva a la izquierda y a la derecha del nudo  $u_j$ .



**FIGURA 8.48.** Efecto del parámetro  $\beta_1$  sobre la forma de una curva con *splines* beta.



**FIGURA 8.49.** Efecto del parámetro  $\beta_2$  en la forma de una curva con *splines* beta.

La continuidad de segundo orden (*continuidad de vector de curvatura*),  $G^2$ , se impone con la condición:

$$\beta_1^2 \mathbf{P}_{j-1}''(u_j) + \beta_2 \mathbf{P}'_{j-1}(u_j) = \mathbf{P}_j''(u_j) \quad (8.67)$$

donde a  $\beta_2$  se le puede asignar cualquier número real y  $\beta_1 > 0$ . El vector de curvatura proporciona una medida de la cantidad que se dobla la curva en la posición  $u_j$ . Cuando  $\beta_1 = 1$  y  $\beta_2 = 0$ , los *splines* beta se convierten en *splines* B.

El parámetro  $\beta_1$  se denomina *parámetro de desplazamiento* ya que controla el desplazamiento de la curva. Para  $\beta_1 > 1$ , la curva tiende a alisarse a la derecha en la dirección del vector unitario tangente en los nudos. Para  $0 < \beta_1 < 1$ , la curva tiende a alisarse hacia la izquierda. El efecto de  $\beta_1$  sobre la forma de la curva con *splines* se muestra en la Figura 8.48.

El parámetro  $\beta_2$  se denomina *parámetro de tensión* ya que controla cómo de tenso o suelto se ajusta el *spline* al grafo de control. A medida que  $\beta_2$  crece, la curva se approxima a la forma del grafo de control, como se muestra en la Figura 8.49.

### Representación matricial de *splines* beta cúbicos y periódicos

Aplicando las condiciones en los límites de un *spline* beta a un polinomio cúbico con un vector uniforme de nudos, obtenemos la representación matricial de un *spline* beta periódico.

$$\mathbf{M}_\beta = \frac{1}{\delta} \begin{bmatrix} -2\beta_1^3 & 2(\beta_2 + \beta_1^3 + \beta_1^2 + \beta_1) & -2(\beta_2 + \beta_1^2 + \beta_1 + 1) & 2 \\ 6\beta_1^3 & -3(\beta_2 + 2\beta_1^3 + 2\beta_1^2) & 3(\beta_2 + 2\beta_1^2) & 0 \\ -6\beta_1^3 & 6(\beta_1^3 - \beta_1) & 6\beta_1 & 0 \\ 2\beta_1^3 & \beta_2 + 4(\beta_1^2 + \beta_1) & 2 & 0 \end{bmatrix} \quad (8.68)$$

donde  $\delta = \beta_2 + 2\beta_1^3 + 4\beta_1^2 + 4\beta_1 + 2$ .

Obtenemos la matriz  $M_B$  del *spline* B cuando  $\beta_1 = 1$  y  $\beta_2 = 0$ . Y tenemos la matriz  $M_{Bt}$  de tensión del *spline* B (Ecuación 8.59) cuando,

$$\beta_1 = 1, \quad \beta_2 = \frac{12}{t}(1-t)$$

## 8.15 SPLINES RACIONALES

---

Una función racional es simplemente el cociente de dos polinomios. Por tanto, un ***spline* racional** es el cociente de dos funciones de *splines*. Por ejemplo, una curva con *splines* B racionales se puede describir con el vector de posición:

$$\mathbf{P}(u) = \frac{\sum_{k=0}^n \omega_k \mathbf{p}_k B_{k,d}(u)}{\sum_{k=0}^n \omega_k B_{k,d}(u)} \quad (8.69)$$

donde  $\mathbf{p}_k$  define un conjunto de  $n + 1$  puntos de control. Los parámetros  $\omega_k$  son los factores de ponderación de los puntos de control. Cuanto mayor es valor de un  $\omega_k$  concreto, tanto más es atraída la curva hacia el punto de control  $\mathbf{p}_k$  ponderado por aquel parámetro. Cuando todos los factores de ponderación tienen el valor 1, tenemos la curva estándar con *splines* B, ya que el denominador de la Ecuación 8.69 es entonces simplemente la suma de las funciones de combinación, la cual toma el valor 1 (Ecuación 8.54).

Los *splines* racionales presentan dos ventajas importantes comparados con los *splines* no racionales. En primer lugar, proporcionan una representación exacta de curvas cuadráticas (cónicas), tales como los círculos y las elipses. Los *splines* no racionales, que son polinomios, sólo representan de forma aproximada las cónicas. Esto permite a los paquetes gráficos modelar todas las formas de las curvas con una representación, *splines* racionales, sin necesidad de una biblioteca de funciones de curvas para manejar formas de diseño diferentes. La segunda ventaja de los *splines* racionales es que son invariantes frente a las transformaciones de visualización (Sección 7.8). Esto significa que podemos aplicar una transformación de visualización a los puntos de control de una curva racional, y obtendremos la vista correcta de la curva. Los *splines* no racionales, en cambio, no son invariantes frente a una transformación de visualización de la perspectiva. Habitualmente, los paquetes de diseño de gráficos utilizan representaciones con vectores no uniformes de nudos para construir *splines* B racionales. Estos *splines* se denominan NURBS (nonuniform rational B-splines; *splines* B racionales no uniformes), o NURBs.

Las representaciones con coordenadas homogéneas se utilizan en *splines* racionales, ya que el denominador se puede tratar como el factor homogéneo  $h$  en una representación de cuatro dimensiones de los puntos de control. Por tanto, un *spline* racional se puede considerar como la proyección de un *spline* no racional de cuatro dimensiones sobre un espacio tridimensional.

Por lo general, la construcción de una representación con un *spline* B racional se realiza utilizando los mismos procedimientos que empleamos para obtener una representación no racional. Dado el conjunto de puntos de control, el grado del polinomio, los factores de ponderación y el vector de nudos, aplicamos las relaciones de redundancia para obtener las funciones de combinación. En algunos sistemas CAD, construimos una sección de una cónica especificando tres puntos de un arco. Una representación de un *spline* racional en coordenadas homogéneas se determina a continuación calculando las posiciones de los puntos de control que generan el tipo de cónica seleccionado.

Como ejemplo de la descripción de secciones de cónicas con *splines* racionales, podemos utilizar una función de *splines* B cuadráticos ( $d = 3$ ), tres puntos de control y un vector abierto de nudos,

$$\{0, 0, 0, 1, 1, 1\}$$

que es el mismo que el del *spline* cuadrático de Bézier. Despues establecemos las funciones de ponderación en los valores:

$$\begin{aligned}\omega_0 &= \omega_2 = 1 \\ \omega_1 &= \frac{r}{1-r}, \quad 0 \leq r < 1\end{aligned}\tag{8.70}$$

y la representación del *spline* B racional es:

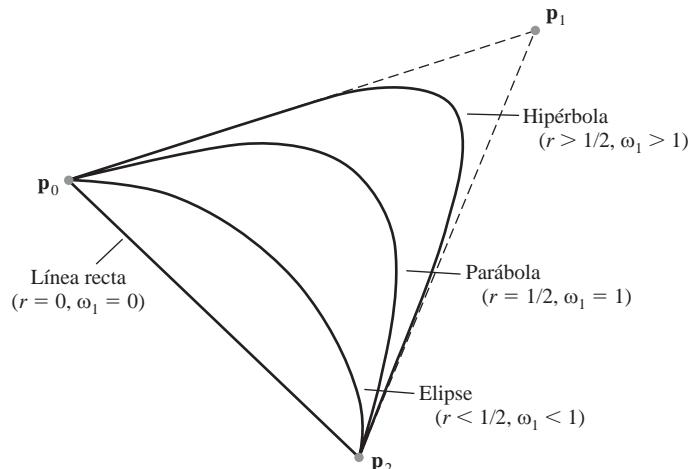
$$\mathbf{P}(u) = \frac{\mathbf{p}_0 B_{0,3}(u) + [r/(1-r)]\mathbf{p}_1 B_{1,3}(u) + \mathbf{p}_2 B_{2,3}(u)}{B_{0,3}(u) + [r/(1-r)]B_{1,3}(u) + B_{2,3}(u)}\tag{8.71}$$

Posteriormente, obtenemos las distintas cónicas (Figura 8.50) con los siguientes valores del parámetro  $r$ .

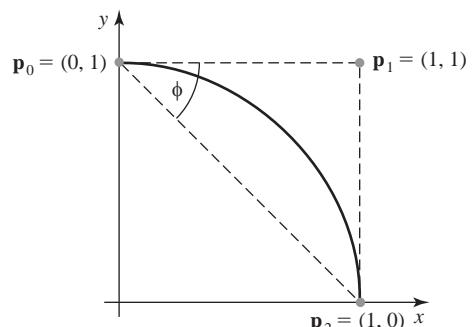
$r > 1/2$ ,	$\omega_1 > 1$	sección de hipérbola
$r = 1/2$ ,	$\omega_1 = 1$	sección de parábola
$r < 1/2$ ,	$\omega_1 < 1$	sección de elipse
$r = 0$ ,	$\omega_1 = 0$	segmento de línea recta

Podemos generar un arco formado por un cuarto de un círculo unidad del primer cuadrante del plano  $xy$  (Figura 8.51) estableciendo el valor de  $\omega_1$  en  $\cos \phi$  y eligiendo los siguientes puntos de control:

$$\mathbf{p}_0 = (0, 1), \quad \mathbf{p}_1 = (1, 1), \quad \mathbf{p}_2 = (1, 0)$$



**FIGURA 8.50.** Secciones de cónicas generadas utilizando varios valores del factor de ponderación de *splines* racionales  $\omega_1$ .



**FIGURA 8.51.** Un arco circular en el primer cuadrante del plano  $xy$ .

Se puede obtener un círculo generando secciones en los otros tres cuadrantes utilizando puntos de control similares. O podríamos producir un círculo completo a partir de la sección del primer cuadrante utilizando transformaciones geométricas en el plano  $xy$ . Por ejemplo, podemos reflejar el arco circular de un cuadrante según los ejes  $x$  e  $y$  para obtener los arcos circulares de los otros tres cuadrantes.

Ésta es una representación homogénea de un arco circular unidad del primer cuadrante del plano  $xy$ .

$$\begin{bmatrix} x_h(u) \\ y_h(u) \\ z_h(u) \\ h(u) \end{bmatrix} = \begin{bmatrix} 1-u^2 \\ 2u \\ 0 \\ 1+u^2 \end{bmatrix} \quad (8.72)$$

Esta representación homogénea conduce a las ecuaciones paramétricas del círculo en el primer cuadrante.

$$\begin{aligned} x &= \frac{x_h(u)}{h(u)} = \frac{1-u^2}{1+u^2} \\ y &= \frac{y_h(u)}{h(u)} = \frac{2u}{1+u^2} \end{aligned} \quad (8.73)$$

## 8.16 CONVERSIÓN ENTRE REPRESENTACIONES DE SPLINES

---

A veces es conveniente ser capaz de pasar de una representación de un *spline* a otra. Por ejemplo, una representación de Bézier es más conveniente para subdividir una curva con *splines*, mientras que una representación con *splines B* ofrece una mayor flexibilidad en el diseño. Por lo que podríamos diseñar una curva utilizando secciones de *splines B*, después realizar una conversión a una representación equivalente de Bézier para visualizar el objeto utilizando un procedimiento de subdivisión recursivo para posicionar los puntos a lo largo de la curva.

Supóngase que tenemos una descripción mediante *splines* de un objeto que se puede expresar con el siguiente producto matricial:

$$\mathbf{P}(u) = \mathbf{U} \cdot \mathbf{M}_{\text{spline1}} \cdot \mathbf{M}_{\text{geom1}} \quad (8.74)$$

donde  $\mathbf{M}_{\text{spline1}}$  es la matriz que caracteriza la representación con *splines* y  $\mathbf{M}_{\text{geom1}}$  es la matriz columna con las restricciones geométricas (por ejemplo, las coordenadas de los puntos de control). Para transformarla a una segunda representación con la matriz del *spline*  $\mathbf{M}_{\text{spline2}}$ , debemos determinar la matriz de restricciones geométricas  $\mathbf{M}_{\text{geom2}}$  que produce la misma función vectorial de punto para el objeto. Es decir,

$$\mathbf{P}(u) = \mathbf{U} \cdot \mathbf{M}_{\text{spline2}} \cdot \mathbf{M}_{\text{geom2}} \quad (8.75)$$

o

$$\mathbf{U} \cdot \mathbf{M}_{\text{spline2}} \cdot \mathbf{M}_{\text{geom2}} = \mathbf{U} \cdot \mathbf{M}_{\text{spline1}} \cdot \mathbf{M}_{\text{geom1}} \quad (8.76)$$

Resolviendo en  $\mathbf{M}_{\text{geom2}}$ , tenemos:

$$\begin{aligned} \mathbf{M}_{\text{geom2}} &= \mathbf{M}_{\text{spline2}}^{-1} \cdot \mathbf{M}_{\text{spline1}} \cdot \mathbf{M}_{\text{geom1}} \\ &= \mathbf{M}_{s1,s2} \cdot \mathbf{M}_{\text{geom1}} \end{aligned} \quad (8.77)$$

Por tanto, la matriz requerida de transformación que convierte la primera representación con *splines* a la segunda es:

$$\mathbf{M}_{s1,s2} = \mathbf{M}_{\text{spline2}}^{-1} \cdot \mathbf{M}_{\text{spline1}} \quad (8.78)$$

Un *spline* B no uniforme no se puede caracterizar con una matriz general de *splines*. Pero podemos reorganizar la secuencia de nudos para convertir el *spline* B no uniforme en una representación de Bézier. A continuación la matriz de Bézier se puede convertir a cualquier otra forma.

El siguiente ejemplo calcula la matriz de transformación para convertir una representación con *spline* B periódico y cúbico a una representación con *spline* de Bézier cúbico.

$$\begin{aligned} \mathbf{M}_{B,\text{Bez}} &= \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}^{-1} \cdot \frac{1}{6} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 4 & 1 & 0 \\ 0 & 4 & 2 & 0 \\ 0 & 2 & 4 & 0 \\ 0 & 1 & 4 & 1 \end{bmatrix} \end{aligned} \quad (8.79)$$

Y la matriz de transformación para convertir desde una representación de Bézier cúbica a una representación con *spline* B periódico y cúbico es:

$$\begin{aligned} \mathbf{M}_{\text{Bez},B} &= \begin{bmatrix} -\frac{1}{6} & \frac{1}{2} & -\frac{1}{2} & \frac{1}{6} \\ \frac{1}{2} & -1 & \frac{1}{2} & 0 \\ -\frac{1}{2} & 0 & \frac{1}{2} & 0 \\ \frac{1}{6} & \frac{2}{3} & \frac{1}{6} & 0 \end{bmatrix}^{-1} \cdot \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \\ &= \begin{bmatrix} 6 & -7 & 2 & 0 \\ 0 & 2 & -1 & 0 \\ 0 & -1 & 2 & 0 \\ 0 & 2 & -7 & 6 \end{bmatrix} \end{aligned} \quad (8.80)$$

## 8.17 VISUALIZACIÓN DE CURVAS Y SUPERFICIES CON SPLINES

Para visualizar una curva o una superficie con *splines*, debemos determinar las coordenadas de la curva o de la superficie que se proyectan en los píxeles del dispositivo de visualización. Esto significa que debemos evaluar las funciones polinómicas paramétricas del *spline* en ciertos incrementos sobre el rango de las funciones. Se han desarrollado varios métodos para realizar esta evaluación de forma eficiente.

### Regla de Horner

El método más simple para evaluar un polinomio, aparte del cálculo directo de cada término de forma sucesiva, es la *regla de Horner*, que realiza los cálculos mediante una factorización sucesiva. Esto requiere una multiplicación y una suma en cada paso. Para un polinomio de grado  $n$ , hay  $n$  pasos.

A modo de ejemplo, suponga que tenemos una representación con *spline* cúbico donde la coordenada  $x$  se expresa como:

$$x(u) = a_x u^3 + b_x u^2 + c_x u + d_x \quad (8.81)$$

Las expresiones para las coordenadas  $y$  y  $z$  son similares. Para un valor concreto del parámetro  $u$ , evaluamos este polinomio en el siguiente orden factorizado.

$$x(u) = [(a_x u + b_x) u + c_x] u + d_x \quad (8.82)$$

El cálculo de cada valor de  $x$  requiere tres multiplicaciones y tres sumas, de modo que la determinación de cada posición en coordenadas  $(x, y, z)$  sobre la curva con *spline* cúbico requiere nueve multiplicaciones y nueve sumas.

Se podrían aplicar manipulaciones adicionales de factorización para reducir el número de cálculos requeridos por el método de Horner, especialmente en el caso de polinomios de mayor orden (grado mayor que 3). Pero la determinación repetida de las posiciones en coordenadas sobre el rango de una función de *spline*, se puede calcular mucho más rápidamente utilizando cálculos de diferencias hacia delante o métodos de subdivisión de *splines*.

## Cálculos de diferencias hacia adelante

Un método rápido para evaluar funciones polinómicas consiste en generar valores sucesivos de forma recursiva incrementando los valores previamente calculados, por ejemplo, de este modo:

$$x_{k+1} = x_k + \Delta x_k \quad (8.83)$$

Por tanto, una vez que conocemos el incremento y el valor de  $x_k$  de un paso cualquiera, obtenemos el valor siguiente simplemente añadiendo el incremento a  $x_k$ . El incremento  $\Delta x_k$  en cada paso se denomina *diferencia hacia adelante*. En el caso de la representación paramétrica de la curva, obtenemos las diferencias hacia adelante a partir de los intervalos que seleccionamos para el parámetro  $u$ . Si dividimos el rango total de variación de  $u$  en subintervalos de tamaño fijo  $\delta$ , entonces dos valores sucesivos de  $x$  son  $x_k = x(u_k)$  y  $x_{k+1} = x(u_{k+1})$ , donde:

$$u_{k+1} = u_k + \delta, \quad k = 0, 1, 2, \dots \quad (8.84)$$

y  $u_0 = 0$ .

Para ilustrar este método, consideraremos en primer lugar la representación polinómica  $x(u) = a_x u + b_x$  para las coordenadas a lo largo de una curva con *spline* lineal. Dos valores sucesivos de la coordenada  $x$  se representan como:

$$\begin{aligned} x_k &= a_x u_k + b_x \\ x_{k+1} &= a_x(u_k + \delta) + b_x \end{aligned} \quad (8.85)$$

Restando las dos ecuaciones, obtenemos la siguiente diferencia hacia adelante:

$$\Delta x_k = x_{k+1} - x_k = a_x \delta \quad (8.86)$$

En este caso, la diferencia hacia adelante es una constante. En el caso de polinomios de orden más elevado, la diferencia hacia adelante es ella misma una función polinómica del parámetro  $u$ . Esta diferencia hacia adelante polinómica tiene un grado menos que el polinomio original.

En la representación con *splines* cúbicos de la Ecuación 8.81, dos valores sucesivos de la coordenada  $x$  tienen las representaciones polinómicas:

$$\begin{aligned} x_k &= a_x u_k^3 + b_x u_k^2 + c_x u_k + d_x \\ x_{k+1} &= a_x(u_k + \delta)^3 + b_x(u_k + \delta)^2 + c_x(u_k + \delta) + d_x \end{aligned} \quad (8.87)$$

El resultado de la evaluación de la diferencia hacia adelante ahora es:

$$\Delta x_k = 3a_x \delta u_k^2 + (3a_x \delta^2 + 2b_x \delta) u_k + (a_x \delta^3 + b_x \delta^2 + c_x \delta) \quad (8.88)$$

que es una función cuadrática del parámetro  $u_k$ . Ya que  $\Delta x_k$  es una función polinómica de  $u$ , podemos utilizar el mismo procedimiento incremental para obtener valores sucesivos de  $\Delta x_k$ . Es decir,

$$\Delta x_{k+1} = \Delta x_k + \Delta_2 x_k \quad (8.89)$$

donde la segunda diferencia hacia adelante es la función lineal:

$$\Delta_2 x_k = 6a_x \delta^2 u_k + 6a_x \delta^3 + 2b_x \delta^2 \quad (8.90)$$

Repitiendo este proceso una vez más, podemos escribir,

$$\Delta_2 x_{k+1} = \Delta_2 x_k + \Delta_3 x_k \quad (8.91)$$

en la que la tercera diferencia hacia adelante es la expresión constante:

$$\Delta_3 x_k = 6a_x \delta^3 \quad (8.92)$$

Las Ecuaciones 8.83, 8.89, 8.91 y 8.92 proporcionan un cálculo incremental de las diferencias hacia adelante de los puntos a lo largo de la curva cúbica. Comenzando por  $u_0 = 0$  con un paso constante de tamaño  $\delta$ , los valores iniciales de la coordenada  $x$  y sus primeras dos diferencias hacia adelante son:

$$\begin{aligned} x_0 &= d_x \\ \Delta x_0 &= a_x \delta^3 + b_x \delta^2 + c_x \delta \\ \Delta_2 x_0 &= 6a_x \delta^3 + 2b_x \delta^2 \end{aligned} \quad (8.93)$$

Una vez que se han calculado estos valores iniciales, el cálculo de cada valor sucesivo de la coordenada  $x$  requiere sólo tres sumas.

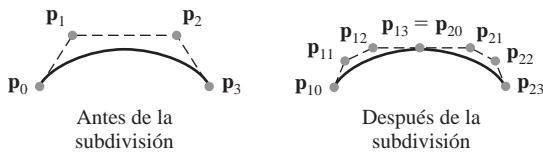
Podemos aplicar los métodos de las diferencias hacia adelante para determinar los puntos a lo largo de curvas con *splines* de cualquier grado  $n$ . Cada punto sucesivo  $(x, y, z)$  se evalúa con una serie de  $3n$  sumas. Para las superficies, los cálculos incrementales se aplican tanto al parámetro  $u$  como al parámetro  $v$ .

## Métodos de subdivisión

Los procedimientos *recursivos de subdivisión de splines* se utilizan para dividir repetidamente una sección de curva por la mitad, incrementando el número de puntos de control en cada paso. Los métodos de subdivisión son útiles para visualizar curvas de aproximación con *splines*, ya que podemos continuar con el proceso de subdivisión hasta que el grafo de control se aproxime a la trayectoria de la curva. Las coordenadas de los puntos de control se pueden entonces dibujar como puntos de la curva. Otra aplicación de la subdivisión es generar más puntos de control para perfilar una curva. Por tanto, podríamos diseñar una forma general de una curva con unos pocos puntos de control, para a continuación aplicar un procedimiento de subdivisión para obtener puntos de control adicionales. Con los puntos de control añadidos, podemos entonces realizar un ajuste fino en secciones pequeñas de la curva.

La subdivisión de *splines* se aplica más fácilmente a una sección de curva de Bézier, porque la curva comienza en el primer punto de control y termina en el último punto de control, el rango de variación del parámetro  $u$  es siempre de 0 a 1 y es fácil determinar cuándo los puntos de control están «suficientemente cerca» de la trayectoria de la curva. La subdivisión de Bézier se puede aplicar a otras representaciones con *splines* mediante la siguiente secuencia de operaciones.

- (1) Conversión de la representación actual con *splines* en una representación de Bézier.
- (2) Aplicación del algoritmo de subdivisión de Bézier.
- (3) Conversión de la representación de Bézier en la representación original con *splines*.



**FIGURA 8.52.** Subdivisión de una sección de curva de Bézier cúbica en dos segmentos, cada uno con cuatro puntos de control.

La Figura 8.52 muestra el primer paso de una subdivisión recursiva de una sección de curva de Bézier cúbica. Los puntos a lo largo de la curva de Bézier se describen con la función de punto paramétrica  $\mathbf{P}(u)$  con  $0 \leq u \leq 1$ . En el primer paso de la subdivisión, utilizamos el punto medio  $\mathbf{P}(0.5)$  para dividir la curva original en dos segmentos. El primer segmento se describe a continuación con la función de punto  $\mathbf{P}_1(s)$  y el segundo segmento se describe con  $\mathbf{P}_2(t)$ , donde

$$\begin{aligned} s &= 2u, && \text{para } 0.0 \leq u \leq 0.5 \\ t &= 2u - 1, && \text{para } 0.5 \leq u \leq 1.0 \end{aligned} \quad (8.94)$$

Cada uno de los dos segmentos de curva tiene el mismo número de puntos de control que la curva original. También, las condiciones en los límites (posición y pendiente paramétrica) de los extremos de cada uno de los dos segmentos de curva deben coincidir con la posición y la pendiente de la función de la curva original  $\mathbf{P}(u)$ . Esto nos proporciona cuatro condiciones para cada segmento de curva que podemos utilizar para determinar la posición de los puntos de control. Para el primer segmento, los cuatro puntos de control son:

$$\begin{aligned} \mathbf{p}_{1,0} &= \mathbf{p}_0 \\ \mathbf{p}_{1,1} &= \frac{1}{2}(\mathbf{p}_0 + \mathbf{p}_1) \\ \mathbf{p}_{1,2} &= \frac{1}{4}(\mathbf{p}_0 + 2\mathbf{p}_1 + \mathbf{p}_2) \\ \mathbf{p}_{1,3} &= \frac{1}{8}(\mathbf{p}_0 + 3\mathbf{p}_1 + 3\mathbf{p}_2 + \mathbf{p}_3) \end{aligned} \quad (8.95)$$

Y para el segundo segmento de curva, obtenemos los cuatro puntos de control:

$$\begin{aligned} \mathbf{p}_{2,0} &= \frac{1}{8}(\mathbf{p}_0 + 3\mathbf{p}_1 + 3\mathbf{p}_2 + \mathbf{p}_3) \\ \mathbf{p}_{2,1} &= \frac{1}{4}(\mathbf{p}_1 + 2\mathbf{p}_2 + \mathbf{p}_3) \\ \mathbf{p}_{2,2} &= \frac{1}{2}(\mathbf{p}_2 + \mathbf{p}_3) \\ \mathbf{p}_{2,3} &= \mathbf{p}_3 \end{aligned} \quad (8.96)$$

Se puede establecer un orden eficiente para el cálculo del nuevo conjunto de puntos de control, utilizando sólo operaciones de suma y desplazamiento (división por 2) del siguiente modo:

$$\begin{aligned} \mathbf{p}_{1,0} &= \mathbf{p}_0 \\ \mathbf{p}_{1,1} &= \frac{1}{2}(\mathbf{p}_0 + \mathbf{p}_1) \\ \mathbf{T} &= \frac{1}{1}(\mathbf{p}_1 + \mathbf{p}_2) \end{aligned}$$

$$\begin{aligned}
 \mathbf{p}_{1,2} &= \frac{1}{2}(\mathbf{p}_{1,1} + \mathbf{T}) \\
 \mathbf{p}_{2,3} &= \mathbf{p}_3 \\
 \mathbf{p}_{2,2} &= \frac{1}{2}(\mathbf{p}_2 + \mathbf{p}_3) \\
 \mathbf{p}_{2,1} &= \frac{1}{2}(\mathbf{T} + \mathbf{p}_{2,2}) \\
 \mathbf{p}_{2,0} &= \frac{1}{2}(\mathbf{p}_{1,2} + \mathbf{p}_{2,1}) \\
 \mathbf{p}_{1,3} &= \mathbf{p}_{2,0}
 \end{aligned} \tag{8.97}$$

Los pasos anteriores se pueden repetir cualquier número de veces, dependiendo de si subdividimos la curva para obtener más puntos de control o intentamos localizar puntos aproximados de la curva. Cuando subdividimos para obtener un conjunto de puntos de visualización, podemos terminar el procedimiento de subdivisión cuando los segmentos de la curva son suficientemente pequeños. Un modo de determinar esto consiste en comprobar la distancia desde el primero al último punto de control de cada segmento. Si esta distancia es «suficientemente» pequeña, podemos detener la subdivisión. Otra prueba consiste en comprobar las distancias entre pares adyacentes de puntos de control. O podríamos detener la subdivisión cuando el conjunto de puntos de control de cada segmento esté próximo a la trayectoria de una línea recta.

Los métodos de subdivisión se pueden aplicar a curvas de Bézier de cualquier grado. En el caso de un polinomio de Bézier de grado  $n - 1$ , los  $2n$  puntos de control de cada uno de los dos segmentos iniciales de curva son:

$$\begin{aligned}
 \mathbf{p}_{1,k} &= \frac{1}{2^k} \sum_{j=0}^k C(k, j) \mathbf{p}_j, & k = 0, 1, 2, \dots, n \\
 \mathbf{p}_{2,k} &= \frac{1}{2^{n-k}} \sum_{j=k}^n C(n-k, n-j) \mathbf{p}_j
 \end{aligned} \tag{8.98}$$

donde  $C(k, j)$  y  $C(n - k, n - j)$  son los coeficientes binomiales.

Los métodos de subdivisión se pueden aplicar directamente a *splines* B no uniformes añadiendo valores al vector de nudos. Pero, por lo general, estos métodos no son tan eficientes como la subdivisión de *splines* de Bézier.

## 8.18 FUNCIONES OpenGL DE APROXIMACIÓN CON SPLINES

---

Tanto los *splines* de Bézier como los *splines* B se pueden visualizar utilizando funciones de OpenGL, así como curvas de recorte de superficies con *splines*. La biblioteca de núcleo contiene funciones de Bézier y la biblioteca GLU (OpenGL Utility) contiene las funciones para *splines* B y curvas de recorte. A menudo, las funciones de Bézier se implementan en el hardware, y las funciones de GLU proporcionan una interfaz que accede a las funciones de dibujo de puntos y de dibujo de líneas de OpenGL.

### Funciones OpenGL para curvas con *splines* de Bézier

Especificamos los parámetros y activamos las subrutinas para la visualización de curvas de Bézier con las siguientes funciones OpenGL:

```
glMap1* (GL_MAP1_VERTEX_3, uMin, uMax, stride, nPts, *ctrlPts);
 glEnable (GL_MAP1_VERTEX_3);
```

Y desactivamos las subrutinas con:

```
glDisable (GL_MAP1_VERTEX_3);
```

Se emplea el sufijo `f` o `d` en `glMap1` para indicar que los valores de los datos se especifican en punto flotante o doble precisión. Los valores mínimo y máximo del parámetro de la curva  $u$  se especifican en los parámetros `uMin` y `uMax`, aunque estos valores se establecen para una curva de Bézier habitualmente en 0 y 1.0, respectivamente. Los valores de las coordenadas cartesianas tridimensionales en punto flotante de los puntos de control de Bézier se enumeran en la matriz `ctrlPts`, y el número de elementos de esta matriz se proporciona como un entero positivo utilizando el parámetro `nPts`. Al parámetro `stride` se le asigna un entero de incremento (offset) que indica el número de datos entre el comienzo de una posición en coordenadas en la matriz `ctrlPts` y el comienzo de la siguiente posición de coordenadas. Para una lista de puntos de control tridimensionales, planteamos que `stride = 3`. Un valor mayor en `stride` se utilizaría si especificásemos los puntos de control utilizando coordenadas homogéneas de cuatro dimensiones o entrelazásemos los valores de las coordenadas con otros datos, tales como el color. Para expresar las posiciones de los puntos de control en coordenadas homogéneas de cuatro dimensiones ( $x, y, z, h$ ), sólo necesitamos cambiar el valor de `stride` y de la constante simbólica en `glMap1` y en `glEnable` a `GL_MAP1_VERTEX_4`.

Después de que hayamos establecido los parámetros de Bézier y activado las subrutinas de generación de curvas, necesitamos evaluar las posiciones a lo largo de la trayectoria del *spline* y visualizar la curva resultante. Una posición de coordenadas a lo largo de la trayectoria de la curva se calcula con:

```
glEvalCoord1* (uValue);
```

donde al parámetro `uValue` se le asigna algún valor dentro del intervalo que varía desde `uMin` a `uMax`. El código de sufijo para esta función puede ser `f` o `d`, y podemos también utilizar el código de sufijo `v` para indicar que el valor del argumento se proporciona en forma de matriz. La función `glEvalCoord1` calcula una posición de coordenadas utilizando la Ecuación 8.37 con el valor del parámetro:

$$u = \frac{u_{\text{value}} - u_{\text{min}}}{u_{\text{max}} - u_{\text{min}}} \quad (8.99)$$

que mapea el valor `uValue` al intervalo que varía 0 a 1.0.

Cuando `glEvalCoord1` procesa un valor del parámetro de la curva  $u$ , genera una función `glVertex3`. Para obtener una curva de Bézier, invocamos por tanto repetidamente la función `glEvalCoord1` para producir un conjunto de puntos a lo largo de la trayectoria de la curva, utilizando valores seleccionados del rango de variación desde `uMin` a `uMax`. Uniendo estos puntos con segmentos de línea recta, podemos aproximar la curva con *splines* mediante una polilínea.

Como un ejemplo de subrutinas para creación de curvas de Bézier con OpenGL, el código siguiente utiliza los cuatro puntos de control del programa de la Sección 8.10 para generar una curva de Bézier cónica bidimensional. En este ejemplo, se dibujan 50 puntos a lo largo de la trayectoria de la curva, y se conectan los puntos de la curva mediante segmentos de línea recta. La trayectoria de la curva se visualiza a continuación como una polilínea azul, y los puntos de control se dibujan como puntos rojos de tamaño 5 (Figura 8.53 en escala de grises).

---

```
GLfloat ctrlPts [4][3] = { {-40.0, 40.0, 0.0}, {-10.0, 200.0, 0.0},
                           {10.0, -200.0, 0.0}, {40.0, 40.0, 0.0} };

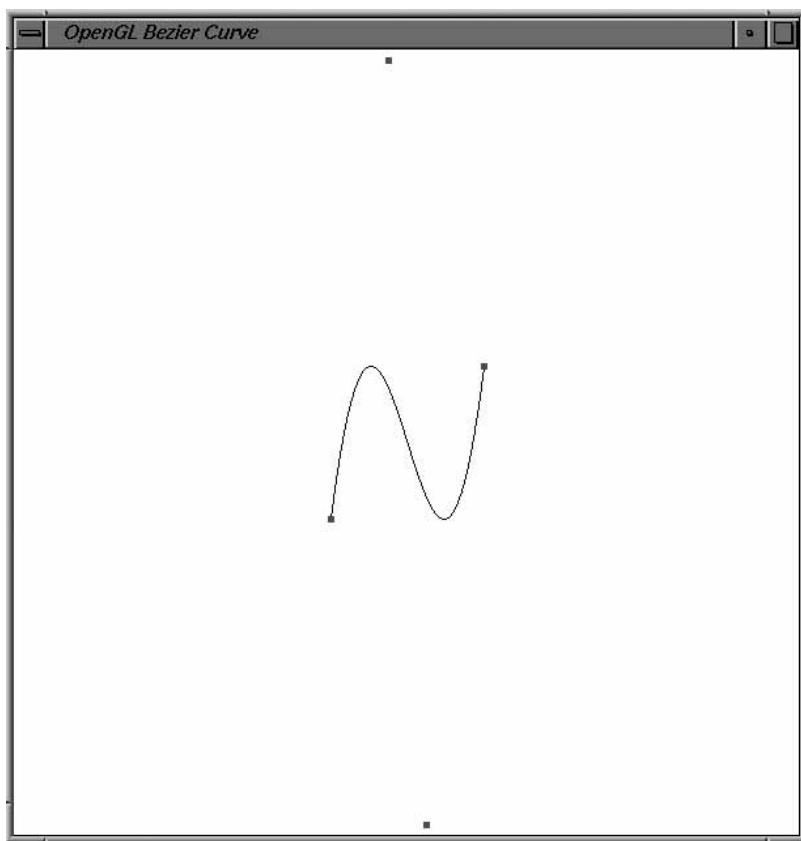
glMap1f (GL_MAP1_VERTEX_3, 0.0, 1.0, 3, 4, *ctrlPts);
 glEnable (GL_MAP1_VERTEX_3);
```

```

GLint k;
glColor3f (0.0, 0.0, 1.0);           // Establece el color de línea en azul.
glBegin (GL_LINE_STRIP);           // Genera la "curva" de Bézier.
    for (k = 0; k <= 50; k++)
        glEvalCoord1f (GLfloat (k) / 50.0);
glEnd ( );

glColor (1.0, 0.0, 0.0);           // Establece el color de los puntos en
// rojo.
glPointSize (5.0);                // Establece el tamaño del punto en 5.0.
glBegin (GL_POINTS);              // Dibuja los puntos de control.
    for (k = 0; k < 4; k++)
        glVertex3fv (&ctrlPts [k] [0]);
glEnd ( );

```



**FIGURA 8.53.** Un conjunto de cuatro puntos de control y la curva de Bézier asociada, visualizados con las subrutinas de OpenGL como una polilínea de aproximación.

Aunque el ejemplo anterior generó una curva con *splines* con valores del parámetro espaciados uniformemente, podemos utilizar la función `glEvalCoord1f` para obtener cualquier espaciado del parámetro  $u$ .

Habitualmente, sin embargo, una curva con *splines* se genera con valores del parámetro uniformemente espaciados, y OpenGL proporciona las siguientes funciones que podemos utilizar para producir un conjunto de valores del parámetro uniformemente espaciados.

```
glMapGrid1* (n, u1, u2);
glEvalMesh1 (mode, n1, n2);
```

El código de sufijo de `glMapGrid1` puede ser `f` o `d`. El parámetro `n` especifica el número entero de subdivisiones iguales sobre el rango de variación de `u1` a `u2`, y los parámetros `n1` y `n2` especifican un rango entero correspondiente a `u1` y `u2`. Al parámetro `mode` se le asigna `GL_POINT` o `GL_LINE`, dependiendo de si queremos visualizar la curva utilizando puntos discretos (una curva de puntos) o utilizando segmentos de línea recta. Para una curva que hay que visualizar como una polilínea, la salida de estas dos funciones es la misma que la salida del siguiente código, excepto que el argumento de `glEvalCoord1` se establece en `u1` o `u2` si `k = 0` o si `k = n`, respectivamente, para evitar errores de redondeo. En otras palabras, con `mode = GL_LINE`, los comandos anteriores de OpenGL son equivalentes a:

```
glBegin (GL_LINE_STRIP);
for (k = n1; k <= n2; k++)
    glEvalCoord1f (u1 + k * (u2 - u1) / n);
glEnd ();
```

Por tanto, en el ejemplo de programación anterior, podríamos reemplazar el bloque de código que contiene el bucle de generación de la curva de Bézier por las siguientes líneas.

```
	glColor3f (0.0, 0.0, 1.0);
glMapGrid1f (50, 0.0, 1.0);
glEvalMesh1 (GL_LINE, 0, 50);
```

Mediante el uso de las funciones `glMapGrid1` y `glEvalMesh1`, podemos dividir un curva en un número de segmentos y seleccionar el parámetro de espaciado de cada segmento de acuerdo con su curvatura. Por tanto, a un segmento con más oscilaciones se le podrían asignar más intervalos y a una sección más recta de la curva se le podrían asignar menos intervalos.

En lugar de visualizar curvas de Bézier, podemos utilizar la función `glMap1` para especificar valores para otras clases de datos. Hay disponibles otras siete constantes simbólicas de OpenGL para este propósito. Con la constante simbólica `GL_MAP1_COLOR_4`, utilizamos la matriz `ctrl1Pts` para especificar una lista de colores de cuatro elementos (rojo, verde, azul, alfa). Después se puede generar un conjunto de colores linealmente interpolados para su uso en una aplicación, y estos valores de color generados no cambian la configuración actual del estado de color. De forma similar, podemos especificar una lista de valores de la tabla de color indexado con `GL_MAP1_INDEX`. Y una lista de vectores tridimensionales normales a la superficie se especifica en la matriz `ctrl1Pts` cuando utilizamos la constante simbólica `GL_MAP1_NORMAL`. Las cuatro restantes constantes simbólicas se utilizan con listas de información sobre la textura de la superficie.

Se pueden activar simultáneamente múltiples funciones `glMap1`, y las llamadas a `glEvalCoord1` o `glMapGrid1` y `glEvalMesh1` producen entonces puntos con datos para cada tipo de datos habilitado. Esto nos permite generar combinaciones de posiciones en coordenadas, valores de color, vectores normales a la superficie y datos de la textura de la superficie. Pero no podemos activar simultáneamente `GL_MAP1_VERTEX_3` y `MAP1_VERTEX_4`. Sólo podemos activar uno de los generadores de texturas de la superficie en cualquier momento.

## Funciones OpenGL para superficies con *splines* de Bézier

La activación y la especificación de parámetros de las subrutinas de OpenGL para superficies de Bézier se realizan con:

```
glMap2* (GL_MAP2_VERTEX_3, uMin, uMax, uStride, nuPts,
          vMin, vMax, vStride, nvPts, *ctrlPts);
 glEnable (GL_MAP2_VERTEX_3);
```

Se utiliza un código de sufijo f o d en `glMap2` para indicar si los valores de los datos se especifican en punto flotante o de doble precisión. Para una superficie, especificamos los valores mínimo y máximo tanto del parámetro  $u$  como del parámetro  $v$ . Las coordenadas cartesianas tridimensionales de los puntos de control de Bézier se enumeran en la matriz de doble índice `ctrlPts`, y el tamaño entero de la matriz se proporciona con los parámetros `nuPts` y `nvPts`. Si hay que especificar los puntos de control con coordenadas homogéneas de cuatro dimensiones, utilizamos la constante simbólica `GL_MAP2_VERTEX_4` en lugar de `GL_MAP2_VERTEX_3`. El incremento entero entre el comienzo de los valores de las coordenadas del punto de control  $\mathbf{p}_{j,k}$  lo proporciona `uStride`. Y el incremento entero entre el comienzo de las coordenadas del punto de control  $\mathbf{p}_{j,k}$  y las coordenadas del punto de control  $\mathbf{p}_{j,k+1}$  lo proporciona `vStride`. Esto permite entrelazar los datos de las coordenadas con otros datos, de modo que sólo necesitamos especificar los incrementos para localizar los valores de las coordenadas. Desactivamos las subrutinas para las superficies de Bézier con:

```
glDisable {GL_MAP2_VERTEX_3}
```

Las posiciones de coordenadas sobre la superficie de Bézier se pueden calcular con:

```
glEvalCoord2* (uValue, vValue);
```

o con

```
glEvalCoord2*v (uvArray);
```

Al parámetro `uValue` se le asigna algún valor dentro del intervalo que varía desde `uMin` a `uMax`, y al parámetro `vValue` se le asigna un valor dentro del intervalo que varía desde `vMin` a `vMax`. El vector de versión es `uvArray = (uValue, vValue)`. El código de sufijo para cualquiera de las dos funciones puede ser f o d. La función `glEvalCoord2` calcula una posición de coordenadas utilizando la Ecuación 8.51 con los valores de los parámetros:

$$u = \frac{uValue - uMin}{uMax - uMin}, \quad v = \frac{vValue - vMin}{vMax - vMin} \quad (8.100)$$

que mapea cada uno de los valores `uValue` y `vValue` al intervalo que varía de 0 a 1.0.

Para visualizar una superficie de Bézier, invocamos repetidamente `glEvalCoord2`, que genera una serie de funciones `glVertex3`. Esto es similar a la generación de una curva con *splines*, excepto en que ahora tenemos dos parámetros,  $u$  y  $v$ . Por ejemplo, una superficie definida con 16 puntos de control, distribuidos en una cuadrícula de tamaño 4 por 4, se puede visualizar como un conjunto de líneas de superficie con el código siguiente. El incremento para los valores de las coordenadas en la dirección de  $u$  es 3, y el incremento en la dirección de  $v$  es 12. Cada posición en coordenadas se especifica con tres valores, siendo la coordenada  $y$  de cada grupo de cuatro posiciones constante.

```
GLfloat ctrlPts [4][4][3] = {
    { {-1.5, -1.5, 4.0}, {-0.5, -1.5, 2.0},
      {-0.5, -1.5, -1.0}, { 1.5, -1.5, 2.0} },
    { {-1.5, -0.5, 1.0}, {-0.5, -0.5, 3.0},
      { 0.5, -0.5, 0.0}, { 1.5, -0.5, -1.0} },
    { {-1.5, 0.5, 4.0}, {-0.5, 0.5, 0.0},
      { 0.5, 0.5, 3.0}, { 1.5, 0.5, 4.0} },
    { {-1.5, 1.5, -2.0}, {-0.5, 1.5, -2.0},
      { 0.5, 1.5, 0.0}, { 1.5, 1.5, -1.0} }
};
```

```

glMap3f (GL_MAP2_VERTEX_3, 0.0, 1.0, 3, 4,
          0.0, 1.0, 12, 4, &ctrlPts[0][0][0]);
 glEnable (GL_MAP2_VERTEX_3);

 GLint k, j;

 glColor3f (0.0, 0.0, 1.0);
 for (k = 0; k <= 8; k++)
 {
    glBegin (GL_LINE_STRIP); // Genera las líneas de la superficie
                           Bézier.
    for (j = 0; j <= 40; j++)
       glEvalCoord2f (GLfloat (j) / 40.0, GLfloat (k) / 8.0);
    glEnd ( );
    glBegin (GL_LINE_STRIP);
    for (j = 0; j <= 40; j++)
       glEvalCoord2f (GLfloat (k) / 8.0, GLfloat (j) / 40.0);
    glEnd ( );
 }

```

En lugar de utilizar la función `glEvalCoord2`, podemos generar valores del parámetro uniformemente espaciados sobre la superficie con:

```

glMapGrid2* (nu, u1, u2, nv, v1, v2);
glEvalMesh2 (mode, nu1, nu2, nv1, nv2);

```

El código de sufijo para `glMapGrid2` es de nuevo `f` o `d`, y al parámetro `mode` se le puede asignar el valor `GL_POINT`, `GL_LINE` o `GL_FILL`. Se produce una cuadrícula bidimensional de puntos, con `nu` intervalos igualmente espaciados entre `u1` y `u2`, y con `nv` intervalos igualmente espaciados entre `v1` y `v2`. El rango entero correspondiente al parámetro `u` varía desde `nu1` a `nu2`, y el rango entero correspondiente al parámetro `v` varía desde `nv1` a `nv2`.

Para una superficie que hay que visualizar como una cuadrícula de polilíneas, la salida de `glMapGrid2` y `glEvalMesh2` es la misma que la del siguiente fragmento de programa, excepto en las condiciones que evitan los errores de redondeo en los valores inicial y final de las variables del bucle. En el comienzo de los bucles, el argumento de `glEvalCoord1` se establece en `(u1, v1)`. Y al final del bucle, el argumento de `glEvalCoord1` se establece en `(u2, v2)`.

```

for (k = nu1; k <= nu2; k++) {
    glBegin (GL_LINES);
    for (j = nv1; j <= nv2; j++)
        glEvalCoord2f (u1 + k * (u2 - u1) / nu,
                      v1 + j * (v2 - v1) / nv);
    glEnd ( );
}
for (j = nv1; j <= nv2; j++) {
    glBegin (GL_LINES);
    for (k = nu1; k <= nu2; k++)

```

```

        glEvalCoord2f (u1 + k * (u2 - u1) / nu,
                      v1 + j * (v2 - v1) / nv);
    glEnd ( );
}

```

De forma similar, para una superficie visualizada como un conjunto de caras poligonales rellenas (`mode = GL_FILL`), la salida de `glMapGrid2` y `glEvalMesh2` es la misma que la del siguiente fragmento de programa, excepto en las condiciones que evitan los errores de redondeo en los valores inicial y final de las variables del bucle.

```

for (k = nu1; k < nu2; k++) {
    glBegin (GL_QUAD_STRIP);
    for (j = nv1; j <= nv2; j++) {
        glEvalCoord2f (u1 + k * (u2 - u1) / nu,
                      v1 + j * (v2 - v1) / nv);
        glEvalCoord2f (u1 + (k + 1) * (u2 - u1) / nu,
                      v1 + j * (v2 - v1) / nv);
    }
}

```

Podemos utilizar la función `glMap2` para especificar los valores de otras clases de datos, del mismo modo que hicimos con `glMap1`. Para este propósito hay disponibles constantes simbólicas similares, tales como `GL_MAP2_COLOR_4` y `GL_MAP2_NORMAL`. Y podemos activar múltiples funciones `glMap2` para generar varias combinaciones de datos.

## Funciones GLU para curvas con *splines B*

Aunque las subrutinas de GLU para *splines B* se denominan funciones «nurbs», se pueden utilizar para generar *splines B* que no son ni no uniformes ni racionales. Por tanto, podemos utilizar estas subrutinas para visualizar un *spline B* polinómico que tiene un espacio uniforme de nudos. Y también se puede utilizar las subrutinas de GLU para producir *splines* de Bézier, racionales y no racionales. Para generar un *spline B* (o *spline* de Bézier), necesitamos definir el nombre del *spline*, activar el sombreador de *splines B* de GLU, y entonces definir los parámetros del *spline*.

Las líneas siguientes ilustran la secuencia básica de llamadas para visualizar una curva con *splines B*.

```

GLUnurbsObj *curveName;

curveName = gluNewNurbsRenderer ();
gluBeginCurve (curveName);
    gluNurbsCurve (curveName, nknots, *knotVector, stride, *ctrlPts,
                  degParam, GL_MAP1_VERTEX_3);
gluEndCurve (curveName);

```

En la primera línea, asignamos un nombre a la curva, a continuación invocamos las subrutinas de sombreado de GLU para *splines B* para esta curva utilizando el comando `gluNewNurbsRenderer`. Se asigna un

valor 0 a `curveName` cuando no hay memoria disponible suficiente para crear una curva con *splines B*. Dentro de un par `gluBeginCurve/gluEndCurve`, establecemos a continuación los atributos de la curva utilizando la función `gluNurbsCurve`. Esto nos permite configurar múltiples secciones de curva, y cada sección se referencia con un nombre de curva distinto. El parámetro `knotVector` hace referencia al conjunto de valores de los nudos en punto flotante, y el parámetro entero `nknots` especifica el número de elementos del vector de nudos. El grado del polinomio es `degParam - 1`. Enumeramos los valores de las coordenadas de los puntos de control tridimensionales en el argumento `ctrlPts`, que es un vector y contiene `nknots - degParam` elementos. Y el desplazamiento entero inicial entre el comienzo de las sucesivas posiciones en coordenadas en el vector `ctrlPts` se especifica con el parámetro entero `stride`. Si las posiciones de los puntos de control son contiguas (no entrelazadas con otros tipos de datos), el valor de `stride` se establece en 3. Eliminamos un *spline B* definido con:

```
gluDeleteNurbsRenderer (curveName);
```

A modo de ejemplo de utilización de las subrutinas de GLU para visualización de una curva con *splines*, el código siguiente genera un polinomio cúbico de Bézier. Para obtener esta curva cúbica, establecemos el parámetro de grado en el valor 4. Utilizamos cuatro puntos de control y seleccionamos una secuencia de nudos de ocho elementos abierta uniforme con cuatro valores repetidos en cada extremo.

---

```
GLfloat knotVector [8] = {0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0};
GLfloat ctrlPts [4][3] = { {-4.0, 0.0, 0.0}, {-2.0, 8.0, 0.0},
                           {2.0, -8.0, 0.0}, {4.0, 0.0, 0.0} };
GLUnurbsObj *cubicBezCurwe;

cubicBezCurve = gluNewNurbsRenderer ();
gluBeginCurve (cubicBezCurve);
    gluNurbsCurve (cubicBezCurve, 8, knotVector, 3, &ctrlPts [0][0],
                   4, GL_MAP1_VERTEX_3);
    gluEndCurve(cubicBezCurve);
```

---

Para crear una curva con *splines B* racionales, sustituimos la constante simbólica `GL_MAP1_VERTEX_3` por `GL_MAP1_VERTEX_4`. A continuación, se utilizan las coordenadas homogéneas de cuatro dimensiones ( $x_h, y_h, z_h, h$ ) para especificar los puntos de control y la división homogénea resultante produce el polinomio racional deseado.

También podemos utilizar la función `gluNurbsCurve` para especificar las listas de los valores de color, vectores normales, o propiedades de la textura de la superficie, del mismo modo que hicimos con las funciones `glMap1` y `glMap2`. Cualquiera de las constantes simbólicas, tales como `GL_MAP1_COLOR_4` o `GL_MAP1_NORMAL`, se pueden utilizar como último argumento en la función `gluNurbsCurve`. Cada llamada se enumera a continuación dentro del par `gluBeginCurve/gluEndCurve`, con dos restricciones: no podemos enumerar más de una función para cada tipo de dato y debemos incluir exactamente una función para generar la curva con *splines B*.

Una curva con *splines B* se divide automáticamente en un número de secciones y se visualiza como una polilínea con las subrutinas de GLU. Pero también se puede seleccionar una gran variedad de opciones de sombreado de *splines B* con llamadas repetidas a la siguiente función.

```
gluNurbsProperty (splineName, property, value);
```

Al parámetro `splineName` se le asigna el nombre del *spline B*, al parámetro `property` se le asigna una constante simbólica de GLU que identifica la propiedad de sombreado que queremos cambiar y al parámetro `value` se le asigna un valor numérico en punto flotante o una constante simbólica de GLU que establece el

valor de la propiedad seleccionada. Se pueden especificar varias funciones `gluNurbsProperty` después de la línea con `gluNewNurbsRenderer`. Muchas de las propiedades que se pueden cambiar usando la función `gluNurbsProperty` son parámetros de la superficie, como se describe en la sección siguiente.

## Funciones GLU para la creación de superficies con *splines* B

El siguiente fragmento de código ilustra la secuencia básica de llamadas para generar una superficie con *splines* B.

```
GLUnurbsObj *surfName

surfName = gluNewNurbsRenderer ( );
gluNurbsProperty (surfName, property1, value1);
gluNurbsProperty (surfName, property2, value2);
gluNurbsProperty (surfName, property3, value3);

.

.

.

gluBeginSurface (surfName);
    gluNurbsSurface (surfName, nuKnots, uKnotVector, nvKnots,
                      vKnotVector, uStride, vStride, &ctrlPts [0][0][0],
                      uDegParam, vDegParam, GL_MAP2_VERTEX_3);
    gluEndSurface (surfName);
```

Por lo general, el código y los parámetros de GLU para definir una superficie con *splines* B es similar al de una curva con *splines* B. Después de invocar las subrutinas de sombreado de *splines* B con `gluNewNurbsRenderer`, podríamos especificar valores opcionales de las propiedades de la superficie. Los atributos de la superficie se establecen a continuación con una llamada a `gluNurbsSurface`. De este modo se pueden definir múltiples superficies, cada una con un nombre distinto. El sistema devuelve un valor 0 en la variable `surfName` cuando no hay suficiente memoria disponible para almacenar un objeto de tipo *spline* B. Los parámetros `uKnotVector` y `vKnotVector` hacen referencia a las matrices con los valores de los nudos en punto flotante en las direcciones de los parámetros *u* y *v*. Podemos especificar el número de elementos de cada vector de nudos con los parámetros `nuKnots` y `nvKnots`. El grado del polinomio en el parámetro *u* lo proporciona el valor de `uDegParam` - 1, y el grado del polinomio en el parámetro *v* es el valor de `vDegParam` - 1. Enumeramos los valores en punto flotante de las coordenadas tridimensionales de los puntos de control del parámetro `ctrlPts`, que es un vector y que contiene  $(\text{nuKnots} - \text{uDegParam}) \times (\text{nvKnots} - \text{vDegParam})$  elementos. El desplazamiento entero inicial entre el comienzo de los sucesivos puntos de control en la dirección paramétrica *u* se especifica con el parámetro entero `uStride`, y el desplazamiento en la dirección paramétrica *v* se especifica con el parámetro entero `vStride`. Borramos una superficie con *splines* para liberar su memoria reservada con la misma función `gluDeleteNurbsRenderer` que usamos para una curva con *splines* B.

De forma predeterminada, una superficie con *splines* B se visualiza automáticamente como un conjunto de áreas de relleno poligonales mediante las subrutinas GLU, pero podemos elegir otras opciones y parámetros de visualización. Se pueden establecer nueve propiedades, con dos o más posibles valores en cada propiedad, en una superficie con *splines* B. Como ejemplo de definición de propiedades, el siguiente fragmento de código especifica una visualización de una superficie en su modelo alámbrico teselada con triángulos.

```
gluNurbsProperty (surfName, GLU_NURBS_MODE,
                   GLU_NURBS_TESSELLATOR);
```

```
gluNurbsProperty (surfName, GLU_DISPLAY_MODE,
                   GLU_OUTLINE_POLYGON};
```

Las subrutinas de teselación de GLU dividen la superficie en un conjunto de triángulos y muestran cada triángulo como el contorno de un polígono. Además, estas primitivas para triángulos se pueden recuperar utilizando la función `gluNurbsCallback`. Otros valores de la propiedad `GLU_DISPLAY_MODE` son `GLU_OUTLINE_PATCH` y `GLU_FILL` (el valor predeterminado). Mediante el valor `GLU_OUTLINE_PATCH`, también obtenemos una visualización en modelo alámbrico, pero la superficie no se divide en secciones triangulares. En su lugar, se dibuja el contorno de la superficie original, junto con cualquier curva de recorte que se haya especificado. El único valor restante de la propiedad `GLU_NURBS_MODE` que se puede modificar es `GLU_NURBS_RENDERER`, que sombra los objetos sin dejar disponibles los datos teselados para su devolución.

Establecemos el número de puntos de muestreo por unidad de longitud con las propiedades `GLU_U_STEP` y `GLU_V_STEP`. El valor predeterminado para cada una de ellas es 100. Para establecer los valores de muestreo de  $u$  o  $v$ , también debemos establecer la propiedad `GLU_SAMPLING_METHOD` con el valor `GLU_DOMAIN_DISTANCE`. Se pueden utilizar otros valores con la propiedad `GLU_SAMPLING_METHOD` para especificar cómo se lleva a cabo la teselación de la superficie. Las propiedades `GLU_SAMPLING_TOLERANCE` y `GLU_PARAMETRIC_TOLERANCE` se utilizan para establecer las longitudes máximas de muestreo. Modificando la propiedad `GLU_CULLING` con el valor `GL_TRUE`, podemos mejorar las prestaciones del sombreado no teselando objetos que se encuentran fuera del volumen de visualización. El valor predeterminado de la selección (culling) de GLU es `GL_FALSE`. Y la propiedad `GLU_AUTO_LOAD_MATRIX` permite que se descarguen del servidor de OpenGL las matrices de visualización y proyección cuando su valor es `GL_TRUE` (el valor predeterminado). De lo contrario, si cambiamos el valor a `GL_FALSE`, una aplicación debe proporcionar estas matrices empleando la función `gluLoadSamplingMatrices`.

Para determinar el valor actual de una propiedad de un *spline* B, utilizamos la siguiente función de consulta.

```
gluGetNurbsProperty (splineName, property, value);
```

Para nombre de *spline* `splineName` y una propiedad `property` especificadas el valor correspondiente se obtiene en el parámetro `value`.

Cuando la propiedad `GLU_AUTO_LOAD_MATRIX` se establece en el valor `GL_FALSE`, invocamos,

```
gluLoadSamplingMatrices (splineName, modelviewMat, projMat,
                        viewport);
```

Esta función especifica la matriz de vista del modelo, la matriz de proyección y el visor que hay que utilizar en las subrutinas de muestreo y de selección para un objeto de tipo *spline*. Las matrices de vista de modelo y de proyección actuales se pueden obtener mediante llamadas a la función `glGetFloatv`, y la vista actual se puede obtener con una llamada a `glGetIntegerv`.

Varios eventos asociados a los objetos de tipo *spline* se procesan empleando,

```
gluNurbsCallback (splineName, event, fcn);
```

Al parámetro `event` se le asigna una constante simbólica de GLU, y el parámetro `fcn` especifica una función que hay que invocar cuando el evento correspondiente a la constante de GLU se produce. Por ejemplo, si establecemos el parámetro `event` en `GLU_NURBS_ERROR`, entonces se llama a `fcn` cuando se produce un error. Las subrutinas para *splines* de GLU utilizan otros eventos para devolver los polígonos de OpenGL generados por el proceso de teselación. La constante simbólica `GL_NURBS_BEGIN` indica el comienzo de primitivas tales como segmentos de línea, triángulos o cuadriláteros, y `GL_NURBS_END` indica el final de la primitiva. El argumento de la función de comienzo de una primitiva es una constante simbólica tal como `GL_LINE_STRIP`, `GL_TRIANGLES` o `GL_QUAD_STRIP`. La constante simbólica indica que aquellos datos con las coordenadas tridimensionales se deben suministrar y se invoca una función de los vértices. Hay disponibles constantes adicionales para indicar otros datos tales como los valores de color.

Los valores de los datos de la función `gluNurbsCallback` se proporcionan mediante:

```
gluNurbsCallbackData (splineName, dataValues);
```

Al parámetro `splineName` se le asigna el nombre de objeto de tipo *spline* que hay que teselar, y al parámetro `dataValues` se le asigna una lista con los valores de los datos.

## Funciones GLU para el recorte de superficies

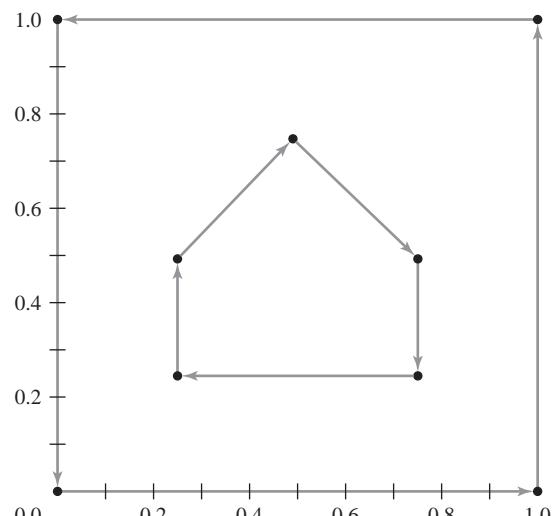
Se especifica un conjunto de una o más curvas de recorte bidimensionales de una superficie con *splines B* mediante el siguiente fragmento de código.

```
gluBeginTrim (surfName);
    gluPwlCurve (surfName, nPts, *curvePts, stride, GLU_MAP1_TRIM_2);
    .
    .
    .
gluEndTrim (surfName);
```

El parámetro `surfName` es el nombre de la superficie con *splines B* que hay que recortar. En el parámetro `curvePts`, que es un vector ya que contiene `nPts` posiciones de coordenadas, se especifica un conjunto de coordenadas en punto flotante para la curva de recorte. En el parámetro `stride` se especifica un desplazamiento inicial entero entre las sucesivas posiciones de coordenadas. Las coordenadas especificadas de la curva se utilizan para generar una función de recorte lineal por tramos para la superficie con *splines B*. En otras palabras, la «curva» de recorte generada es una polilínea. Si los puntos de la curva se deben proporcionar en el espacio tridimensional y homogéneo del parámetro  $(u, v, h)$ , entonces el argumento final de `gluPwlCurve` se establece en la constante simbólica de GLU `GLU_MAP1_TRIM_3`.

También podemos utilizar una o más funciones `gluNurbsCurve` como en el caso del recorte de curvas. Y podemos construir curvas de recorte que sean combinaciones de funciones `gluPwlCurve` y `gluNurbsCurve`. Cualquier «curva» de recorte de GLU que se especifique no se debe intersectar y debe ser una curva cerrada.

El siguiente código ilustra las funciones de recorte de GLU para una superficie cúbica de Bézier. En primer lugar establecemos las coordenadas de los puntos de la curva de recorte más exterior. Estas posiciones



**FIGURA 8.54.** Una curva exterior de recorte alrededor del perímetro del cuadrado unidad se especifica en dirección contraria al movimiento de las agujas del reloj, y las secciones de la curva interior de recorte se definen en el sentido del movimiento de las agujas del reloj.

se especifican en sentido contrario al movimiento de las agujas del reloj alrededor del cuadrado unidad. A continuación, establecemos las coordenadas de los puntos de la curva de recorte más interna en dos secciones, y estas posiciones se especifican en el sentido de las agujas del reloj. Y los vectores de nudos tanto para la superficie como para la primera sección de la curva de recorte interior se configuran para producir curvas cúbicas de Bézier. En la Figura 8.54 se muestra un dibujo de las curvas de recorte interior y exterior sobre el cuadrado unidad.

```

GLUnurbsObj *bezSurface;

GLfloat outerTrimPts [5][2] = { {0.0, 0.0}, {1.0, 0.0}, {1.0, 1.0},
                                {0.0, 1.0}, {0.0, 0.0} };
GLfloat innerTrimPts1 [3][2] = { {0.25, 0.5}, {0.5, 0.75},
                                {0.75, 0.5} };
GLfloat innerTrimPts2 [4][2] = { {0.75, 0.5}, {0.75, 0.25},
                                {0.25, 0.25}, {0.25, 0.5} };

GLfloat surfKnots [8] = (0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0);
GLfloat trimCurveKnots [8] = (0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0);

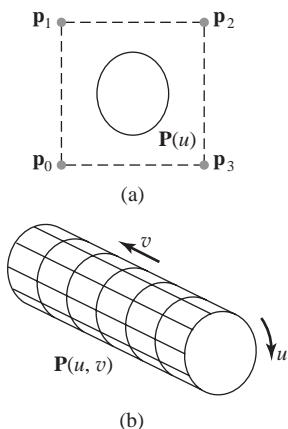
bezSurface = gluNewNurbsRenderer ( );

gluBeginSurface (bezSurface);
    gluNurbsSurface (bezSurface, 8, surfKnots, 8, surfKnots, 4 * 3, 3,
                      &ctrlPts [0][0][0], 4, 4, GL_MAP2_VERTEX_3);
gluBeginTrim (bezSurface);
    /* Curva de recorte exterior en sentido contrario */
    /* a las agujas del reloj.*/
    gluPwlCurve (bezSurface, 5, &outerTrimPts [0][0], 2,
                  GLU_MAP1_TRIM_2);
gluEndTrim (bezSurface);
gluBeginTrim (bezSurface);
    /* Secciones de la curva de recorte interior*/
    /* en el sentido de las agujas del reloj.*/
    gluPwlCurve (bezSurface, 3, &innerTrimPts1 [0][0], 2,
                  GLU_MAP1_TRIM_3);
    gluNurbsCurve (bezSurface, 8, trimCurveKnots, 2,
                    &innerTrimPts2 [0][0], 4, GLU_MAP1_TRIM_2);
gluEndTrim (bezSurface);
gluEndSurface (bezSurface);

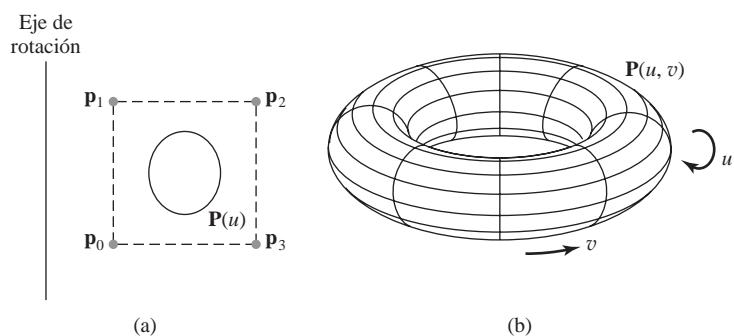
```

## 8.19 REPRESENTACIONES DE BARRIDO

Los paquetes de modelado de sólidos proporcionan a menudo un gran número de técnicas de construcción. **Las representaciones de barrido** son útiles para construir objetos tridimensionales que poseen simetrías de traslación, de rotación o de otra clase. Podemos representar tales objetos especificando una forma bidimensional y un recorrido por el que se mueve la forma a través de una región del espacio. Se puede disponer de



**FIGURA 8.55.** Construcción de un sólido con un barrido de traslación. La traslación de los puntos de control de una curva con *splines* periódicos de (a) genera el sólido mostrado en (b), cuya superficie se puede describir con la función de punto  $\mathbf{P}(u, v)$ .



**FIGURA 8.56.** Construcción de un sólido con un barrido de rotación. La rotación de los puntos de control de una curva con *splines* periódicos (a) alrededor del eje de rotación dado genera el sólido mostrado en (b), cuya superficie se puede describir con la función de punto  $\mathbf{P}(u, v)$ .

un conjunto de primitivas bidimensionales, tales como círculos o rectángulos, para representaciones de barrido como opciones de menú. Entre otros métodos para obtener figuras bidimensionales se pueden incluir las construcciones con curvas con *splines* y las secciones rectas de objetos sólidos.

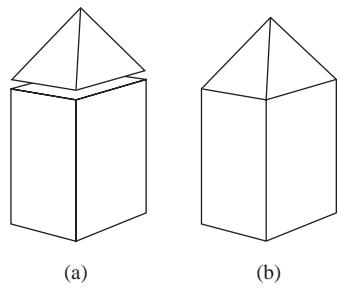
La Figura 8.55 ilustra un barrido de traslación. La curva con *splines* periódicos de la Figura 8.55(a) define la sección recta del objeto. Después realizamos un barrido de traslación para una distancia específica moviendo los puntos de control  $p_0$  a  $p_3$  a lo largo de una trayectoria de una línea recta perpendicular al plano de la sección recta. A intervalos a lo largo de esta trayectoria, repetimos la forma de la sección recta y dibujamos un conjunto de líneas de conexión en la dirección del recorrido para obtener la representación alámbrica mostrada en la Figura 8.55(b).

En la Figura 8.56 se proporciona un ejemplo de un diseño de un objeto utilizando un barrido de rotación. Esta vez, la sección recta con *spline* periódico se rota alrededor de un eje específico en el plano de la sección recta, para producir la representación alámbrica mostrada en la Figura 8.56(b). Se puede elegir cualquier eje para el barrido de rotación. Si utilizamos un eje de rotación perpendicular al plano de la sección recta con *splines* de la Figura 8.56(a), generamos una forma bidimensional. Pero si la sección recta mostrada en esta figura tiene profundidad, entonces utilizamos un objeto tridimensional para generar otro.

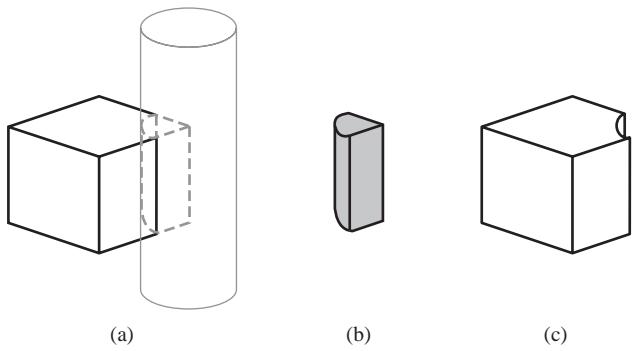
Por lo general, podemos especificar construcciones de barrido utilizando cualquier trayectoria. En barridos de rotación, podemos realizar el movimiento a lo largo de una trayectoria circular de cualquier distancia angular desde 0 a 360°. En trayectorias no circulares, podemos especificar la función de curva que describe la trayectoria y la distancia recorrida a lo largo de la trayectoria. Además, podemos variar la forma o el tamaño de la sección recta a lo largo de la trayectoria de barrido. O podríamos variar la orientación de la sección recta, con respecto a la trayectoria de barrido a medida que movemos la forma a través de una región del espacio.

## 8.20 MÉTODOS DE GEOMETRÍA CONSTRUCTIVA DE SÓLIDOS

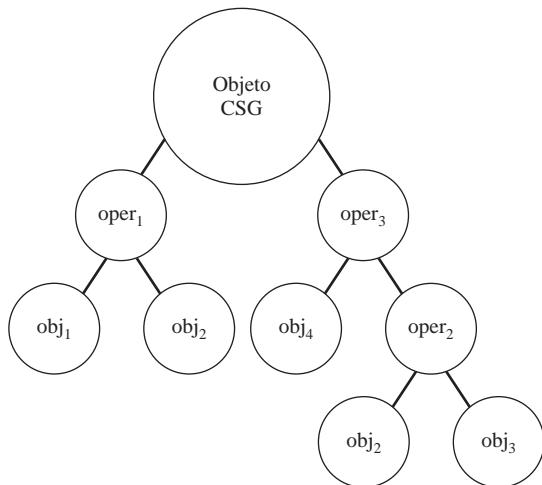
Otra técnica de modelado de sólidos consiste en generar un nuevo objeto a partir de dos objetos tridimensionales utilizando una operación de conjuntos. Este método de modelado, llamado **geometría constructiva de sólidos** (Constructive Solid Geometry; CSG), crea el nuevo objeto aplicando las operaciones de unión, intersección o diferencia de dos sólidos seleccionados.



**FIGURA 8.57.** La combinación de los dos objetos mostrados en (a) utilizando un operación de unión produce el nuevo objeto sólido compuesto de (b).



**FIGURA 8.58.** Dos objetos que se superponen (a) se pueden combinar para producir el objeto con forma de cuña de (b), utilizando la operación de intersección, o el bloque modificado mostrado en (c), utilizando la operación diferencia.



**FIGURA 8.59.** Un ejemplo de representación de árbol CSG de un objeto.

Las Figuras 8.57 y 8.58 muestran ejemplos de formación de formas nuevas utilizando las operaciones de los conjuntos. En la Figura 8.57(a), un bloque y una pirámide se colocan adyacentes una a otra. Mediante la operación de unión, obtenemos el objeto combinado de la Figura 8.57(b). La Figura 8.58(a) muestra un bloque y un cilindro que se superponen en volumen. Utilizando la operación de intersección, obtenemos el sólido de la Figura 8.58(b). Mediante la operación diferencia, podemos visualizar el sólido mostrado en la Figura 8.58(c).

Una aplicación de CSG comienza con un conjunto inicial de objetos tridimensionales, llamados primitivas CSG, tales como un bloque, una pirámide, un cilindro, un cono, una esfera y tal vez algunos sólidos con superficies con *splines*. Las primitivas se pueden proporcionar en el paquete CSG como un menú de selección, o las primitivas se podrían formar utilizando métodos de barrido, construcciones con *splines*, u otros procedimientos de modelado. En un paquete interactivo de CSG, podemos seleccionar una operación (unión, intersección o diferencia) y arrastrar dos primitivas a una posición dentro de alguna región del espacio para formar un nuevo objeto. Este nuevo objeto se podría entonces combinar con una de las formas existentes para formar otro objeto nuevo. Podemos continuar este proceso hasta que tengamos la forma final del objeto que estamos diseñando. Un objeto construido con este procedimiento se representa con un árbol binario, como en la Figura 8.59.

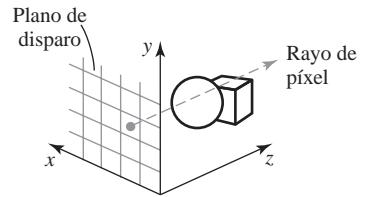


FIGURA 8.60. Implementación de operaciones CSG utilizando trazado de rayos.

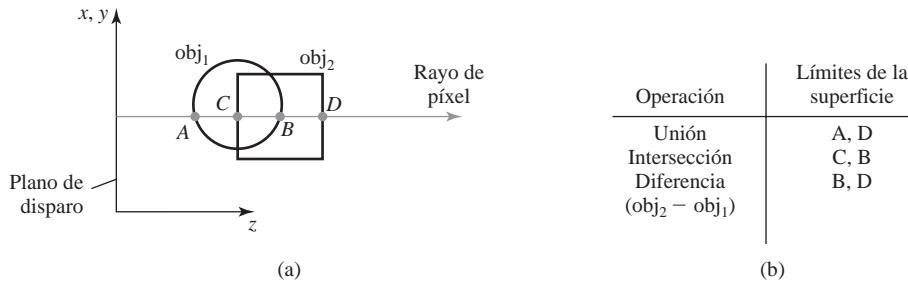


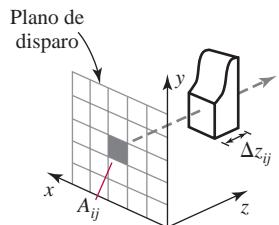
FIGURA 8.61. Determinación de los límites de la superficie a lo largo de un rayo de píxel.

Los métodos de **trazado de rayos** (*ray casting*) se utilizan habitualmente para implementar operaciones de geometría constructiva de sólidos cuando los objetos se describen mediante representaciones por contorno. Aplicamos el trazado de rayos determinando los objetos que son intersectados por un conjunto de líneas paralelas que emanan del plano *xy* según la dirección del eje *z*. Este plano se denomina **plano de disparo** (*firing plane*) y cada rayo tiene su origen en un píxel, como se muestra en la Figura 8.60. Entonces calculamos las intersecciones con la superficie a lo largo de la trayectoria de cada rayo, y ordenamos los puntos de intersección según la distancia al plano de disparo. Los límites de la superficie del objeto compuesto se determinan a continuación mediante la operación de conjunto especificada. En la Figura 8.61 se proporciona un ejemplo de determinación mediante trazado de rayos de los límites de la superficie de un objeto CSG, que muestra las secciones rectas *yz* de dos objetos (un bloque y una esfera) y la trayectoria de un rayo perpendicular de píxel al plano de disparo. En la operación de unión, el volumen nuevo es el interior combinado ocupado por los dos objetos. En la operación de intersección, el volumen nuevo es la región interior común a ambos objetos. Y una operación de diferencia sustrae el interior de un objeto del otro donde los dos objetos se superponen.

Cada primitiva de CSG se define habitualmente en su propias coordenadas locales (de modelado). La posición correspondiente en coordenadas universales se determina mediante las matrices de transformación de modelado utilizadas para crear una posición de solapamiento con otro objeto. La inversa de las matrices de modelado del objeto se pueden utilizar entonces para transformar los rayos de píxel a coordenadas de modelado, donde los cálculos de las intersecciones con la superficie se realizan en las primitivas individuales. Entonces las intersecciones con la superficie de los dos objetos superpuestos se ordenan según la distancia a lo largo de la trayectoria del rayo y se usan para determinar los límites del objeto compuesto, según la operación de conjuntos especificada. Este procedimiento se repite para cada par de objetos que hay que combinar en el árbol CSG de un objeto concreto.

Una vez que se ha diseñado un objeto CSG, el trazado de rayos se utiliza para determinar propiedades físicas, tales como el volumen y la masa. Para determinar el volumen del objeto, aproximamos el área de cada píxel del plano de disparo mediante un pequeño cuadrado (Figura 8.62). Podemos entonces aproximar el volumen  $V_{ij}$  de una sección transversal del objeto de área  $A_{ij}$  en la dirección de la trayectoria de un rayo de píxel de la posición  $(i, j)$  de este modo:

$$V_{ij} \approx A_{ij}\Delta z_{ij} \quad (8.101)$$



**FIGURA 8.62.** Determinación del volumen de un objeto en la dirección de la trayectoria de un rayo de un área de píxel  $A_{ij}$  del plano de disparo.

donde  $\Delta z_{ij}$  es la profundidad del objeto en la dirección del rayo desde la posición  $(i, j)$ . Si el objeto tiene agujeros internos,  $\Delta z_{ij}$  es la suma de las distancias entre pares de puntos de intersección en la dirección del rayo. Aproximamos el volumen total del objeto CSG mediante la suma de los volúmenes individuales según las trayectorias de los rayos:

$$V \approx \sum_{i,j} V_{ij} \quad (8.102)$$

Dada la función de densidad,  $\rho(x, y, z)$ , del objeto, podemos aproximar la masa en la dirección del rayo desde la posición  $(i, j)$  con esta integral:

$$m_{ij} \approx A_{ij} \int \rho(x_{ij}, y_{ij}, z) dz \quad (8.103)$$

donde la integral simple se puede aproximar a menudo sin realizar la integral, dependiendo de la forma de la función de densidad. La masa total del objeto CSG se aproxima entonces mediante la suma:

$$m \approx \sum_{i,j} m_{ij} \quad (8.104)$$

Otras propiedades físicas, tales como el centro de masas y el momento de inercia, se pueden obtener con cálculos similares. Podemos mejorar la precisión de los cálculos de los valores de las propiedades físicas mediante rayos adicionales generados desde posiciones de subpíxeles en el plano de disparo.

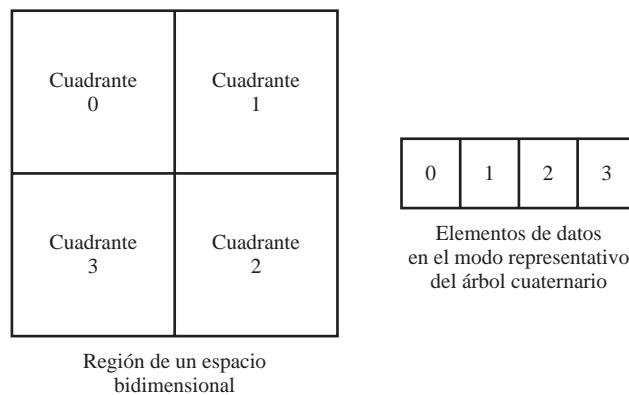
Si las formas de los objetos se representan mediante árboles octales, podemos implementar el conjunto de operaciones de los procedimientos CSG rastreando la estructura del árbol que describe los contenidos de los octantes del espacio. Este procedimiento, descrito en la sección siguiente, busca en los octantes y los suboctantes de un cubo unidad para localizar las regiones ocupadas por los dos objetos que hay que combinar.

## 8.21 ÁRBOLES OCTALES

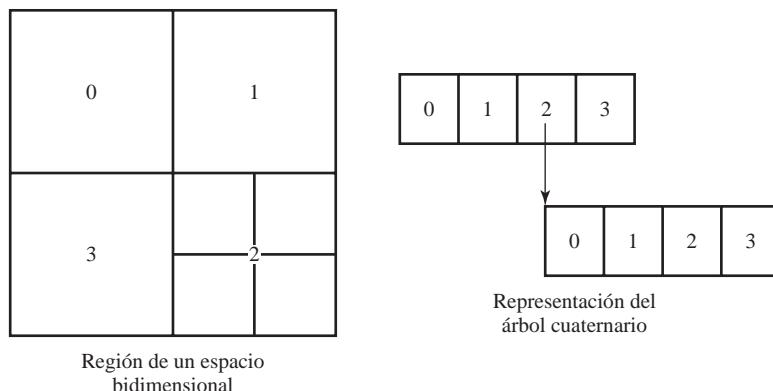
---

Las estructuras jerárquicas con árboles, llamadas **árboles octales**, se utilizan para representar objetos sólidos en algunos sistemas gráficos. La generación de imágenes médicas y otras aplicaciones que requieren la visualización de secciones rectas de objetos utilizan a menudo representaciones con árboles octales. La estructura de un árbol está organizada de modo que cada nodo se corresponde con una región del espacio tridimensional. Esta representación de sólidos se aprovecha de la coherencia espacial para reducir los requerimientos de almacenamiento de los objetos tridimensionales. También proporciona una representación adecuada para almacenar información acerca del interior de los objetos.

La representación mediante árboles octales de un objeto tridimensional es una ampliación de una técnica de representación bidimensional similar, llamada codificación con **árboles cuaternarios**. Los árboles cuaternarios se generan mediante divisiones sucesivas de una región bidimensional (habitualmente un cuadrado) en cuadrantes. Cada nodo del árbol cuádrico tiene cuatro elementos de datos, uno por cada uno de los cuadrantes de la región (Figura 8.63). Si todos los puntos contenidos en un cuadrante tienen el mismo color (un cuadrante homogéneo), el elemento correspondiente de datos del nodo almacena dicho color. Además, se modifica un indicador (*flag*) en el elemento de datos para indicar que el cuadrante es homogéneo. Si, por ejemplo,



**FIGURA 8.63.** Una región cuadrada del plano  $xy$  dividida en cuadrantes numerados y el nodo asociado del árbol cuaternario con cuatro elementos de datos.

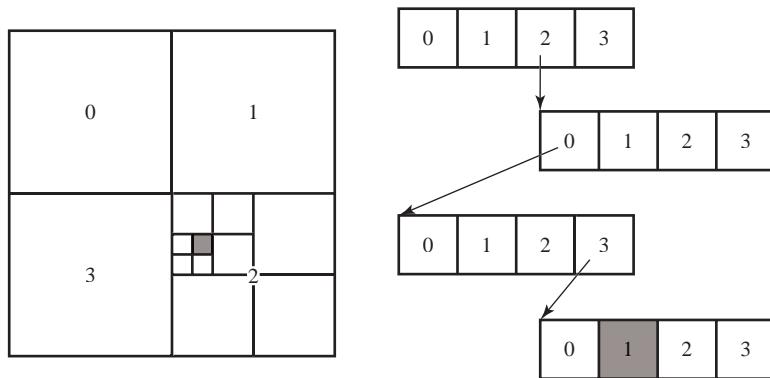


**FIGURA 8.64.** Una región cuadrada del plano  $xy$  con divisiones en cuadrantes de dos niveles y la representación asociada mediante árbol cuaternario.

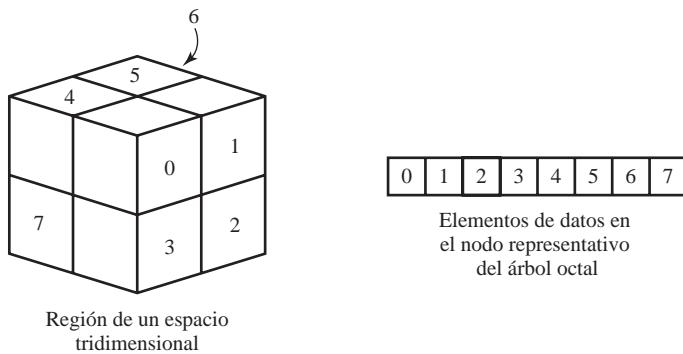
todos los puntos del cuadrante 2 de la Figura 8.63 son de color rojo, el código de color para el rojo se coloca entonces en el elemento de datos número 2 del nodo. De lo contrario el cuadrante es heterogéneo, y se divide en subcuadrantes, como se muestra en la Figura 8.64. En el elemento de datos correspondiente del nodo del cuadrante 2 ahora se modifica un indicador para marcar el cuadrante como heterogéneo y se almacena un puntero al nodo siguiente del árbol cuaternario.

Un algoritmo de generación de un árbol cuaternario comprueba los valores de color asignados a los objetos dentro de una región bidimensional seleccionada y configura los nodos del árbol cuaternario consecuentemente. Si cada cuadrante de espacio original tiene un único color, el árbol cuaternario tiene un único nodo. En el caso de una región heterogénea del plano, las sucesivas subdivisiones en cuadrantes continúan hasta que todas las partes de la región subdividida son homogéneas. La Figura 8.65 muestra la representación mediante un árbol cuaternario de una región que contiene un área con un color liso que es diferente del color uniforme del resto de áreas dentro de dicha región.

Las codificaciones mediante árboles cuaternarios proporcionan ahorros considerables de almacenamiento cuando existen amplias zonas de un único color en una región del espacio, ya que un único nodo puede representar una gran parte del espacio. Y esta técnica de representación se puede utilizar para almacenar los valores del color de los píxeles. Para un área que contiene  $2n$  por  $2n$  píxeles, una representación mediante un árbol cuaternario contiene como máximo  $n$  niveles. Y cada nodo del árbol cuaternario tiene como máximo cuatro descendientes inmediatos.



**FIGURA 8.65.** Una representación mediante árbol cuaternario de una región cuadrada del plano  $xy$  que contiene una única área de primer plano sobre un fondo de color liso.



**FIGURA 8.66.** Un cubo dividido en octantes numerados y el nodo asociado del árbol octal de ocho elementos de datos.

Una técnica de codificación mediante un árbol octal divide una región del espacio tridimensional (habitualmente un cubo) en octantes y almacena ocho elementos de datos por nodo del árbol, como se muestra en la Figura 8.66. Las subregiones individuales del espacio tridimensional subdividido se denominan **elementos de volumen**, o **vóxeles**, por analogía con los píxeles de un área de visualización rectangular. Un elemento de voxel de una representación mediante un árbol octal almacena los valores de la propiedad de una subregión homogénea del espacio. Entre las propiedades de los objetos dentro de una región tridimensional del espacio se puede incluir el color, el tipo de material, la densidad y otras características físicas. Por ejemplo, entre los objetos de una región seleccionada del espacio podrían estar incluidas piedras y árboles o pañuelos de papel, huesos y órganos humanos. Las regiones vacías del espacio se representan mediante el tipo de voxel «vacío» (void). Como en el caso de la representación mediante un árbol cuaternario, un octante heterogéneo de una región se divide hasta que las subdivisiones son homogéneas. En un árbol octal, cada nodo puede tener desde cero a ocho descendientes inmediatos.

Los algoritmos de generación de árboles octales se pueden estructurar para aceptar definiciones de objetos de cualquier forma, tales como una malla poligonal, parches de superficie curvada, o construcciones de geometría sólida. En el caso de un único objeto, el árbol octal se puede construir a partir de la caja que lo contiene (paralelepípeda) determinada mediante las extensiones de coordenadas del objeto.

Una vez que se ha establecido una representación mediante un árbol octal de un objeto sólido, se pueden aplicar al objeto varias subrutinas de manipulación. Se puede aplicar un algoritmo para realizar operaciones de conjuntos a dos representaciones mediante árboles octales de la misma región del espacio. En una operación de unión, se construye un nuevo árbol octal utilizando los nodos del árbol octal de cada uno de los árbo-

les de partida. Para establecer una representación de intersección de dos árboles octales, construimos el nuevo árbol utilizando los octantes donde los dos objetos se superponen. De forma similar, en una operación de diferencia, buscamos las regiones ocupadas por un objeto y no por el otro.

Se ha desarrollado un gran número de otros algoritmos de procesamiento de árboles octales. Las rotaciones tridimensionales, por ejemplo, se realizan aplicando las transformaciones a las regiones espaciales representadas por los octantes ocupados. Para localizar los objetos visibles de una escena, podemos determinar en primer lugar si cualquiera de los octantes frontales están ocupados. Si no, proseguimos con los octantes situados detrás de los octantes frontales. Este proceso continúa hasta que los octantes ocupados son localizados en la dirección del punto de vista. El primer objeto detectado en la dirección de cualquier trayectoria de punto de vista a través de los octantes espaciales, desde la parte frontal hasta la parte trasera, es visible, y la información de dicho objeto se puede transferir a una representación mediante un árbol cuaternario para su visualización.

## 8.22 ÁRBOLES BSP

---

Esta estrategia de representación es similar a la codificación mediante árboles octales, excepto que ahora dividimos el espacio en dos particiones en lugar de ocho en cada etapa. Mediante un árbol de **particionamiento binario del espacio** (binary space-partitioning; **BSP**), subdividimos una escena en dos partes en cada etapa mediante un plano que puede estar en cualquier posición y con cualquier orientación. En una codificación mediante un árbol octal, la escena se subdivide en cada paso mediante tres planos perpendiculares entre sí, alineados con los planos de coordenadas cartesianas.

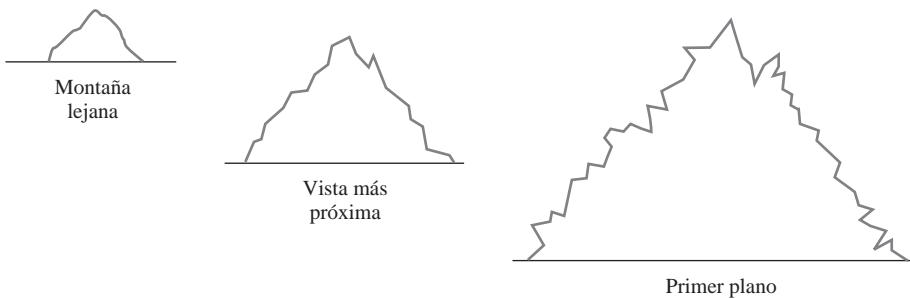
En la subdivisión adaptativa del espacio, los árboles BSP pueden proporcionar un particionamiento más eficiente, ya que podemos posicionar y orientar los planos de corte para acomodarnos a la distribución espacial de los objetos. Esto puede reducir la profundidad de la representación de la escena mediante un árbol, en comparación con un árbol octal y, por tanto, reducir el tiempo de búsqueda en el árbol. Además, los árboles BSP son útiles para identificar la superficie visible y para el particionamiento del espacio en algoritmos de trazado de rayos.

## 8.23 MÉTODOS DE GEOMETRÍA FRACTAL

---

Todas las representaciones de objetos que hemos considerado en las secciones anteriores usaban métodos de geometría euclídea; es decir, las formas de los objetos se han descrito mediante ecuaciones. Estos métodos son adecuados para describir objetos fabricados: aquellos que tienen superficies suaves y formas regulares. Pero los objetos naturales, tales como las montañas y las nubes, tienen características irregulares o fragmentadas, y los métodos euclídeos no proporcionan representaciones realistas de tales objetos. Los objetos naturales se pueden describir de forma realista mediante **métodos de geometría fractal**, en donde se utilizan procedimientos en lugar de ecuaciones para modelar los objetos. Como es lógico pensar, los objetos definidos mediante procedimientos tienen características bastante diferentes de los objetos descritos con ecuaciones. Las representaciones de geometría fractal de objetos se aplican habitualmente en muchos campos para describir y explicar las características de fenómenos naturales. En gráficos por computadora, utilizamos los métodos fractales para generar visualizaciones de objetos naturales y de sistemas matemáticos y físicos.

Un objeto fractal posee dos características básicas: detalle infinito en cada punto, y una cierta *autosimilitud* entre las partes del objeto y las características totales del objeto. Las propiedades de autosimilitud de un objeto pueden presentar formas diferentes, dependiendo de la representación que elijamos del fractal. Describimos un objeto fractal mediante un procedimiento que especifica una operación repetida para producir el detalle en las subpartes del objeto. Los objetos naturales se representan con procedimientos que se repiten teóricamente un número infinito de veces. Las visualizaciones gráficas de objetos naturales se generan, por supuesto, mediante un número finito de pasos.



**FIGURA 8.67.** Apariencia escarpada del contorno de una montaña con diferentes niveles de ampliación.

Si ampliamos una forma continua euclídea, no importa cómo sea de complicada, podemos finalmente obtener la vista ampliada para verla con contornos más suaves. Pero si ampliamos un objeto fractal, continuamente viendo cada vez más detalles en las ampliaciones sin obtener finalmente una apariencia del objeto con contornos suaves. El contorno de una montaña frente al cielo continúa teniendo el mismo aspecto escarpado a medida que la vemos desde una posición cada vez más cercana (Figura 8.67). A medida que nos acercamos a la montaña, el detalle más pequeño de los salientes y de las rocas se hace visible. Acercándonos aún más, vemos los contornos de las rocas, después las piedras y a continuación los granos de arena. En cada paso, el contorno revela más curvas y vueltas. Si tomamos los granos de arena y los analizamos con un microscopio, veríamos de nuevo el mismo detalle repetido a nivel molecular. Formas similares describen las costas y los bordes de plantas y nubes.

Para obtener una vista ampliada de un fractal, podemos seleccionar una parte del fractal para su visualización dentro de un área de visualización del mismo tamaño. Entonces llevamos a cabo las operaciones de construcción del fractal en aquella parte del objeto y visualizamos el mayor detalle de aquel nivel de ampliación. A medida que repetimos este proceso, continuamos visualizando cada vez más detalles del objeto. Como consecuencia del detalle infinito inherente a los procedimientos de construcción, un objeto fractal no tiene un tamaño definido. Cuando añadimos más detalle a la descripción de un objeto, las dimensiones crecen sin límite, pero las amplitudes de las coordenadas del objeto permanecen confinadas dentro de una región finita del espacio.

Podemos caracterizar la cantidad de variación del detalle de un objeto mediante un número llamado *dimensión fractal*. A diferencia de la dimensión euclídea, este número no es necesariamente un entero. La dimensión fractal de un objeto se denomina a veces *dimensión fraccional*, que es la base del nombre «fractal».

Los métodos fractales han resultado ser útiles para modelar una gran variedad de fenómenos naturales. En aplicaciones gráficas, las representaciones fractales se utilizan para modelar el terreno, las nubes, el agua, los árboles y otra flora, las plumas, el pelo y variadas texturas superficiales, y a veces sólo para elaborar patrones atractivos. En otras disciplinas, los patrones fractales se han encontrado en la distribución de estrellas, islas de ríos y cráteres de la Luna; en la lluvia; en las variaciones de la bolsa; en música; en el tráfico; en la utilización de la propiedad urbana; y en los límites de las regiones de convergencia en técnicas de análisis numérico.

### Procedimientos para generación de fractales

Un objeto fractal se genera por aplicación repetida de una función de transformación específica a puntos dentro de una región del espacio. Si  $\mathbf{P}_0 = (x_0, y_0, z_0)$  se selecciona como posición inicial, cada iteración de una función de transformación  $F$  genera niveles sucesivos de detalle mediante los cálculos:

$$\mathbf{P}_1 = F(\mathbf{P}_0), \quad \mathbf{P}_2 = F(\mathbf{P}_1), \quad \mathbf{P}_3 = F(\mathbf{P}_2), \quad \dots \quad (8.105)$$

Por lo general, la función de transformación se puede aplicar a un conjunto de puntos específico o a un conjunto inicial de primitivas, tales como líneas rectas, curvas, áreas de color o superficies. También, pode-

mos utilizar procedimientos deterministas o aleatorios. La función de transformación se podría definir utilizando transformaciones geométricas (cambio de escala, traslación, rotación), o mediante transformaciones no lineales de coordenadas y parámetros de decisión estadísticos.

Aunque los objetos fractales, por definición, contienen detalles infinitos, aplicamos la función de transformación un número finito de veces, y, por supuesto, los objetos que visualizamos tienen dimensiones finitas. Una representación procedural aproxima un fractal «real» a medida que el número de transformaciones se incrementa para producir cada vez más detalle. La cantidad de detalle incluido en la visualización gráfica final de un objeto depende del número de iteraciones realizadas y la resolución del sistema de visualización. No podemos visualizar variaciones de detalle que sean menores que el tamaño de un píxel. Pero podemos ampliar de forma repetida porciones seleccionadas de un objeto para ver más de sus detalles.

## Clasificación de fractales

Los fractales **autosimilares** poseen partes que son versiones reducidas en tamaño del objeto entero. Comenzando por una forma inicial, construimos las subpartes del objeto aplicando un parámetro de escala  $s$  a toda la forma. Podemos utilizar el mismo factor de escala  $s$  para todas las subpartes, o podemos utilizar factores de escala diferentes en partes del objeto diferentes de escala reducida. Si también aplicamos variaciones aleatorias a las supartes de escala reducida, se dice que el fractal es *estadísticamente autosimilar*. Las partes entonces tienen las mismas propiedades estadísticas. Los fractales estadísticamente similares se utilizan habitualmente para modelar árboles, arbustos y otras plantas.

Los fractales **autoafines** poseen partes que se forman mediante parámetros de escala diferentes,  $s_x$ ,  $s_y$ , y  $s_z$ , según direcciones de coordenadas diferentes. Y podemos también incluir variaciones aleatorias para obtener fractales *autoafines estadísticamente*. El terreno, el agua y las nubes se modelan habitualmente mediante métodos de construcción de fractales autoafines estadísticamente.

Los **conjuntos invariantes de fractales** se forman mediante transformaciones no lineales. En esta clase de fractales se incluyen los fractales *autocuadráticos*, tales como el conjunto de Mandelbrot (formado con funciones cuadráticas en el espacio complejo), y los fractales *autoinversos*, construidos mediante procedimientos de inversión.

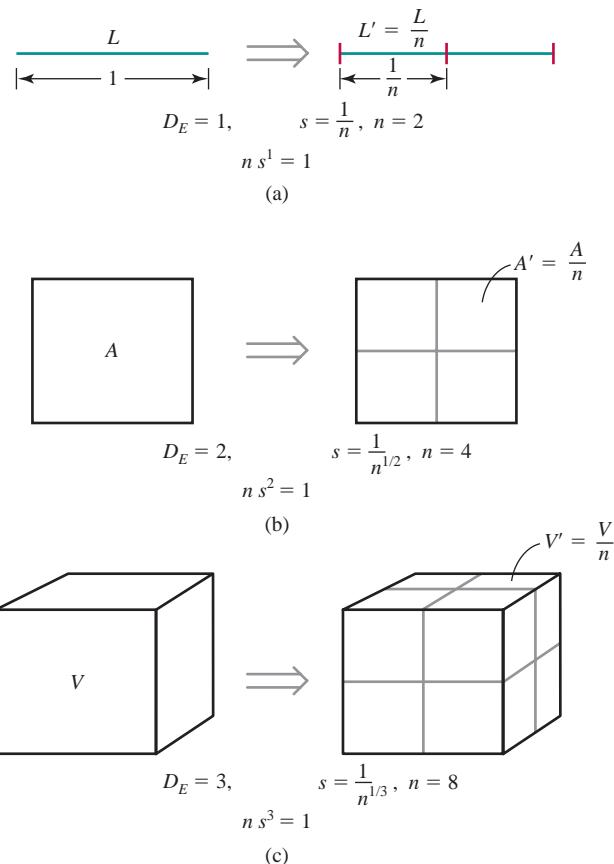
## Dimensión fractal

La cantidad de variación de la estructura de un objeto fractal se puede describir mediante un número  $D$ , llamado **dimensión fractal**, que es una medida de la aspereza, o de la fragmentación, del objeto. Los objetos con mayor aspecto dentado presentan mayores dimensiones fractales. Un método para generar un objeto fractal consiste en establecer un procedimiento iterativo que utiliza un valor seleccionado de  $D$ . Otra técnica es determinar la dimensión fractal a partir de las propiedades deseadas de un objeto, aunque, por lo general, puede ser difícil calcular la dimensión fractal. Los métodos para calcular  $D$  se basan en conceptos desarrollados en ramas de las matemáticas, particularmente de topología.

Mediante analogía con la subdivisión de un objeto euclídeo se obtiene una expresión de la dimensión fractal de un fractal autosimilar construido con un único factor de escala  $s$ . La Figura 8.68 muestra las relaciones entre el factor de escala y el número de subpartes  $n$  de división de un segmento unidad de línea recta, un cuadrado unidad y un cubo unidad. Para  $s = \frac{1}{2}$ , el segmento de línea unidad (Figura 8.68(a)) se divide en dos subpartes de igual longitud. Con el mismo factor de escala, el cuadrado de la Figura 8.68(b) se divide en cuatro subpartes de igual área, y el cubo (Figura 8.68(c)) se divide en ocho subpartes de igual volumen. En cada uno de estos objetos, la relación entre el número de subpartes y el factor de escala es  $n \cdot s^{D_E} = 1$ . En analogía con los objetos euclídeos, la dimensión fractal  $D$  de objetos autosimilares se puede obtener a partir de:

$$n s^D = 1 \quad (8.106)$$

Resolviendo esta expresión en  $D$ , la **dimensión de similitud fractal**, tenemos:



**FIGURA 8.68.** Subdivisión de una línea unidad (a), un cuadrado unidad (b), y un cubo unidad (c). La dimensión euclídea se representa como  $D_E$ , y el factor de escala de cada objeto es  $s = \frac{1}{2}$ .

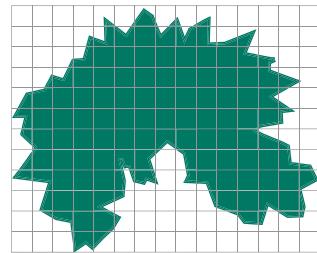
$$D = \frac{\ln n}{\ln(1/s)} \quad (8.107)$$

En un fractal autosimilar construido con diferentes factores de escala en las diferentes subpartes del objeto, la dimensión de similitud fractal se obtiene a partir de la relación implícita:

$$\sum_{k=1}^n s_k^D = 1 \quad (8.108)$$

donde  $s_k$  es el factor de escala de la subparte  $k$ .

En la Figura 8.68, consideramos la subdivisión de formas simples (línea recta, rectángulo y caja). Si tenemos formas más complejas, incluidas las líneas curvas y los objetos con superficies no planas, la determinación de la estructura y las propiedades de las subpartes es más difícil. Para formas de objetos generales, podemos utilizar *métodos de cobertura topológica* que aproximan las subpartes del objeto mediante formas simples. Una curva subdividida, por ejemplo, se podría aproximar mediante secciones de línea recta, y una superficie con *splines* subdividida se podría aproximar mediante pequeños cuadrados o rectángulos. Otras formas de cobertura, tales como círculos, esferas y cilindros, también se pueden utilizar para aproximar las características de un objeto dividido en partes más pequeñas. Los métodos de cobertura se utilizan habitualmente en matemáticas para determinar propiedades geométricas tales como la longitud, el área, o el volumen de un



**FIGURA 8.69.** Caja de cobertura de un objeto de forma irregular.

objeto complejo mediante la suma de las propiedades de un conjunto de objetos de cobertura más pequeños. También podemos utilizar los métodos de cobertura para determinar la dimensión fractal  $D$  de algunos objetos.

Los conceptos de cobertura topológica se utilizaron originariamente para ampliar el significado de las propiedades geométricas a formas no estándar. Una ampliación de los métodos de cobertura que utiliza círculos o esferas condujo a la idea de *dimensión de Hausdorff-Besicovitch*, o *dimensión fraccional*. La dimensión de Hausdorff-Besicovitch se puede utilizar como la dimensión fractal de algunos objetos, pero por lo general, es más difícil su evaluación. De forma más habitual, la dimensión fractal de un objeto se estima mediante *métodos de cobertura con cajas* que utilizan rectángulos o paralelepípedos. La Figura 8.69 ilustra la idea de caja de cobertura. Aquí, el área dentro de los límites irregulares se puede aproximar mediante la suma de las áreas de los pequeños rectángulos de cobertura.

Los métodos de cobertura mediante cajas se aplican determinando en primer lugar las amplitudes de las coordenadas de un objeto, después subdividiendo el objeto en un número de pequeñas cajas utilizando los factores de escala proporcionados. El número de cajas  $n$  que es necesario para cubrir un objeto se denomina *dimensión de caja*, y  $n$  está relacionada con la dimensión fractal  $D$ . En objetos autosimilares estadísticamente con un único factor de escala  $s$ , podemos cubrir el objeto con cuadrados o cubos. A continuación contamos el número  $n$  de cajas de cobertura y utilizamos la Ecuación 8.107 para estimar la dimensión fractal. Para objetos autoafines, cubrimos el objeto con cajas rectangulares, ya que las direcciones diferentes se cambian de escala de forma diferente. En este caso, estimamos la dimensión fractal utilizando tanto el número de cajas  $n$  como los parámetros de transformación afín.

La dimensión fractal de un objeto es siempre mayor que la dimensión euclídea correspondiente (o dimensión topológica), que es simplemente el menor número de parámetros necesarios para especificar el objeto. Una curva euclídea tiene una sola dimensión ya que podemos determinar sus puntos con un parámetro,  $u$ . Una superficie euclídea es bidimensional, con parámetros de superficie  $u$  y  $v$ . Y un sólido euclídeo, que requiere tres parámetros en cada especificación de coordenadas, es tridimensional.

En una curva fractal que se encuentra completamente dentro de un plano bidimensional, la dimensión fractal  $D$  es mayor que 1 (la dimensión euclídea de una curva). Cuanto más cercano es  $D$  a 1, más suave es la curva fractal. Si  $D = 2$ , tenemos una *curva de Peano*; es decir, la «curva» rellena completamente una región finita del espacio bidimensional. Para  $2 < D < 3$ , la curva se intersecta a sí misma y el área se podría cubrir con un número infinito de veces. Las curvas fractales se pueden utilizar para modelar los límites de objetos naturales, tales como las costas.

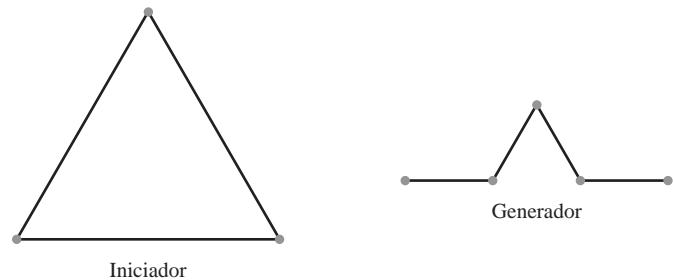
Las curvas fractales espaciales (aquellas que no se encuentran completamente dentro de un único plano) también tienen dimensión fractal  $D$  mayor que 1, pero  $D$  puede ser mayor que 2 sin intersección consigo misma. Una curva que rellena un volumen del espacio tiene dimensión  $D = 3$ , y una curva en el espacio que se intersecta a sí misma tiene una dimensión fractal  $D > 3$ .

Las superficies fractales tienen habitualmente una dimensión dentro del rango  $2 < D \leq 3$ . Si  $D = 3$ , la «superficie» rellena un volumen del espacio. Y si  $D > 3$ , hay una cobertura de superposición del volumen. El terreno, las nubes y el agua se modelan habitualmente con superficies fractales.

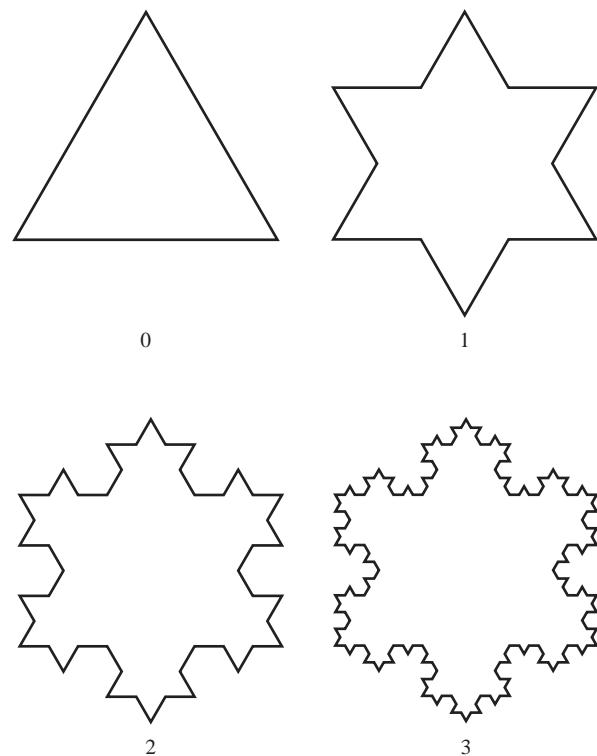
La dimensión de un fractal sólido se encuentra habitualmente en el rango  $3 < D \leq 4$ . De nuevo, si  $D > 4$ , tenemos un objeto que se autosuperpone. Los sólidos fractales se pueden utilizar, por ejemplo, para modelar las propiedades de las nubes como densidad de vapor de agua o la temperatura dentro de una región del espacio.

## Construcción geométrica de fractales deterministas autosimilares

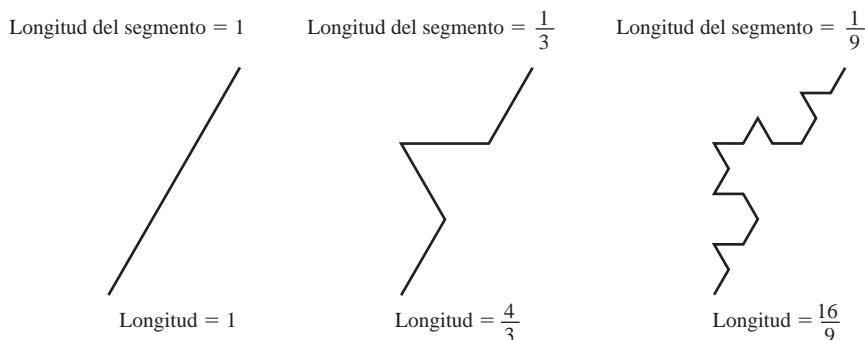
Para construir geométricamente un fractal determinista (no aleatorio) autosimilar, comenzamos por una forma geométrica dada, llamada *iniciador*. Las subpartes del iniciador se reemplazan a continuación por un patrón, llamado *generador*. A modo de ejemplo, si utilizamos el iniciador y el generador mostrados en la Figura 8.70, podemos construir el patrón de copo de nieve, o curva de Koch, mostrada en la Figura 8.71. Cada segmento de línea recta del iniciador se reemplaza por el patrón del generador, que consta de cuatro segmentos de línea de igual longitud. Después se cambia de escala el generador y se aplica a los segmentos de línea del iniciador modificado, y este proceso se repite un número de pasos. El factor de escala en cada paso es  $\frac{1}{3}$ , por lo que la dimensión fractal es  $D = \ln 4/\ln 3 \approx 1.2619$ . También, la longitud de cada segmento de línea del iniciador se incrementa en un factor  $\frac{4}{3}$  en cada paso, de modo que la longitud de la curva fractal tiende a infinito a medida que se añade más detalle a la curva (Figura 8.72). La Figura 8.73 ilustra los patrones adicionales de generador que se podrían utilizar para construir curvas fractales autosimilares. Los generadores de la Figura 8.73(b) y (c) contienen más detalle que el generador de la curva de Koch y tienen mayores dimensiones fractales.



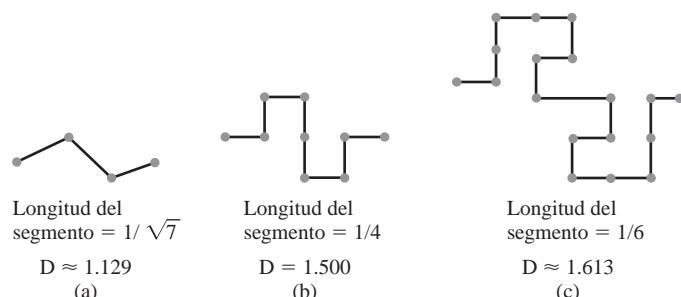
**FIGURA 8.70.** Iniciador y generador de la curva de Koch.



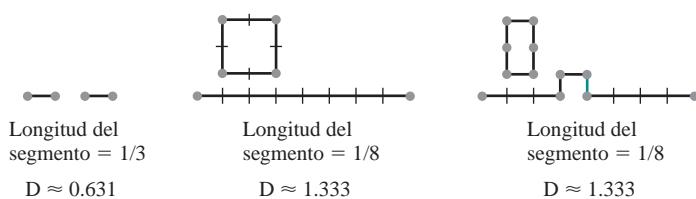
**FIGURA 8.71.** Tres primeras iteraciones de la generación de la curva de Koch.



**FIGURA 8.72.** La longitud de cada lado de la curva de Koch se incrementa en un factor de  $\frac{4}{3}$  en cada paso, mientras que las longitudes de los segmentos de línea se reducen en un factor de  $\frac{1}{3}$ .

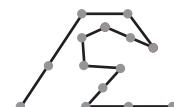


**FIGURA 8.73.** Generadores de curvas fractales autosimilares y sus dimensiones fractales asociadas.



**FIGURA 8.74.** Generadores de fractales con partes múltiples y disjuntas.

**FIGURA 8.75.** La aplicación de este generador a las aristas de un triángulo equilátero produce una curva de Peano de relleno con copos de nieve (también llamada espacio de Peano).

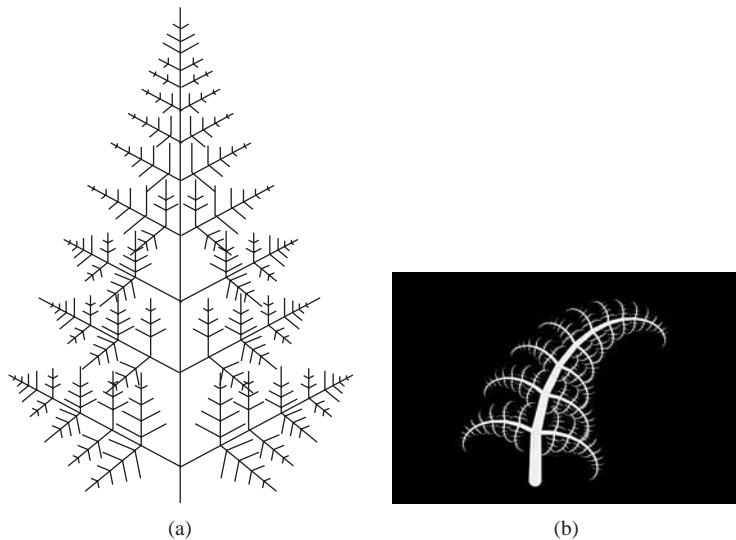


También podemos utilizar generadores con múltiples componentes disjuntas. En la Figura 8.74 se muestran algunos ejemplos de generadores compuestos. Podríamos combinar estos patrones con variaciones aleatorias para modelar varios objetos naturales que presentan múltiples partes desconectadas, tales como distribuciones de islas a lo largo de la costa.

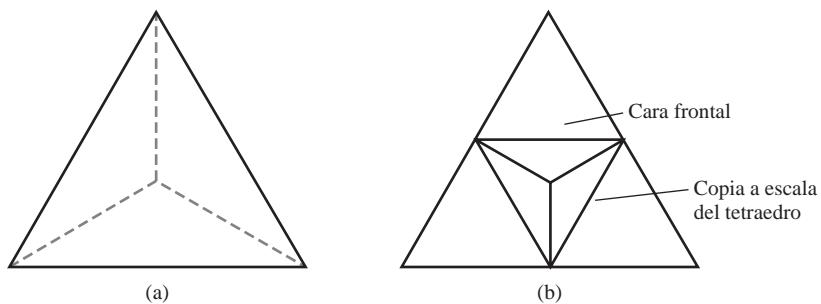
El generador de la Figura 8.75 contiene segmentos de línea de longitud variable y se utilizan múltiples factores de escala para la construcción de la curva fractal. Por tanto, la dimensión fractal de la curva generada se determina a partir de la Ecuación 8.108.

Las visualizaciones de árboles y otras plantas se pueden construir mediante métodos de construcción geométrica autosimilar. Cada rama del contorno del helecho mostrada en la Figura 8.76(a) es una versión cambiada de escala de la forma del helecho total. En la parte (b) de esta figura, el helecho se sombra totalmente con un giro aplicado a cada rama.

Como ejemplo de la construcción de un fractal autosimilar de las superficies de un objeto tridimensional, cambiamos de escala el tetraedro regular mostrado en la Figura 8.77 con un factor 1/2, después colocamos el



**FIGURA 8.76.** Construcciones autosimilares de un helecho. (Cortesía de Meter Oppenheimer, Computer Graphics Lab, New York Institute of Technology.)



**FIGURA 8.77.** Cambio de escala del tetraedro (a) en un factor de 1/2 y posicionando la versión con el cambio de escala aplicado en una cara del tetraedro original produce la superficie fractal mostrada en (b).

objeto que hemos cambiado de escala en cada una de las cuatro superficies originales del tetraedro. Cada cara del tetraedro original se convierte en seis caras más pequeñas y el área de la cara original se incrementa en un factor 3/2. La dimensión fractal de esta superficie es:

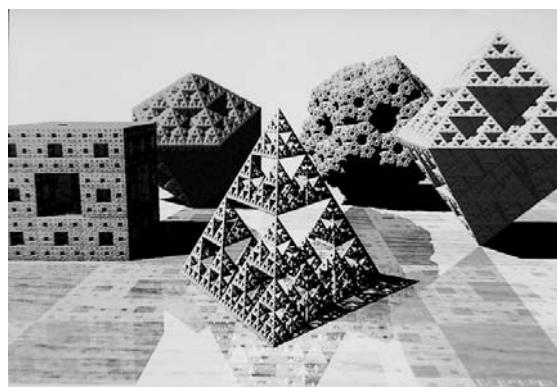
$$D = \frac{\ln 6}{\ln 2} \approx 2.58496$$

que indica que es una superficie bastante fragmentada.

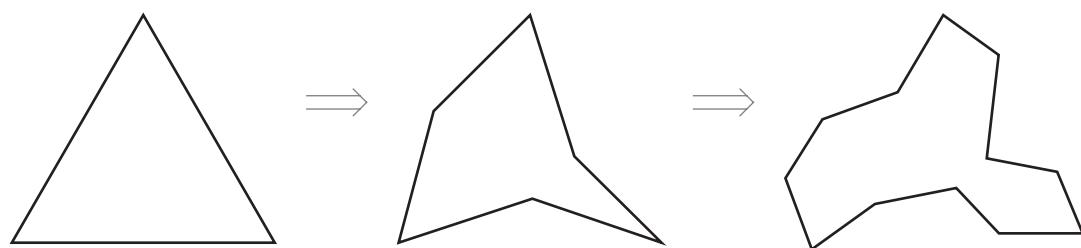
Otro modo de crear objetos fractales autosimilares consiste en perforar agujeros en un iniciador dado, en lugar de añadir más área. La Figura 8.78 muestra algunos ejemplos de objetos fractales creados de este modo.

### Construcción geométrica de fractales estadísticamente autosimilares

Para introducir variabilidad en la construcción geométrica de un fractal autosimilar, podríamos seleccionar de forma aleatoria un generador en cada paso a partir de un menú de patrones. O podríamos construir un fractal autosimilar calculando pequeños desplazamientos de las coordenadas mediante pequeñas variaciones aleato-



**FIGURA 8.78.** Fractales autosimilares tridimensionales formados con generadores que sustraen subpartes de un iniciador. (Cortesía de John C. Hart, Department of Computer Science, Universidad de Illinois en Urbana-Champaign.)



**FIGURA 8.79.** Un patrón modificado de «copo de nieve» que utiliza un desplazamiento aleatorio del punto medio.

rias. Por ejemplo, en la Figura 8.79 utilizamos una función de distribución de probabilidad para calcular desplazamientos variables del punto medio en cada paso de la creación de un patrón aleatorio de copo de nieve.

En la Figura 8.80 se muestra otro ejemplo de este método. En esta visualización se hace un cambio de escala aleatorio de parámetros y ramificación aleatoria de direcciones para modelar los patrones de las venas de una hoja.

Una vez que se ha creado un objeto fractal, podemos modelar una escena utilizando varias instancias transformadas del objeto. La Figura 8.81 muestra la instanciación de un árbol fractal con rotaciones aleatorias. En la Figura 8.82, se muestra un bosque fractal empleando varias transformaciones aleatorias.

Para modelar las formas nudosas y retorcidas de algunos árboles, podemos aplicar funciones de giro así como de cambio de escala para crear las ramas aleatorias y autosimilares. Esta técnica se ilustra en la Figura 8.83. Comenzando por el cilindro con tapas de la parte izquierda de esta figura, podemos aplicar transformaciones para producir (secuencialmente de izquierda a derecha) una espiral, una hélice y un patrón con giro aleatorio. En la Figura 8.84, se muestra un árbol modelado mediante giros aleatorios. La corteza de árbol de esta visualización se modela utilizando mapas de abultamiento (*bump mapping*) y variaciones fractales brownianas de los patrones de abultamiento. Los métodos de generación de curvas fractales brownianas se estudian en la siguiente sección, y los métodos de mapas de abultamiento se exploran en la Sección 10.17.



**FIGURA 8.80.** Construcción aleatoria y autosimilar de ramificación de venas de una hoja en otoño. (Cortesía de Peter Oppenheimer, Computer Graphics Lab, New York Institute of Technology.)



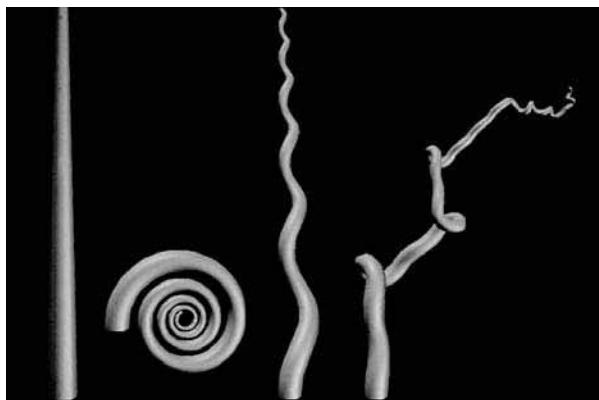
**FIGURA 8.81.** Modelado de una escena que emplea instanciación múltiple de objetos. Las hojas fractales se añaden a un árbol en posiciones transformadas de forma aleatoria, y varias instancias rotadas y con cambio de escala del árbol se utilizan para formar una arboleda. El césped se modela mediante instancias múltiples de conos verdes. (Cortesía de John C. Hart, Department of Computer Science, Universidad de Illinois en Urbana-Champaign.)



**FIGURA 8.82.** Un bosque fractal creado con instancias múltiples de hojas, agujas de pino, césped y corteza de árbol. (Cortesía de John C. Hart, Department of Computer Science, Universidad de Illinois en Urbana-Champaign.)

## Métodos de construcción de fractales afines

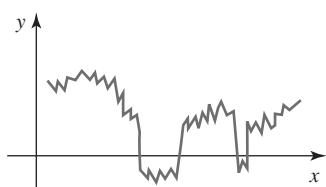
Podemos obtener representaciones altamente realistas del terreno y de otros objetos naturales utilizando métodos con fractales afines que modelan las características de los objetos como el *movimiento browniano fractal*. Este es una ampliación del movimiento browniano estándar, una forma de «paseo aleatorio», que describe el movimiento en zigzag y errático de partículas de gas u otro fluido. La Figura 8.85 ilustra la trayectoria de un paseo aleatorio en el plano  $xy$ . Comenzando por una posición dada, generamos un segmento de línea recta según una dirección aleatoria y con una longitud aleatoria. Otra línea aleatoria se construye a continuación desde el extremo de esta primera línea, y el proceso se repite un número concreto de segmentos de línea.



**FIGURA 8.83.** Modelado de ramas de árboles mediante giros en espiral, helicoidales y aleatorios. (Cortesía de Peter Oppenheimer, Computer Graphics Lab, New York Institute of Technology.)



**FIGURA 8.84.** Ramas de árbol modeladas con serpenteos aleatorios. (Cortesía de Peter Oppenheimer, Computer Graphics Lab, New York Institute of Technology.)



**FIGURA 8.85.** Un ejemplo de movimiento browniano (paseo aleatorio) en el plano  $xy$ .



**FIGURA 8.86.** Un planeta con movimiento browniano observado desde la superficie de un planeta con movimiento browniano, con cráteres añadidos, en primer plano. (Cortesía de R. V. Voss and B. B. Mandelbrot, adaptado a partir de *The Fractal Geometry of Nature* de Benoit B. Mandelbrot (W. H. Freeman and Co., Nueva York, 1983).)

El movimiento browniano fraccional se obtiene al añadir un parámetro adicional a la distribución estadística que describe el movimiento browniano. Este parámetro adicional modifica la dimensión fractal de la trayectoria del «movimiento».

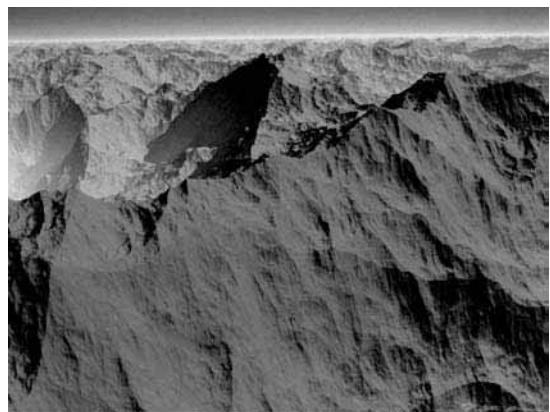
Una única trayectoria browniana fraccional se puede utilizar para modelar una curva fractal. Y con una matriz bidimensional de alturas brownianas fraccionales aleatorias sobre una cuadrícula de un plano de tierra, podemos modelar la superficie de una montaña uniendo las alturas para formar un conjunto de parches de polígonos. Si las alturas aleatorias se generan sobre la superficie de una esfera, podemos modelar las monta-

ñas, los valles, y los océanos de un planeta. En la Figura 8.86 el movimiento browniano se utilizó para crear las variaciones de altura sobre la superficie del planeta. Las alturas se codificaron en color de manera que las alturas más bajas se pintaron en azul (los océanos) y las alturas más elevadas en blanco (nieve sobre las montañas). El movimiento browniano fraccional se utilizó para crear las características del terreno en primer plano. Los cráteres se crearon con diámetros aleatorios y en posiciones aleatorias, utilizando procedimientos con fractales afines que describen fielmente la distribución de los cráteres observados, de las islas de los ríos, de los patrones de lluvia y de otros sistemas similares de objetos.

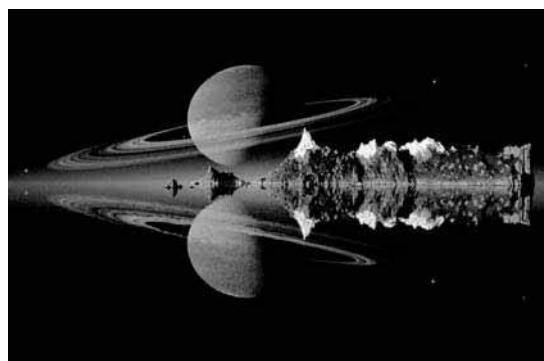
Al ajustar la dimensión fractal en los cálculos del movimiento browniano fraccional, podemos variar el escarpado de las características del terreno. Los valores de la dimensión fractal próximos a  $D \approx 2.15$  producen características realistas en las montañas, mientras que valores próximos a 3.0 se pueden utilizar para crear paisajes extraterrestres de apariencia inusual. También podemos cambiar de escala las alturas calculadas para hacer más profundos los valles e incrementar la altura de los picos de las montañas. En la Figura 8.87 se muestran algunos ejemplos de características del terreno que se pueden modelar mediante procedimientos fractales. En la Figura 8.88 se muestra una escena modelada con nubes fractales sobre una montaña fractal.



(a)



(b)

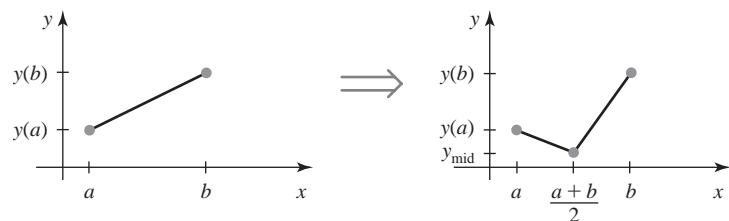


(c)

**FIGURA 8.87.** Variaciones de las características del terreno modeladas mediante movimiento browniano fraccional. (Cortesía de (a) R. V. Voss and B. B. Mandelbrot, adaptado a partir de *The Fractal Geometry of Nature* de Benoit B. Mandelbrot (W. H. Freeman and Co., New York, 1983); y (b) y (c) Ken Musgrave y Benoit B. Mandelbrot, *Mathematics and Computer Science*, Universidad de Yale)



**FIGURA 8.88.** Una escena modelada mediante nubes fractales y montañas fractales. (Cortesía de Ken Musgrave y Benoit B. Mandelbrot, Mathematics and Computer Science, Universidad de Yale.)



**FIGURA 8.89.** Desplazamiento aleatorio del punto medio de un segmento de línea recta.

### Métodos de desplazamiento aleatorio del punto medio

Los cálculos del movimiento browniano fraccional requieren tiempo, ya que las coordenadas de la elevación del terreno sobre un plano de tierra se calculan mediante series de Fourier, que son sumas de términos con senos y cosenos. Los métodos que utilizan la transformada rápida de Fourier (Fast Fourier Transform; FFT) se utilizan habitualmente, pero son aún un proceso lento para generar escenas con montañas fractales. Por tanto, *métodos de desplazamiento aleatorio del punto medio* más rápidos, similares a los métodos de desplazamiento utilizados en las construcciones geométricas, se han desarrollado para aproximar las representaciones con movimiento browniano aleatorio del terreno y otros fenómenos naturales. Estos métodos se utilizaron originariamente para generar cuadros de animación en películas de ciencia ficción que involucraban características inusuales de terrenos y planetas. Los métodos de desplazamiento del punto medio ahora se utilizan habitualmente en muchas otras aplicaciones de gráficos por computadora, entre las que se incluyen las animaciones para anuncios de televisión.

Aunque los métodos de desplazamiento aleatorio del punto medio son más rápidos que los cálculos del movimiento browniano fraccional, producen características de terrenos de apariencia menos realista. La Figura 8.89 ilustra el método del desplazamiento del punto medio para generar una trayectoria de un paseo aleatorio en el plano  $xy$ . Comenzando por un segmento de línea recta, calculamos un valor de la ordenada  $y$  desplazado de la posición del punto medio de la línea como el valor medio de los valores de la ordenada  $y$  del extremo más un desplazamiento aleatorio:

$$y_{\text{mid}} = \frac{1}{2}[y(a) + y(b)] + r \quad (8.109)$$

Para aproximar el movimiento browniano fraccional, seleccionamos un valor para  $r$  de la distribución gaussiana de media 0 y una varianza proporcional a  $|(b - a)|^{2H}$ , donde  $H = 2 - D$  y  $D > 1$  es la dimensión fractal. Otro método para obtener un desplazamiento aleatorio consiste en tomar  $r = sr_g|b - a|$ , donde el pará-

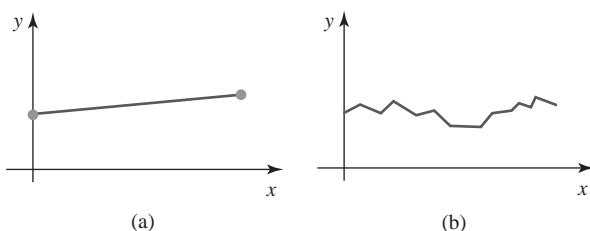
metro  $s$  es un factor seleccionado de «rugosidad» de la superficie y  $r_g$  es un valor aleatorio gaussiano de media 0 y varianza 1. Se pueden utilizar tablas de búsqueda para obtener los valores gaussianos. El proceso después se repite calculando un valor desplazado de la ordenada  $y$  para la posición media de cada mitad de la línea subdividida. Y continuamos la subdivisión para obtener un cierto número de segmentos o hasta que las longitudes de las secciones de la línea subdividida son menores que una longitud seleccionada. En cada paso, el valor de la variable aleatoria  $r$  disminuye, ya que es proporcional al ancho  $|b - a|$  de la sección de la línea que hay que subdividir. La Figura 8.90 muestra una curva fractal obtenida con este método.

Las características del terreno se generan mediante la aplicación de los procedimientos de desplazamiento aleatorio del punto medio a un plano rectangular de tierra (Figura 8.91). Comenzamos asignando un valor de altura  $z$  a cada una de las cuatro esquinas (**a**, **b**, **c**, **d** de la Figura 8.91) del plano de tierra. A continuación, dividimos el plano de tierra por el punto medio de cada arista para obtener los nuevos cinco puntos de la cuadrícula: **e**, **f**, **g**, **h** y **m**. Las alturas en los puntos medios **e**, **f**, **g** y **h** de las aristas del plano de tierra se pueden calcular como la media de la altura de los dos vértices más cercanos más un desplazamiento aleatorio. Por ejemplo, la altura  $z_e$  en el punto **e** se calcula utilizando los vértices **a** y **b**, mientras que la altura en el punto medio **f** se calcula utilizando los vértices **b** y **c**:

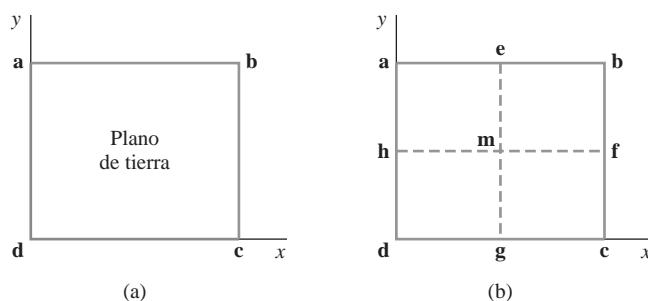
$$z_e = (z_a + z_b)/2 + r_e, \quad z_f = (z_b + z_c)/2 + r_f$$

Los valores aleatorios  $r_e$  y  $r_f$  se pueden obtener a partir de una distribución gaussiana de media 0 y varianza proporcional a la separación de la cuadrícula elevada a la potencia  $2H$ , donde  $H = 3 - D$  y  $D > 2$ . Valores más elevados de  $D$ , dimensión fractal de la superficie, producen un terreno con más dientes de sierra, mientras que valores más bajos generan un terreno más suave. También podríamos calcular los desplazamientos aleatorios como el producto de un factor de rugosidad por la separación de la cuadrícula por un valor de la tabla de búsqueda de un valor gaussiano de media 0 y varianza 1. La altura  $z_m$  de la posición media del plano de tierra **m** se puede calcular utilizando los puntos **e** y **g**, o los puntos **f** y **h**. De forma alternativa, podríamos calcular  $z_m$  utilizando las alturas asignadas a las cuatro esquinas del plano de tierra y un desplazamiento aleatorio del siguiente modo:

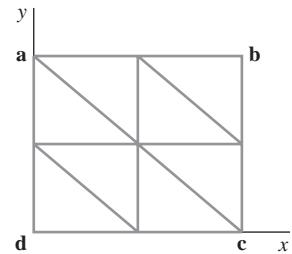
$$z_m = (z_a + z_b + z_c + z_d)/4 + r_m$$



**FIGURA 8.90.** Una trayectoria de un paseo aleatorio generada a partir de un segmento de línea recta con cuatro iteraciones del procedimiento de desplazamiento aleatorio del punto medio.



**FIGURA 8.91.** Un plano rectangular de tierra (a) se divide en una cuadrícula de cuatro secciones iguales (b) en el primer paso de un procedimiento de desplazamiento aleatorio del punto medio para calcular las alturas del terreno.



**FIGURA 8.92.** Ocho parches de superficie formados sobre un plano de tierra en el primer paso de un procedimiento de desplazamiento aleatorio del punto medio para la generación de características del terreno.

Este proceso se repite para cada una de las nuevas cuatro partes de la cuadrícula en cada paso, hasta que la separación de la cuadrícula llega a ser más pequeña que un valor seleccionado.

Los parches triangulares de superficie para la superficie del terreno se pueden formar a medida que las alturas se generan. La Figura 8.92 muestra ocho parches de superficie que se podrían construir en el primer paso de la subdivisión. En cada nivel de recursión, los triángulos se dividen sucesivamente en parches planos más pequeños. Cuando el proceso de subdivisión se ha completado, los parches se sombrean utilizando las posiciones seleccionadas de las fuentes de luz, los valores de otros parámetros de iluminación, y los colores y texturas seleccionados de la superficie del terreno.

El método del desplazamiento aleatorio del punto medio se puede aplicar para generar otros componentes de una escena además del terreno. Por ejemplo, podríamos utilizar los mismos métodos para obtener características superficiales de ondas en el agua o patrones de nubes sobre un plano de tierra.

### Control de la topografía del terreno

Un modo de controlar la colocación de los picos y de los valles en una escena con terreno fractal que se modela con un método de desplazamiento de punto medio, consiste en restringir las alturas calculadas a ciertos intervalos sobre varias secciones del plano de tierra. Podemos realizar esto designando un conjunto de *superficies de control* sobre el plano de tierra, como se ilustra en la Figura 8.93. Después calculamos una altura aleatoria en cada punto medio de la cuadrícula del plano de tierra que depende de la diferencia entre la altura de control y la altura media calculada para dicho punto. Este procedimiento obliga a las alturas a pertenecer a un intervalo preseleccionado en torno a las alturas de la superficie de control.

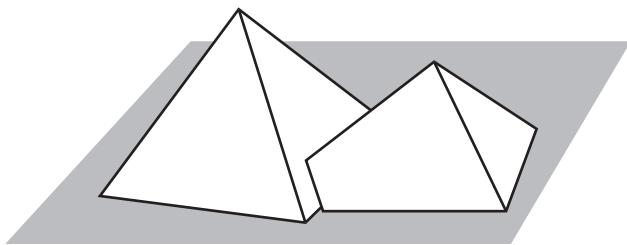
Las superficies de control se pueden utilizar para modelar características existentes del terreno en las Montañas Rocosas, o en alguna otra región, construyendo las facetas del plano utilizando las alturas de un plano topográfico de una región concreta. O podríamos establecer las alturas de los vértices de los polígonos de control para diseñar nuestras propias características del terreno. También, las superficies de control pueden tener cualquier forma. Los planos son los más fáciles de utilizar, pero podríamos utilizar superficies esféricas u otras formas curvas.

Utilizamos el método de desplazamiento aleatorio del punto medio para calcular las alturas de la cuadrícula, pero ahora seleccionamos valores aleatorios a partir de una distribución gaussiana en la que la media  $\mu$  y la desviación estándar  $\sigma$  son funciones de las alturas de control. Un método para obtener los valores  $\mu$  y  $\sigma$  consiste en hacer ambos proporcionales a la diferencia entre la altura media calculada y la altura de control predefinida en cada punto de la rejilla. Por ejemplo, para el punto de la rejilla e de la Figura 8.91, establecemos la media y la desviación estándar del siguiente modo:

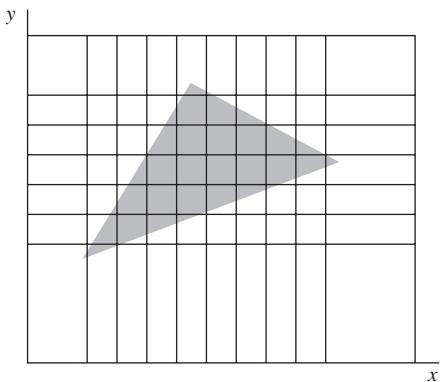
$$\mu_e = zc_e - (z_a + z_b)/2, \sigma_e = s|\mu_e|$$

donde  $zc_e$  es la altura de control del punto e del plano de tierra y  $0 < s < 1$  es el factor de escala. Los valores pequeños de  $s$ , tales como  $s < 0.1$ , producen una concordancia más ajustada a la envolvente del terreno, y valores más grandes de  $s$  permiten mayores fluctuaciones de la altura del terreno.

Para determinar los valores de las alturas de control sobre una superficie de control del plano, en primer lugar, determinamos los valores de los parámetros del plano  $A, B, C$  y  $D$ . En cualquier punto del plano de tierra ( $x, y$ ), la altura en el plano que contiene aquel polígono de control se calcula entonces del siguiente modo:



**FIGURA 8.93.** Superficies de control sobre un plano de tierra.



**FIGURA 8.94.** Proyección de una superficie triangular de control sobre una cuadrícula del plano de tierra.

$$zc = (-Ax - By - D)/C$$

Se pueden utilizar métodos incrementales para calcular las alturas de control sobre los puntos de la cuadrícula del plano de control. Para llevar a cabo estos cálculos eficientemente, subdividimos el plano de tierra con una cuadrícula más pequeña de puntos  $xy$  y proyectamos cada polígono de la superficie de control sobre el plano de tierra, como se muestra en la Figura 8.94. A partir de esta proyección, determinamos los puntos de la cuadrícula que se encuentran por debajo de cada polígono de control. Esto se puede realizar utilizando procedimientos similares a los empleados en el relleno de áreas mediante líneas de barrido. Es decir, para cada «línea de barrido»  $y$  y de la malla del plano de tierra que cruza las aristas del polígono, calculamos las intersecciones de la línea de barrido y determinamos qué puntos de la cuadrícula están dentro de la proyección del polígono de control. Los cálculos de las alturas de control en estos puntos de la cuadrícula se realizan de forma incremental de este modo:

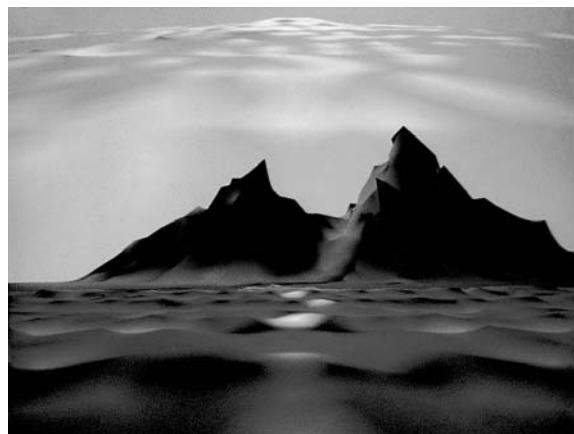
$$zc_{i+1,j} = zc_{i,j} - \Delta x(A/C), \quad zc_{i,j+1} = zc_{i,j} - \Delta y(B/C), \quad (8.110)$$

donde  $\Delta x$  y  $\Delta y$  son los incrementos de la cuadrícula en las direcciones de los ejes  $x$  e  $y$ . Este procedimiento es particularmente rápido cuando se aplican métodos de vectores paralelos para procesar los puntos de la cuadrícula del plano de control.

La Figura 8.95 muestra una escena construida utilizando planos de control para estructurar las superficies del terreno, del agua y las nubes sobre un plano de tierra. A continuación, se aplican algoritmos de sombreado de superficies para suavizar las aristas de los polígonos y proporcionar los colores apropiados a la superficie.

## Fractales autocuadráticos

Otro método para generar objetos fractales consiste en aplicar repetidamente una función de transformación a los puntos del plano complejo. En dos dimensiones, un número complejo se puede representar como  $z = x + iy$ , donde  $x$  e  $y$  son números reales e  $i^2 = -1$ . En el espacio tridimensional y de cuatro dimensiones, los



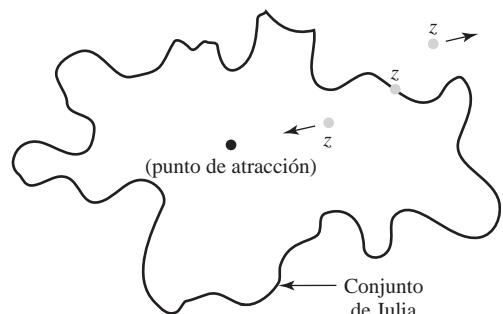
**FIGURA 8.95.** Una escena compuesta modelada mediante un método de desplazamiento aleatorio del punto medio y superficies planas de control sobre un plano de tierra. Las características de la superficie del terreno, del agua y las nubes se modelaron y sombrearon de forma separada, después se combinaron para formar la imagen compuesta. (Cortesía de Eng-Kiat Koh, Encentuate, Inc., Cupertino, California.)

puntos se representan mediante cuaternios. Una función cuadrática compleja  $f(z)$  es aquella que implica el cálculo de  $z^2$ . Podemos utilizar algunas funciones autocuadráticas para generar formas fractales.

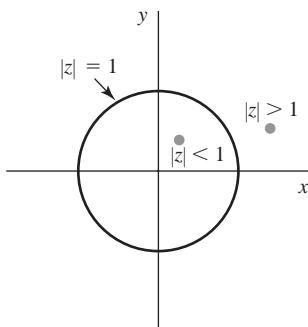
Dependiendo del punto inicial seleccionado para iterar, la aplicación repetida de una función autocuadrática producirá uno de los tres resultados posibles:

- El punto transformado puede diverger hacia infinito.
- El punto transformado puede converger hacia un punto límite finito, llamado punto de atracción (atractor).
- El punto transformado permanece en el límite de alguna región.

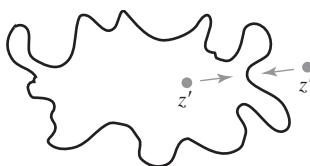
Como ejemplo, la operación cuadrática no fractal  $f(z) = z^2$  en el plano complejo transforma los puntos según su relación con el círculo unidad (Figura 8.97). Cualquier punto  $z$  cuyo módulo  $|z|$  sea mayor que 1 se transforma mediante una secuencia de puntos que tienden hacia infinito. Un punto con  $|z| < 1$  se transforma hacia el origen de coordenadas. Los puntos que están originariamente en el círculo,  $|z| = 1$ , permanecen en el mismo. Aunque la transformación  $z^2$  no produce un fractal, algunas operaciones cuadráticas complejas generan una curva fractal que es la zona límite entre aquellos puntos que se mueven hacia el infinito y aquellos que tienden hacia un límite finito. Un límite cerrado fractal generado mediante una operación cuadrática se denomina *conjunto de Julia*.



**FIGURA 8.96.** Posibles resultados de la aplicación repetida de una transformación autocuadrática  $f(z)$  al plano complejo, dependiendo de la posición del punto inicial seleccionado.



**FIGURA 8.97.** Un círculo unidad en el plano complejo. La función compleja cuadrática no fractal  $f(z) = z^2$  mueve los puntos que se encuentran en el interior del círculo hacia el origen, mientras que los puntos situados fuera del círculo se mueven más lejos del círculo. Cualquier punto inicial del círculo permanece en el círculo.



**FIGURA 8.98.** Localización de la curva frontera fractal utilizando la función auto-cuadrática inversa  $z = f^{-1}(z')$ .

Por lo general, podemos localizar el límite fractal de una función cuadrática comprobando el comportamiento de los puntos seleccionados. Si un punto se transforma de modo que diverge hacia infinito o converge hacia un punto de atracción, podemos probar con otro punto próximo. Repetimos este proceso hasta que finalmente localizamos un punto del límite fractal. A continuación, la iteración de la transformación cuadrática genera la forma del fractal. En transformaciones simples del plano complejo, un método más rápido para localizar los puntos de la curva fractal consiste en utilizar la inversa de la función de transformación. Entonces un punto inicial seleccionado dentro o fuera de la curva convergerá a un punto de la curva fractal (Figura 8.98).

Una función rica en fractales es la transformación cuadrática:

$$z' = f(z) = \lambda z(1 - z) \quad (8.111)$$

donde  $\lambda$  es una constante compleja. En esta función, podemos utilizar el método de la inversa para localizar la curva fractal. En primer lugar reorganizamos los términos para obtener la ecuación cuadrática:

$$z^2 - z + z'/\lambda = 0 \quad (8.112)$$

La transformación inversa es entonces la fórmula cuadrática:

$$z = f^{-1}(z') = \frac{1}{2} \left( 1 \pm \sqrt{\frac{|discr| + \operatorname{Re}(discr)}{2}} \right) \quad (8.113)$$

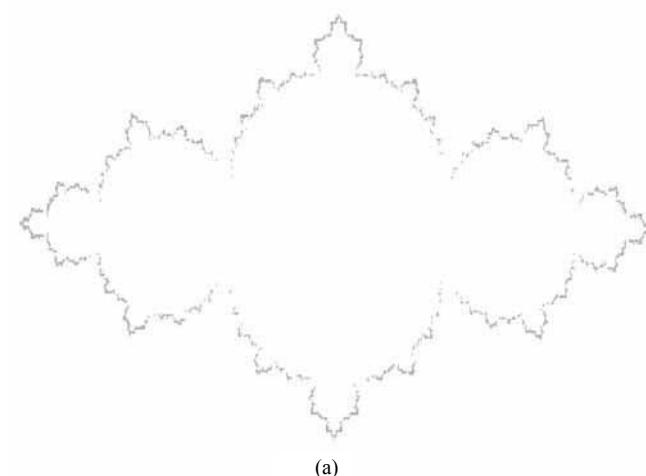
Utilizando operaciones aritméticas complejas, resolvemos esta ecuación en las partes real e imaginaria de  $z$  del siguiente modo:

$$x = \operatorname{Re}(z) = \frac{1}{2} \left( 1 \pm \sqrt{\frac{|discr| + \operatorname{Re}(discr)}{2}} \right) \quad (8.114)$$

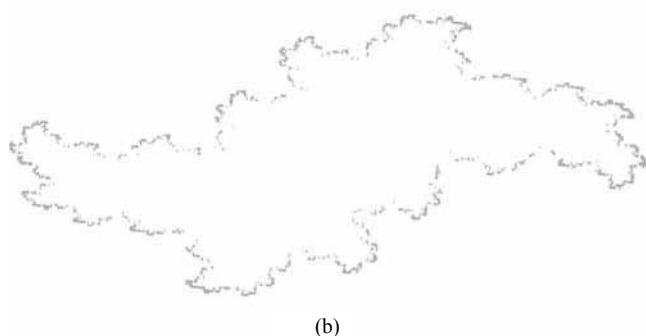
$$y = \operatorname{Im}(z) = \pm \frac{1}{2} \sqrt{\frac{|discr| - \operatorname{Re}(discr)}{2}}$$

donde el discriminante de la fórmula cuadrática es  $discr = 1 - 4z'/\lambda$ . Se puede calcular e ignorar unos pocos valores iniciales de  $x$  e  $y$  (por ejemplo, 10) antes de que empecemos a dibujar la curva fractal. También, ya

que esta función produce dos posibles puntos transformados  $(x, y)$ , podemos elegir de forma aleatoria el signo más o menos en cada paso de la iteración siempre y cuando  $\text{Im}(\text{discr}) \geq 0$ . Cada vez que  $\text{Im}(\text{discr}) < 0$ , los dos posibles puntos están en el segundo y tercer cuadrante. En este caso,  $x$  e  $y$  deben tener signos opuestos. El programa siguiente proporciona una implementación de esta función autocuadrática. En la Figura 8.99 se dibujan dos curvas de ejemplo.



(a)



(b)

**FIGURA 8.99.** Dos curvas fractales generadas con la inversa de la función  $f(z) = \lambda z(1 - z)$  mediante el procedimiento `selfSqTransf`, utilizando (a)  $\lambda = 3$  y (b)  $\lambda = 2 + i$ . Cada curva se ha dibujado con 10000 puntos.

```
#include <GL/glut.h>
#include <stdlib.h>
#include <math.h>

/* Establece el tamaño inicial de la ventana de visualización. */
GLsizei winWidth = 600, winHeight = 600;

/* Establece los límites de las coordenadas del plano complejo. */
GLfloat xComplexMin = -0.25, xComplexMax = 1.25;
GLfloat yComplexMin = -0.75, yComplexMax = 0.75;

struct complexNum
{
    GLfloat x, y;
};
```

```

void init (void)
{
    /* Establece el color de la ventana de visualización en blanco. */
    glClearColor (1.0, 1.0, 1.0, 0.0);
}

void plotPoint (complexNum z)
{
    glBegin (GL_POINTS);
        glVertex2f (z.x, z.y);
    glEnd ( );
}

void solveQuadraticEq (complexNum lambda, complexNum * z)
{
    GLfloat lambdaMagSq, discrMag;
    complexNum discr;
    static complexNum fourOverLambda = { 0.0, 0.0 };
    static GLboolean firstPoint = true;

    if (firstPoint) {
        /* Calcula el número complejo: 4.0 dividido por lambda. */
        lambdaMagSq = lambda.x * lambda.x + lambda.y * lambda.y;
        fourOverLambda.x = 4.0 * lambda.x / lambdaMagSq;
        fourOverLambda.y = -4.0 * lambda.y / lambdaMagSq;
        firstPoint = false;
    }
    discr.x = 1.0 - (z->x * fourOverLambda.x - z->y * fourOverLambda.y);
    discr.y = {-x * fourOverLambda.y + z->y * fourOverLambda.x};
    discrMag = sqrt (discr.x * discr.x + discr.y * discr.y);

    /* Actualiza z, comprobando para evitar la raíz cuadrada de un número
     * negativo. */
    if (discrMag + discr.x < 0)
        z->x = 0;
    else
        z->x = sqrt ((discrMag + discr.x) / 2.0);
    if (discrMag - discr.x < 0)
        z->y = 0;
    else
        z->y = 0.5 * sqrt ((discrMag - discr.x) / 2.0);

    /* Para la mitad de los puntos, utiliza la raíz negativa,
     * situando el punto en el cuadrante 3.
     */
    if (rand () < RAND_MAX / 2) {
        z->x = -z->x;
        z->y = -z->y;
    }
}

```

```
/* Cuando la parte imaginaria del discriminante es negativa, el punto
 * debería estar en el cuadrante 2 o 4, para invertir el signo de x.
 */
if (discr.y < 0)
    z->x = -z->x;

/* Completa el cálculo de la parte real de z. */
z->x = 0.5 * (1 - z->x);
}

void selfSqTransf (complexNum lambda, complexNum z, GLint numPoints)
{
    GLint k;

    /* Salta los primeros puntos. */
    for (k = 0; k < 10; k++)
        solveQuadraticEq (lambda, &z);

    /* Dibuja el número específico de puntos de transformación. */
    for (k = 0; k < numPoints; k++) {
        solveQuadraticEq (lambda, &z);
        plotPoint (z);
    }
}

void displayFcn (void)
{
    GLint numPoints = 10000; // Establece el número de puntos que hay que dibujar.
    complexNum lambda = { 3.0, 0.0 }; // Establece el valor complejo de lambda.
    complexNum z0 = { 1.5, 0.4 }; // Establece el punto incial del plano
                                 // complejo. */

    glClear (GL_COLOR_BUFFER_BIT); // Borra la ventana de visualización.

    glColor3f (0.0, 0.0, 1.0); // Establece el color de los puntos en azul.

    selfSqTransf (lambda, z0, numPoints);
    glFlush ( );
}

void winReshapeFcn (GLint newWidth, GLint newHeight)
{
    /* Mantiene una relación de aspecto de 1.0, asumiendo que
     * el ancho de la ventana compleja = altura de la ventana compleja.
     */
    glViewport (0, 0, newHeight, newHeight);

    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ( );
```

```

gluOrtho2D (xComplexMin, xComplexMax, yComplexMin, yComplexMax);

glClear (GL_COLOR_BUFFER_BIT);
}

void main (int argc, char** argv)
{
    glutInit (&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    glutInitWindowPosition (50, 50);
    glutInitWindowSize (winWidth, winHeight);
    glutCreateWindow ("Self-Squasing Fractal");

    init ();
    glutDisplayGunc (displayFcn);
    glutReshapeFunc (winReshapeFcn);

    glutMainLoop ();
}

```

En la Figura 8.100 se proporciona un gráfico tridimensional de las variables  $x$ ,  $y$  y  $\lambda$  correspondiente a la función autocuadrática  $f(z) = \lambda z(1 - z)$ , donde  $|\lambda| = 1$ . Cada sección recta de este gráfico es una curva fractal en el plano complejo.

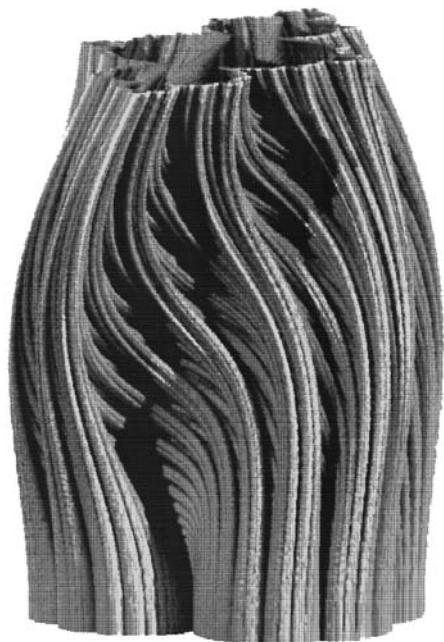
Otra operación cuadrática que produce una gran variedad de formas fractales es una transformación  $z^2$  ligeramente modificada. En este caso, el fractal es el límite de la región alrededor del conjunto de valores complejos  $z$  que no divergen frente a la transformación cuadrática:

$$\begin{aligned} z_0 &= z \\ z_k &= z_{k-1}^2 + z_0 \quad k = 1, 2, 3, \dots \end{aligned} \tag{8.115}$$

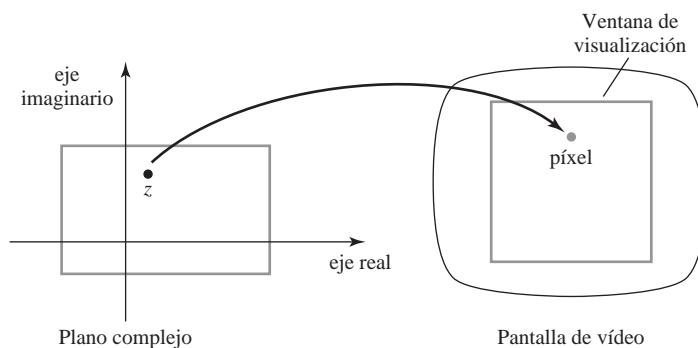
Por tanto, en primer lugar seleccionamos un punto  $z$  del plano complejo, después calculamos el punto transformado  $z^2 + z$ . En el paso siguiente, calculamos el cuadrado de este punto transformado y se lo añadimos al valor original de  $z$ . Repetimos este procedimiento hasta que podamos determinar si la transformación es divergente o no.

Los matemáticos han sido conscientes de las características inusuales de tales funciones cuadráticas durante algún tiempo, pero estas funciones eran difíciles de analizar sin computadoras. Después del desarrollo de la computadora digital, el límite de convergencia de la transformación 8.115 se dibujó con una impresora de líneas. A medida que las capacidades de las computadoras digitales se incrementaron, fue posible una investigación gráfica más profunda. Posteriormente, utilizando técnicas de gráficos por computadora más sofisticadas, Benoit Mandelbrot estudió ampliamente esta función, y el conjunto de puntos que no divergen frente a la transformación 8.115 se conoce como el **conjunto de Mandelbrot**.

Para implementar la transformación 8.115, en primer lugar seleccionamos un área rectangular del plano complejo. Los puntos de este área se mapean a continuación a píxeles codificados en color dentro de una ventana de visualización de un monitor de video (Figura 8.101). Los colores de los píxeles se eligen según la velocidad de divergencia del punto correspondiente del plano complejo frente a la transformación 8.115. Si el módulo del número complejo es mayor que 2, entonces divergirá rápidamente cuando se calcula repetidamente su cuadrado. Por tanto, podemos establecer un bucle para repetir las operaciones de cálculo del cuadrado



**FIGURA 8.100.** La función  $f(z) = \lambda z(1 - z)$  dibujada en tres dimensiones, con valores normalizados de  $\lambda$  que varían según el eje vertical. (Cortesía de Alan Norton, IBM Research.)



**FIGURA 8.101.** Mapeo de puntos desde un área rectangular del plano complejo a píxeles codificados con color dentro de una ventana de visualización.

hasta que el módulo del número complejo sea mayor que 2 o hayamos alcanzado un número preseleccionado de iteraciones. El número máximo de iteraciones depende de la cantidad de detalle que queramos para visualizar y del número de puntos que haya que dibujar. Este valor se establece a menudo en algún valor entre 100 y 1000, aunque se pueden utilizar valores más bajos para acelerar los cálculos. Con valores más bajos del límite de iteración, sin embargo, tendremos a perder cierto detalle a lo largo de los límites (conjunto de Julia) de la región de convergencia. Al final del bucle, seleccionamos un valor de color según el número de iteraciones ejecutadas en el bucle. Por ejemplo, podemos colorear de negro el píxel si el número de iteraciones alcanza el valor máximo (un punto no divergente), podemos colorear el píxel de rojo si el número de iteraciones es próximo a 0. Se pueden seleccionar otros valores de color según el valor del número de iteraciones dentro del intervalo que varía desde 0 al valor máximo. eligiendo diferentes mapeos de color y diferentes partes del plano complejo, podemos generar una gran variedad de visualizaciones dramáticas de los puntos de la vecin-

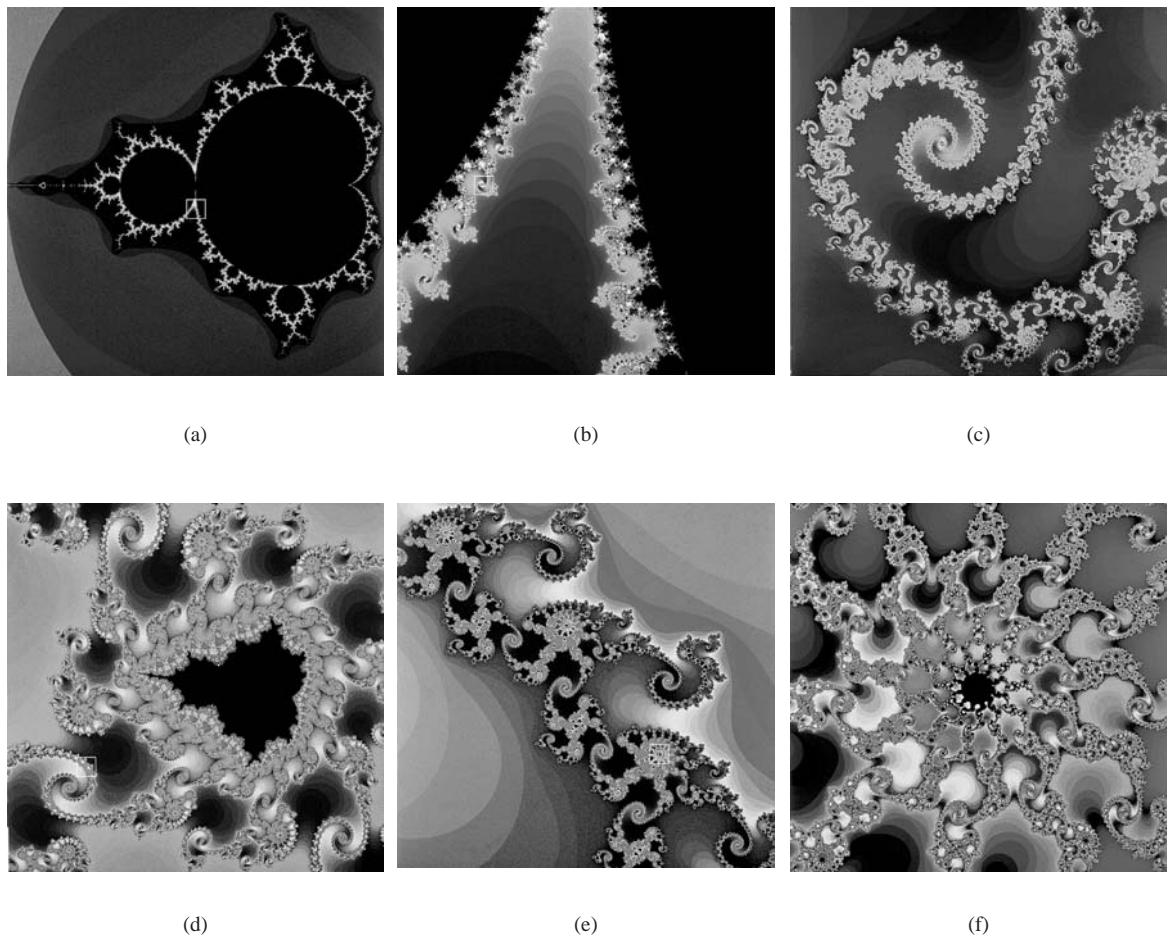
dad de la frontera fractal que encierra los puntos no divergentes. En la Figura 8.102(a) se muestra una elección para codificar con color los píxeles de la región alrededor del conjunto de Mandelbrot.

En el programa siguiente se proporciona una implementación de la transformación 8.115 para visualizar el conjunto de puntos de convergencia y sus fronteras. La parte principal del conjunto de convergencia está contenido dentro de la siguiente región del plano complejo.

$$-2.00 \leq \operatorname{Re}(z) \leq 0.50$$

$$-1.20 \leq \operatorname{Im}(z) \leq 1.20$$

Podemos explorar los detalles a lo largo de la frontera del conjunto de Mandelbrot seleccionando regiones rectangulares del plano complejo sucesivamente más pequeñas de modo que podamos ampliar áreas seleccionadas de la visualización. La Figura 8.102 muestra una visualización codificada en color (aunque la figura se muestra en blanco y negro) de la región alrededor del conjunto de convergencia y una serie de ampliaciones que ilustran algunas características notables de esta transformación cuadrática.



**FIGURA 8.102.** Ampliación de las fronteras fractales de la transformación 8.115. Comenzando por una visualización del conjunto de Mandelbrot, la región negra de (a) y sus áreas circundantes, ampliamos regiones seleccionadas de la frontera desde (b) hasta (f). El contorno blanco de la caja muestra el área rectangular seleccionada en cada ampliación sucesiva. En cada paso se eligen diferentes combinaciones de color para mejorar los patrones visualizados. (Cortesía de Brian Evans, Vanderbilt University.).

```
#include <GL/glut.h>

/* Establece el tamaño inicial de la ventana de visualización. */
GLsizei winWidth = 500, winHeight = 500;

/* Establece los límites del área rectangular del plano complejo. */
GLfloat xComplexMin = -2.00, xComplexMax = 0.50;
GLfloat yComplexMin = -1.25, yComplexMax = 1.25;

GLfloat complexWidth = xComplexMax - xComplexMin;
GLfloat complexHeight = yComplexMax - yComplexMin;

class complexNum {
public:
    GLfloat x, y;
};

struct color { GLfloat r, g, b; };

void init (void)
{
    /* Establece el color de la ventana de visualización en blanco. */
    glClearColor (1.0, 1.0, 1.0, 0.0);
}

void plotPoint (complexNum z)
{
    glBegin (GL_POINTS);
        glVertex2f (z.x, z.y);
    glEnd ( );
}

/* Calcula el cuadrado de un número complejo. */
complexNum complexSquare (complexNum z)
{
    complexNum zSquare;

    zSquare.x = z.x * z.x - z.y * z.y;
    zSquare.y = 2 * z.x * z.y;
    return zSquare;
}

GLint mandelSqTransf (complexNum z0, GLint maxIter)
{
    complexNum z = z0;
    GLint count = 0;

    /* Sale cuando z * z > 4 */
    while ((z.x * z.x + z.y * z.y <= 4.0) && (count < maxIter)) {
```

```

        z = complexSquare (z);
        z.x += z0.x;
        z.y += z0.y;
        count++;
    }
    return count;
}

void mandelbrot (GLint nx, GLint ny, GLint maxIter)
{
    complexNum z, zIncr;
    color ptColor;

    GLint iterCount;

    zIncr.x = complexWidth / GLfloat (nx);
    zIncr.y = complexHeight / GLfloat (ny);

    for (z.x = xComplexMin; z.x < xComplexMax; z.x += zIncr.x)
        for (z.y = yComplexMin; z.y < yComplexMax; z.y += zIncr.y) {
            iterCount = mandelSqTransf (z, maxIter);
            if (iterCount >= maxIter)
                /* Establece el color de los puntos en negro. */
                ptColor.r = ptColor.g = ptColor.b = 0.0;
            else if (iterCount > (maxIter / 8)) {
                /* Establece el color de los puntos en naranja. */
                ptColor.r = 1.0;
                ptColor.g = 0.5;
                ptColor.b = 0.0;
            }
            else if (iterCount > (maxIter / 10)) {
                /* Establece el color de los puntos en rojo. */
                ptColor.r = 1.0;
                ptColor.g = ptColor.b = 0.0;
            }
            else if (iterCount > (maxIter / 20)) {
                /* Establece el color de los puntos en azul
                   oscuro. */
                ptColor.b = 0.5;
                ptColor.r = ptColor.g = 0.0;
            }
            else if (iterCount > (maxIter / 40)) {
                /* Establece el color de los puntos en
                   amarillo. */
                ptColor.r = ptColor.g = 1.0;
                ptColor.b = 0.0;
            }
        }
    else if (iterCount > (maxIter / 100)) {

```

```
        /* Establece el color de los puntos en
         * verde oscuro. */
        ptColor.r = ptColor.b = 0.0;
        ptColor.g = 0.3;
    }

    else {
        /* Establece el color de los
         * puntos en cian. */
        ptColor.r = 0.0;
        ptColor.g = ptColor.b = 1.0;
    }

    /* Dibuja el punto coloreado. */
    glColor3f (ptColor.r, ptColor.g, ptColor.b);
    plotPoint (z);
}

}

void displayFcn (void)
{
    /* Establece el número de subdivisiones en los ejes x e y y las iteraciones
     * máximas. */
    GLint nx = 1000, ny = 1000, maxIter = 1000;

    glClear (GL_COLOR_BUFFER_BIT); // Borra la pantalla de visualización.

    mandelbrot (nx, ny, maxIter);
    glFlush ( );
}

void winReshapeFcn (GLint newWidth, GLint newHeight)
{
    /* Mantiene la relación de aspecto en 1.0, asumiendo que
     * complexWidth = complexHeight.
     */
    glViewport (0, 0, newHeight, newHeight);

    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ( );

    gluOrtho2D (xComplexMin, xComplexMax, yComplexMin, yComplexMax);

    glClear (GL_COLOR_BUFFER_BIT);
}

void main (int argc, char** argv)
{
    glutInit (&argc, argv);
```

```

glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB) ;
glutInitWindowPosition (50, 50);
glutInitWindowSize (winWidth, winHeight);
glutCreateWindow ("Mandelbrot Set");

init ();
glutDisplayFunc (displayFcn);
glutReshapeFunc (winReshapeFcn);

glutMainLoop ( );
}

```

Las transformaciones con funciones complejas, tales como la Ecuación 8.111, se pueden ampliar para producir superficies fractales y sólidos fractales. Los métodos para generar estos objetos utilizan representaciones con *cuaternios* (Apéndice A) para transformar puntos del espacio tridimensional y de cuatro dimensiones. Un cuaternion posee cuatro componentes, un número real y tres números imaginarios. Podemos representar un cuaternion de la siguiente forma, como ampliación del concepto de número del plano complejo,

$$q = s + ia + jb + kc \quad (8.116)$$

donde  $i^2 = j^2 = k^2 = -1$ . El número real  $s$  se denomina también *parte escalar* del cuaternion, y los números imaginarios se llaman *parte vectorial* del cuaternion  $\mathbf{v} = (a, b, c)$ .

Utilizando las reglas de la multiplicación y suma de cuaternios estudiadas en el Apéndice A, podemos aplicar las funciones autocuadráticas y otros métodos de iteración para generar superficies de objetos fractales. Un procedimiento básico consiste en comprobar puntos del plano complejo hasta que podamos identificar la frontera entre los puntos divergentes y los no divergentes. Por ejemplo, si localizamos en primer lugar un punto (interior) no divergente, entonces comprobamos los puntos vecinos respecto de dicho punto hasta que se identifique un punto (exterior) divergente. El punto interior anterior se guarda como un punto de la superficie frontera. A continuación se comprueban los vecinos de este punto de la superficie para determinar si están dentro (convergen) o si están fuera (divergen). Cualquier punto interior conectado con un punto exterior es un punto de la superficie. De este modo, el procedimiento se autodirige a lo largo de la frontera fractal sin desviarse su rumbo lejos de la superficie. Cuando se generan fractales de cuatro dimensiones, los cortes tridimensionales se proyectan sobre la superficie bidimensional del monitor de vídeo.

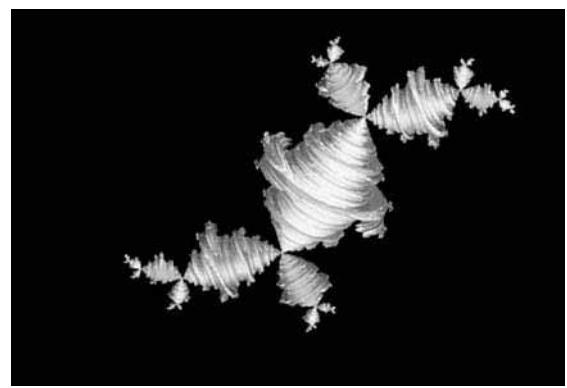
Los procedimientos para generar fractales autocuadráticos en el espacio de cuatro dimensiones requieren un tiempo considerable de cálculo, para evaluar la función de iteración y comprobar la convergencia o divergencia de los puntos. Cada punto sobre una superficie se puede representar como un cubo pequeño, que proporciona los límites interno y externo de la superficie. La salida de tales programas para proyecciones tridimensionales contienen más de un millón de vértices en los cubos de la superficie. Visualizamos el objeto fractal mediante la aplicación de modelos de iluminación para determinar el color de cada cubo de la superficie. También se aplican métodos de detección de la superficie visible de modo que sólo se muestren las superficies visibles del objeto. Las Figuras 8.103 y 8.104 muestran ejemplos de fractales autocuadráticos de cuatro dimensiones mediante proyecciones en tres dimensiones.

## Fractales autoinversos

Se pueden utilizar varias transformaciones geométricas de inversión para crear formas fractales. De nuevo, comenzamos por un conjunto inicial de puntos, y aplicamos repetidamente operaciones no lineales de inversión para transformar los puntos iniciales en un fractal.

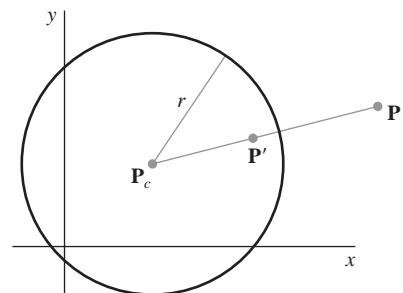
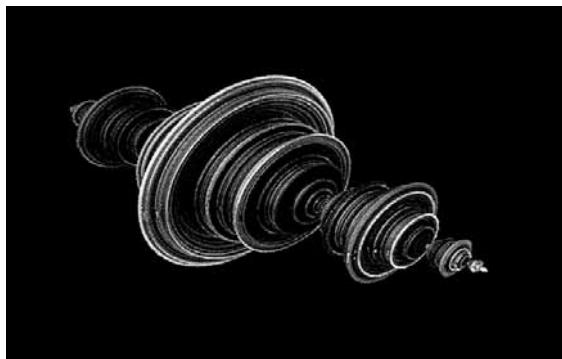


(a)



(b)

**Figura 8.103.** Proyecciones tridimensionales de fractales de cuatro dimensiones generados mediante la función autocuadrática representada con cuaternios  $f(q) = \lambda q(1 - q)$ , que utilizan (a)  $\lambda = 1.475 + 0.9061i$  y (b)  $\lambda = -0.57 + i$ . (Cortesía de Alan Norton, IBM Research.)



**FIGURA 8.104.** Una proyección sobre una superficie tridimensional de un objeto de cuatro dimensiones generado mediante una función autocuadrática representada con cuaternios  $f(q) = q^2 - 1$ . (Cortesía de Alan Norton, IBM Research.)

**FIGURA 8.105.** Inversión del punto  $\mathbf{P}$  al punto  $\mathbf{P}'$  situado dentro de un círculo de radio  $r$ .

Como ejemplo, consideremos una transformación bidimensional de inversión con respecto a un círculo de radio  $r$  y su centro  $\mathbf{P}_c = (x_c, y_c)$ . Un punto  $\mathbf{P}$  situado fuera del círculo se invierte en un punto  $\mathbf{P}'$  situado dentro del círculo (Figura 8.105) mediante la transformación:

$$(\overline{\mathbf{P}_c \mathbf{P}})(\overline{\mathbf{P}_c \mathbf{P}'}) = r^2 \quad (8.117)$$

donde ambos puntos  $\mathbf{P}$  y  $\mathbf{P}'$  se encuentran situados en una línea recta que pasa por el centro del círculo  $\mathbf{P}_c$ . Podemos usar también la Ecuación 8.117 para transformar los puntos que se encuentran dentro del círculo. Algunos puntos situados dentro se transforman en puntos situados fuera, mientras que otros puntos situados dentro se transforman en puntos situados dentro.

Si las coordenadas de los dos puntos se representan como  $\mathbf{P} = (x, y)$  y  $\mathbf{P}' = (x', y')$ , podemos escribir la Ecuación 8.117 del siguiente modo:

$$[(x - x_c)^2 + (y - y_c)^2]^{1/2}[(x' - x_c)^2 + (y' - y_c)^2]^{1/2} = r^2$$

También, ya que los dos puntos están situados en una línea que pasa a través del centro del círculo, tenemos que  $(y - y_c)/(x - x_c) = (y' - y_c)/(x' - x_c)$ . Por tanto, los valores de las coordenadas transformadas del punto  $P'$  son:

$$x' = x_c + \frac{r^2(x - x_c)}{(x - x_c)^2 + (y - y_c)^2}, \quad y' = y_c + \frac{r^2(y - y_c)}{(x - x_c)^2 + (y - y_c)^2} \quad (8.118)$$

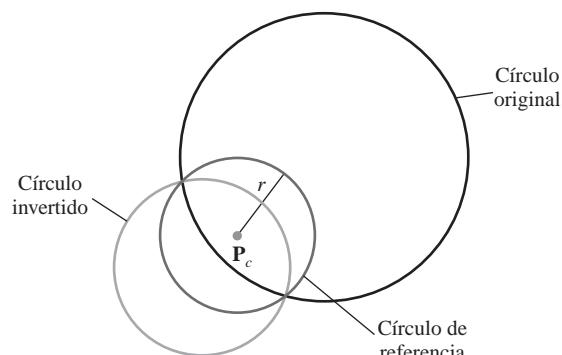
Por tanto, los puntos situados fuera del círculo se mapean a puntos situados dentro de la circunferencia del círculo; los puntos distantes ( $\pm\infty$ ) se transforman en el centro del círculo. A la inversa, los puntos cercanos al centro del círculo se mapean a puntos distantes situados fuera del círculo. A medida que nos alejamos del centro del círculo, los puntos se mapean a puntos más cercanos a la circunferencia del círculo, situados fuera del mismo. Y los puntos situados dentro cerca de la circunferencia se transforman en puntos situados dentro cerca del centro del círculo. Por ejemplo, los valores de la coordenada  $x$  de fuera, dentro del rango que varía desde  $r$  a  $+\infty$ , se mapean a valores  $x'$  dentro del rango que varía desde  $r/2$  a 0, para un círculo centrado en el origen, y los valores de  $x$  de dentro desde  $r/2$  hasta  $r$  se transforman en valores dentro del rango que varía desde  $r$  hasta  $r/2$ . Se obtienen resultados similares para los valores negativos de  $x$ .

Podemos aplicar esta transformación a varios objetos, tales como líneas rectas, círculos o elipses. Una línea recta que pasa por el centro del círculo es invariante frente a esta transformación de inversión; se mapea en sí misma. Pero una línea recta que no pasa por el centro del círculo se invierte en un círculo cuya circunferencia contiene el centro  $P_c$ . Y cualquier círculo que pase por el centro del círculo de referencia se invierte en una línea recta que no pasa por el centro del círculo. Si el círculo no pasa por el centro del círculo de referencia, se invierte en otro círculo, como en la Figura 8.106. Otro invariante frente a la inversión es la transformación de un círculo que es ortogonal al círculo de referencia. Es decir, las tangentes de los dos círculos son perpendiculares en los puntos de intersección.

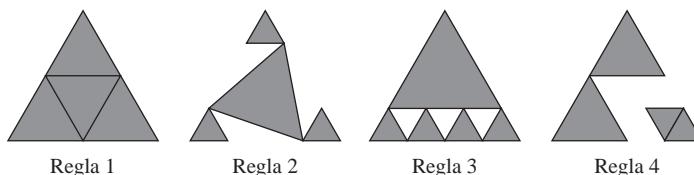
Podemos crear varias formas fractales mediante esta transformación de inversión comenzando por un conjunto de círculos y aplicando la transformación utilizando diferentes círculos de referencia. De forma similar, podemos aplicar la inversión con círculos a un conjunto de líneas rectas. Se pueden desarrollar métodos de inversión similares para otras formas bidimensionales. Y, podemos generalizar el procedimiento a esferas, planos u otros objetos tridimensionales.

## 8.24 GRAMÁTICAS DE FORMAS Y OTROS MÉTODOS PROCEDIMENTALES

Se puede utilizar un gran número de otros métodos procedimentales para diseñar formas de objetos o niveles de detalle de la superficie. Las **gramáticas de formas** son conjuntos de reglas de producción que se pueden aplicar a un objeto inicial, para añadir capas de detalle que son armoniosas con la forma original. Las transformaciones se pueden aplicar para alterar la geometría (forma) del objeto, o las reglas de transformación se pueden aplicar para añadir detalles del color o la textura de la superficie.



**FIGURA 8.106.** Inversión de un círculo que no pasa por el origen del círculo de referencia.

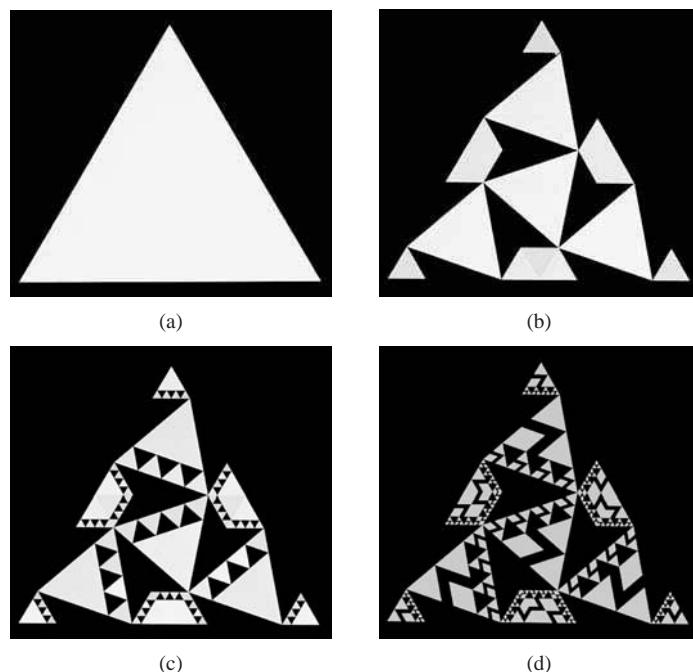


**Figura 8.107.** Cuatro reglas geométricas de sustitución para subdividir y alterar la forma de un triángulo equilátero.

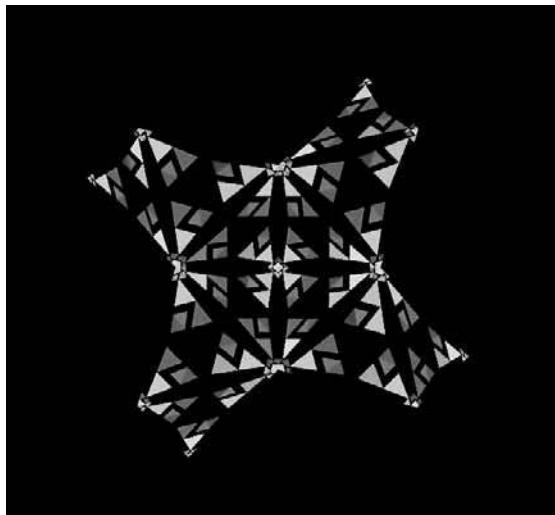
Dado un conjunto de reglas de producción, un diseñador de formas puede experimentar aplicando reglas diferentes en cada paso de la transformación a partir de un objeto inicial dado hasta la estructura final. La Figura 8.107 muestra cuatro reglas geométricas de sustitución para alterar formas de triángulos. Las transformaciones de la geometría de estas reglas se pueden expresar algorítmicamente en el sistema, basándose en una imagen de entrada dibujada con un editor de reglas de producción. Es decir, cada regla se puede describir gráficamente mostrando las formas inicial y final. Se pueden establecer implementaciones con Mathematica o algunos lenguajes de programación con capacidades gráficas.

En la Figura 8.108 se proporciona una aplicación de las sustituciones geométricas de la Figura 8.107, en donde la Figura 8.108(d) se obtiene mediante la aplicación de las cuatro reglas sucesivamente, comenzando por el triángulo inicial de la Figura 8.108(a). La Figura 8.109 muestra otras formas creadas mediante reglas de sustitución de triángulos.

Las formas tridimensionales y las características de la superficie se transforman con operaciones similares. La Figura 8.110 muestra los resultados de sustituciones geométricas aplicadas a poliedros. La forma inicial de los objetos mostrados en la Figura 8.111 es un icosaedro (un poliedro de 20 caras). Las sustituciones geométricas se aplicaron a la caras planas del icosaedro, y los vértices del polígono que resultan se proyectaron sobre la superficie de una esfera circunscrita.



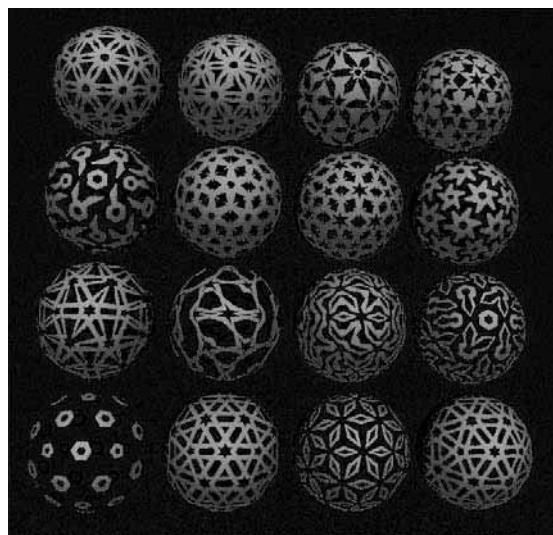
**FIGURA 8.108.** Un triángulo equilátero (a) se convierte en la forma mostrada en (b) utilizando las reglas de sustitución 1 y 2 de la Figura 8.107. La regla 3 se utiliza a continuación para convertir (b) en la forma (c), la que a su vez se transforma en (d) utilizando la regla 4. (Cortesía de Andrew Glassner, Xerox PARC (Palo Alto Research Center). © 1992.)



**FIGURA 8.109.** Un diseño creado mediante reglas geométricas de sustitución para alterar formas de triángulos. [Cortesía de Andrew Glassner, Xerox PARC (Palo Alto Research Center). © 1992.]



**FIGURA 8.110.** Un diseño creado mediante reglas geométricas de sustitución para alterar formas prismáticas. La forma inicial de este diseño es una representación de la Serpiente de Rubik. [Cortesía de Andrew Glassner, Xerox PARC (Palo Alto Research Center). © 1992.]



**FIGURA 8.111.** Diseños creados sobre la superficie de una esfera utilizando reglas de sustitución de triángulos aplicadas a las caras planas de un icosaedro, seguidas de proyecciones sobre la superficie de la esfera. [Cortesía de Andrew Glassner, Xerox PARC (Palo Alto Research Center). © 1992.]

Otro conjunto de reglas de producción para describir la forma de los objetos se llaman *gramáticas L*, o *graftales*. Estas reglas se utilizan habitualmente para generar visualizaciones de plantas. Por ejemplo, la topología de un árbol se puede describir como un tronco, al que se unen algunas ramas y hojas. Un árbol se puede entonces modelar mediante reglas para proporcionar una conexión concreta de las ramas y de las hojas en las ramas individuales. La descripción geométrica se proporciona a continuación colocando las estructuras del objeto en puntos concretos.

La Figura 8.112 muestra una escena que contiene varias plantas y árboles, construidos con un paquete comercial generador de plantas. Los procedimientos del software aplican leyes botánicas para generar las formas de las plantas y de los árboles.

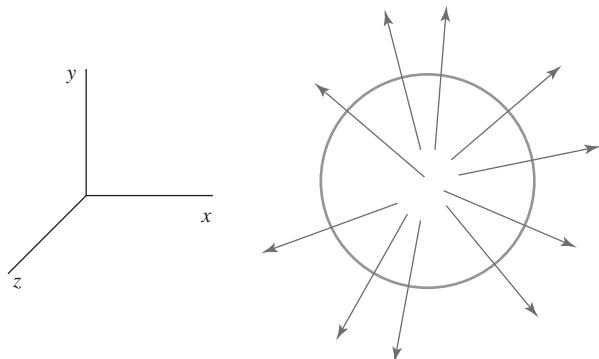
## 8.25 SISTEMAS DE PARTÍCULAS

---

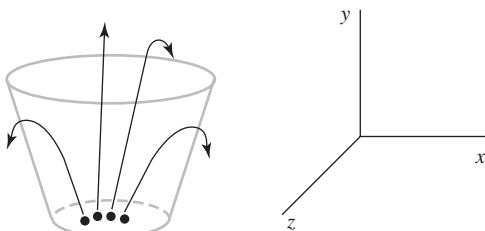
Para algunas aplicaciones, a menudo es útil describir uno o más objetos empleando una colección de partes disjuntas, llamada **sistema de partículas**. Esta técnica se puede aplicar para describir objetos con propiedades semejantes a las de los fluidos que pueden cambiar en el tiempo mediante flujo, ondulación, pulverización, expansión, contracción o explosión. Entre los objetos con estas características se incluyen las nubes, el humo, el fuego, los fuegos artificiales, las cascadas de agua y el agua pulverizada. Los sistemas de partículas se han empleado, por ejemplo, para modelar la explosión de un planeta y la expansión del frente de fuego debidos a la «bomba del génesis» de la película *Star Trek II: La Ira de Khan*. Y los métodos basados en sistemas de partículas se han utilizado para modelar otras clases de objetos, entre las que se incluye la hierba.



**FIGURA 8.112.** Escenario realista generado con el paquete de software TDI-AMAP, que puede generar cerca de cien variedades de plantas y árboles utilizando procedimientos basados en leyes de la Botánica. (Cortesía de Thomson Digital Image.)



**FIGURA 8.113.** Modelado de fuegos artificiales como un sistema de partículas que viajan radialmente hacia fuera desde el centro de una esfera.



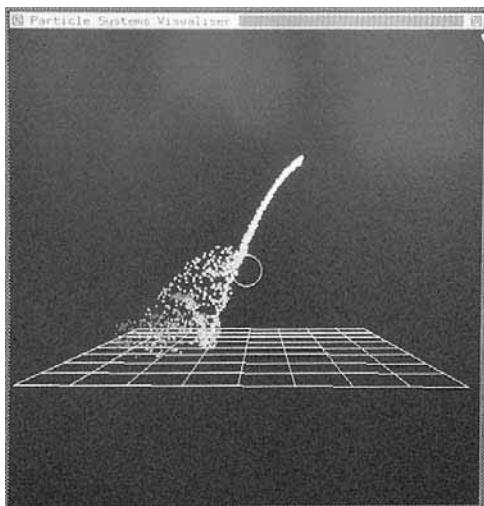
**FIGURA 8.114.** Modelado de hierba mediante el lanzamiento de partículas hacia arriba desde dentro de un cilindro con tapas. Las trayectorias de las partículas son parábolas debido a la fuerza hacia abajo de la gravedad.

En una aplicación típica, un sistema de partículas se define dentro de alguna región del espacio y a continuación se aplican procesos aleatorios para variar en el tiempo los parámetros del sistema. Entre estos parámetros se incluye la trayectoria del movimiento de las partículas individuales, su color y su forma. En algún momento aleatorio, cada partícula se borra.

Las formas de las partículas se podrían describir mediante pequeñas esferas, elipsoides o cajas, que pueden variar de forma aleatoria en el tiempo. También, se elige aleatoriamente la transparencia de las partículas, el color y el movimiento. Las trayectorias del movimiento de las partículas se podrían describir cinemáticamente o definir mediante fuerzas tales como un campo gravitatorio.

A medida que cada partícula se mueve, se dibuja su trayectoria y se visualiza con un color particular. Por ejemplo, un patrón de fuegos artificiales se puede visualizar mediante la generación aleatoria de partículas dentro de una región esférica del espacio y permitiendo que éstas se muevan radialmente hacia fuera, como en la Figura 8.113. Las trayectorias de las partículas se podrían codificar con color desde el rojo hasta el amarillo, por ejemplo, para simular la temperatura de las partículas que explotan. De forma similar, las visualizaciones realistas de la hierba se han modelado mediante partículas «de trayectoria» (Figura 8.114) que surgen del suelo y que vuelven a la tierra bajo la acción de la gravedad. En este caso, las trayectorias de las partículas se pueden originar dentro de un cilindro con tapas y se podrían codificar en color desde el verde hasta el amarillo.

La Figura 8.115 ilustra una simulación de un sistema de partículas de una cascada de agua. Las partículas de agua caen desde una altura fija, se desvían mediante un obstáculo y entonces rebotan en el suelo. Los diferentes colores se utilizan para distinguir las trayectorias de las partículas en cada etapa. En la Figura 8.116 se muestra un ejemplo de una animación que simula la desintegración de un objeto. El objeto de la izquierda se desintegra en la distribución de partículas de la derecha. En la Figura 8.117 se proporciona una escena compuesta formada por una variedad de representaciones. La escena se modeló utilizando hierba con sistema de partículas y montañas fractales, además de mapeado de texturas y otros procedimientos de sombreado de superficies.



**FIGURA 8.115.** Simulación del comportamiento de una cascada que golpea una piedra (círculo). Las partículas de agua se desvían mediante la roca y a continuación rebotan en el suelo. (Cortesía de M. Brooks and T. L. J. Howard, Department of Computer Science, Universidad de Manchester.)



**FIGURA 8.116.** Un objeto que se desintegra en una nube de partículas. (Cortesía de Autodesk, Inc.)

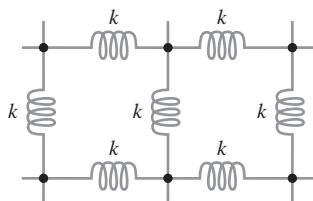
## 8.26 MODELADO BASADO EN LAS CARACTERÍSTICAS FÍSICAS

Un objeto no rígido, tal como una cuerda, una tela o una pelota de goma, se puede representar mediante métodos de *modelado basado en las características físicas* que describen el comportamiento del objeto en función de la interacción de las fuerzas externas y las fuerzas internas. Una descripción precisa de la forma de una toalla de felpa colocada sobre el respaldo de una silla, por ejemplo, se obtiene al considerar el efecto de la silla sobre las ondas de la tela de la toalla y la interacción entre los hilos de ésta.

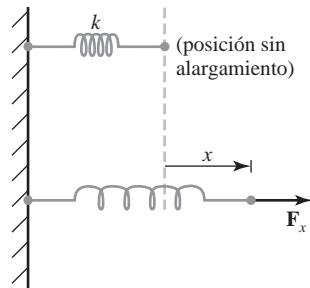
Un método común para modelar un objeto no rígido consiste en aproximar el objeto mediante una red de nodos de puntos con conexiones flexibles entre los nodos. Un tipo sencillo de unión es un muelle. La Figura 8.118 muestra una sección de una red bidimensional de muelles que se podría utilizar para aproximar el comportamiento de una toalla o de una lámina de goma. Se podrían establecer redes similares de muelles en tres dimensiones para modelar una pelota de goma o un bloque de gelatina. Para un objeto homogéneo, podemos utilizar muelles idénticos en toda la red. Si queremos que el objeto posea diferentes propiedades según direcciones diferentes, podemos utilizar distintas propiedades del muelle en las diferentes direcciones. Cuando se aplican fuerzas externas a la red de muelles, la cantidad de estiramiento o compresión de los muelles individuales depende del conjunto de valores de la *constante de elasticidad k*, que también se llama *constante de fuerza del muelle*.



**FIGURA 8.117.** Una escena, titulada *Camino a Point Reyes*, que muestra hierba con sistema de partículas, montañas fractales y superficies con texturas mapeadas. (Cortesía de Pixar. © 1983 Pixar.)



**FIGURA 8.118.** Una red bidimensional de muelles, construida con constantes de elasticidad  $k$  idénticas.



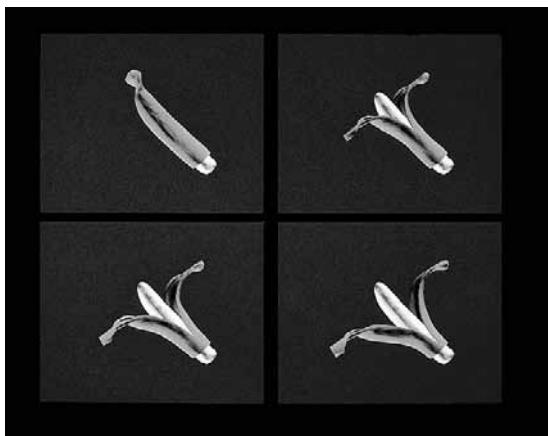
**FIGURA 8.119.** Una fuerza externa  $F_x$  tira de un extremo de un muelle, con una unión rígida en el otro extremo.

El desplazamiento horizontal  $x$  de la posición de un nodo bajo la influencia de una fuerza  $F_x$  se ilustra en la Figura 8.119. Si no se estira demasiado el muelle, podemos aproximar fielmente la cantidad de desplazamiento  $x$  desde la posición de equilibrio mediante el empleo de la ley de Hooke:

$$F_s = -F_x = -kx \quad (8.119)$$

donde  $F_s$  es la fuerza de restablecimiento igual y opuesta del muelle sobre el nodo estirado. Esta relación también se mantiene en la compresión horizontal de un muelle por una cantidad  $x$ , y tenemos relaciones similares para los desplazamientos y fuerzas en las direcciones  $y$  y  $z$ .

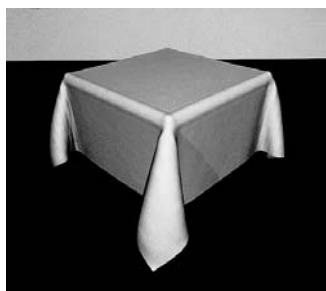
Si los objetos son completamente flexibles, vuelven a su configuración original cuando las fuerzas externas desaparecen. Pero si queremos modelar masilla, o algún otro material deformable, necesitamos modificar las características de los muelles de manera que estos no vuelvan a su forma original cuando desaparecen las fuerzas externas. Otro conjunto de fuerzas aplicadas podría deformar a continuación el objeto de alguna otra manera.



**FIGURA 8.120.** Modelado del comportamiento flexible de la piel de un plátano mediante una red de muelles. (Cortesía de David Laidlaw, John Snyder, Adam Woodbury, and Alan Barr, Computer Graphics Lab, California Institute of Technology. © 1992.)



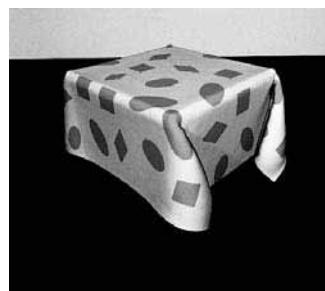
**FIGURA 8.121.** Modelado del comportamiento flexible de tela sobre muebles utilizando minimización de la función de energía. (Cortesía de Gene Greger and David E. Breen, Design Research Center, Rensselaer Polytechnic Institute. © 1992.)



(a)



(b)



(c)

**FIGURA 8.122.** Modelado de las características del (a) algodón, (b) la lana, y (c) una mezcla de poliéster y algodón mediante la utilización de la minimización de la función de energía. (Cortesía de David E. Breen and Donald H. House, Design Research Center, Rensselaer Polytechnic Institute. © 1992.)

En lugar de utilizar muelles, también podemos modelar las uniones entre los nodos con materiales elásticos y minimizar las funciones de energía de tensión para determinar la forma del objeto bajo la influencia de fuerzas externas. Este método proporciona un modelo mejor para la tela, y se han ideado varias funciones de energía para describir el comportamiento de tipos diferentes de materiales textiles.

Para modelar un objeto no rígido, en primer lugar establecemos las fuerzas externas que actúan sobre el objeto. Después consideramos la propagación de las fuerzas a través de la red que representa al objeto. Esto conduce a un sistema de ecuaciones que debemos resolver para determinar el desplazamiento de los nodos de la red.

La Figura 8.120 muestra la piel de un plátano modelada mediante una red de muelles y la escena de la Figura 8.121 muestra ejemplos de modelado de telas mediante la utilización de funciones de energía, con un patrón de mapeo de una textura sobre una tela. Al ajustar los parámetros de una red empleando cálculos mediante funciones de energía, se pueden modelar diferentes clases de tela. La Figura 8.122 ilustra modelos de materiales como el algodón, la lana y una mezcla de poliéster y algodón colocados sobre una mesa.

Los métodos de modelado basados en las características físicas también se aplican en animaciones, para proporcionar descripciones más precisas de las trayectorias del movimiento. Antaño, las animaciones se especificaban a menudo empleando trayectorias con *splines* y cinemática, donde los parámetros del movimiento se basan sólo en la posición y la velocidad. El modelado basado en las características físicas describe el movimiento utilizando ecuaciones dinámicas, que involucran fuerzas y aceleraciones. Las descripciones de animaciones basadas en las ecuaciones de la dinámica producen movimientos más realistas que aquellas basadas en las ecuaciones de la cinemática.

## 8.27 VISUALIZACIÓN DE CONJUNTOS DE DATOS

---

El uso de métodos de gráficos por computadora como ayuda en el análisis científico y de ingeniería se denomina habitualmente **visualización científica**. Ésta involucra la visualización de conjuntos de datos y procesos que pueden ser difíciles o imposibles de analizar sin métodos gráficos. Por ejemplo, las técnicas de visualización se necesitan para tratar la salida de fuentes de grandes volúmenes de datos como los monitores de las computadoras, escáneres de satélites y naves espaciales, telescopios de radioastronomía y escáneres médicos. Se generan con frecuencia millones de puntos de datos a partir de soluciones numéricas de simulaciones por computadora y a partir de equipos de observación, y es difícil determinar tendencias y relaciones mediante la simple exploración de los datos sin tratar. De forma similar, las técnicas de visualización son útiles para analizar procesos que ocurren durante un largo período de tiempo o que no se pueden observar directamente, tales como fenómenos mecánico-cuánticos y efectos de la relatividad especial producidos por objetos que viajan a velocidades cercanas a la de la luz. La visualización científica emplea métodos de los gráficos por computadora, procesamiento de imágenes, visión por computadora y otras áreas para mostrar visualmente, mejorar y manipular información para permitir una mejor comprensión de los datos. Métodos similares empleados por el comercio, la industria y otras áreas no científicas se denominan a veces **visualización empresarial**.

Los conjuntos de datos se clasifican de acuerdo a su distribución espacial y al tipo de datos. Los conjuntos de datos bidimensionales tienen valores distribuidos sobre una superficie, y los conjuntos de datos tridimensionales tienen valores distribuidos en el interior de un cubo, una esfera o alguna otra región del espacio. Entre los tipos de datos se incluyen los valores escalares, los vectores, los tensores y los datos multivariables.

### Representaciones visuales de campos escalares

Una cantidad escalar es aquella que tiene un único valor. Los conjuntos de datos escalares contienen valores que se pueden distribuir en el tiempo, así como en el espacio, y los valores de los datos también pueden ser funciones de otros parámetros escalares. Algunos ejemplos de cantidades físicas escalares son la energía, la densidad, la masa, la temperatura, la presión, la carga eléctrica, la resistencia eléctrica, la reflexividad, la frecuencia y el contenido de agua.

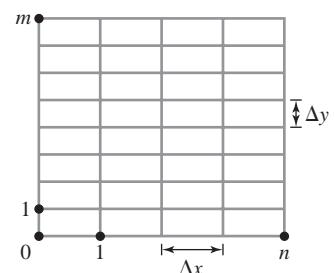
Un método habitual para visualizar un conjunto de datos escalares consiste en utilizar gráficas o diagramas que muestren la distribución de los valores de los datos en función de otros parámetros, tales como la posición y el tiempo. Si los datos se distribuyen sobre una superficie, podríamos dibujar los valores de los datos como barras verticales que se elevan desde la superficie, o podemos interpolar los valores de los datos de algún otro modo en los puntos seleccionados de la superficie. También se utilizan **métodos de pseudocolor** para distinguir los valores diferentes del conjunto de datos escalares, y las técnicas de codificación de color se pueden combinar con métodos de gráficas y diagramas. Para codificar con colores un conjunto de datos escalares, elegimos un rango de colores y mapeamos el rango de los valores de los datos al rango de color. Por ejemplo, el color azul se podría asignar al valor escalar más bajo, y el color rojo se podría asignar al valor más elevado. La Figura 8.123 proporciona un ejemplo de un gráfico de superficie codificado con color, aun-



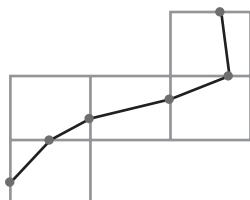
**FIGURA 8.123.** Un gráfico de superficie financiero, que muestra un crecimiento potencial de las acciones durante la caída de la bolsa de octubre de 1987. El color rojo indica alta rentabilidad, y el gráfico muestra que las acciones de bajo crecimiento se comportaron mejor en la caída. (Cortesía de Eng-Kiat Koh, Information Technology Institute, República de Singapur y Encentuate, Inc., Cupertino, California.)

que la figura se muestra en escala de grises. La codificación con color de un conjunto de datos a veces requiere un trato especial, porque ciertas combinaciones de color pueden conducir a interpretaciones erróneas de los datos.

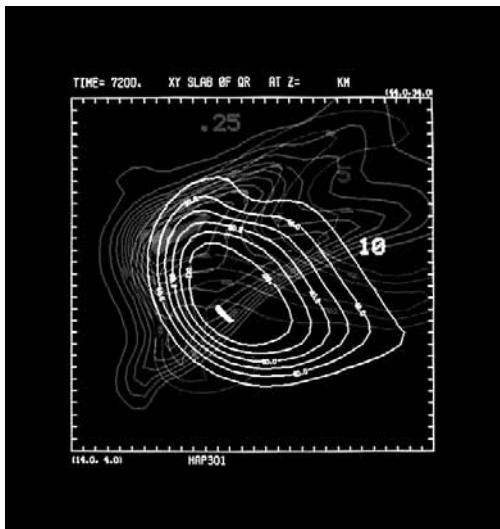
Los **gráficos de nivel** se utilizan para visualizar *curvas de nivel* (líneas de valor constante) de un conjunto de datos escalares distribuido sobre una superficie. Las curvas de nivel se espacian en un intervalo conveniente para mostrar el rango y la variación de los valores de los datos sobre la región del espacio. Una aplicación típica es un gráfico de nivel de las alturas sobre un plano de tierra. Habitualmente, los métodos de nivel se aplican a un conjunto de valores de datos que estén distribuidos sobre una cuadrícula regular, como la de la Figura 8.124. Las cuadrículas regulares tienen las líneas equiespaciadas, y los valores de los datos se conocen en las intersecciones de la cuadrícula. Las soluciones numéricas de las simulaciones por computadora se establecen habitualmente para producir distribuciones de datos sobre cuadrículas regulares, mientras que los conjuntos de datos observados se espacian a menudo de forma irregular. Se han ideado métodos de nivel para varias clases de cuadrículas no regulares, pero las distribuciones no regulares de datos se convierten a menudo en cuadrículas regulares. Un algoritmo bidimensional de nivel traza las líneas de nivel desde una celda a otra de la cuadrícula, mediante la comprobación de las cuatro esquinas de las celdas de la cuadrícula para determinar qué aristas de la celda se cruzan con una línea de nivel concreta. Las líneas de nivel se dibujan habitualmente como secciones de línea recta a través de cada celda, como se ilustra en la Figura 8.125. A veces las líneas de nivel se dibujan mediante curvas con *splines*, pero el ajuste de los *splines* puede conducir a inconsistencias y malas interpretaciones de un conjunto de datos. Por ejemplo, dos líneas de nivel con *splines* se podrían cruzar, o las trayectorias de líneas de nivel curvadas podrían no ser un verdadero indicador de las tendencias de los datos, ya que los valores de los datos sólo se conocen en las esquinas de las celdas. Los paquetes de nivel pueden permitir el ajuste interactivo de las líneas de nivel al investigador para corregir cualquier inconsistencia. En la Figura 8.126 se proporciona un ejemplo con tres gráficos de nivel sobre el plano *xy* que se superponen y que están condificados con color y la Figura 8.127 muestra las líneas de nivel y la codificación con color de un espacio de forma irregular.



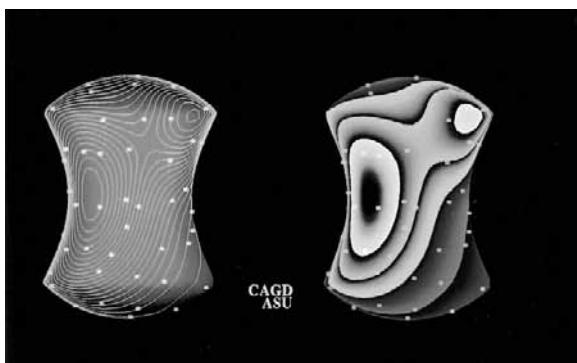
**FIGURA 8.124.** Una cuadrícula regular y bidimensional con los valores de los datos en las intersecciones de las líneas de la cuadrícula. Las líneas *x* de la cuadrícula tienen un espaciado constante  $\Delta x$  y las líneas *y* de la cuadrícula tienen un espaciado constante  $\Delta y$ , en donde el espaciado según los ejes *x* e *y* pueden no coincidir.



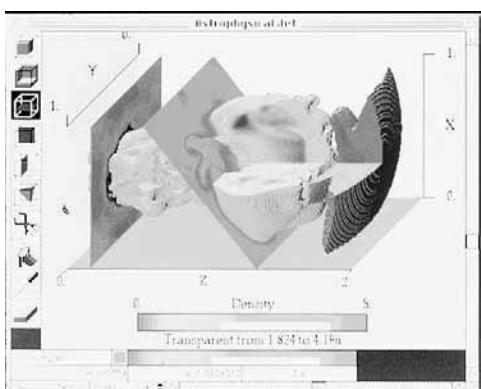
**FIGURA 8.125.** La trayectoria de una línea de nivel a través de cinco celdas de la cuadrícula.



**FIGURA 8.126.** Gráficos de nivel codificados con color de tres conjuntos de datos dentro de la misma región del plano xy. (Cortesía de National Center for Supercomputing Applications, Universidad de Illinois en Urbana-Champaign.)



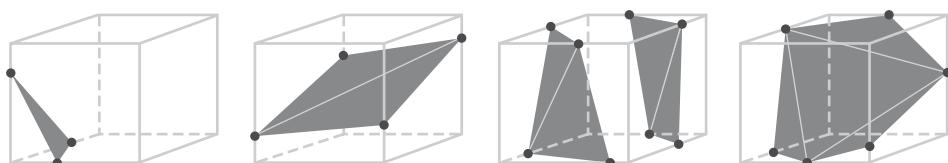
**FIGURA 8.127.** Gráficos de nivel codificados con color sobre la superficie de una región del espacio con forma de núcleo de manzana. (Cortesía de Greg Nielson, Department of Computer Science and Engineering, Universidad del Estado de Arizona.)



**FIGURA 8.128.** Secciones rectas de un conjunto de datos tridimensional. (Cortesía de Spyglass, Inc.)



**FIGURA 8.129.** Una superficie de nivel generada a partir de un conjunto de valores de contenido de agua obtenidos de un modelo numérico de una tormenta. (Cortesía de Bob Wilhelmson, Department of Atmospheric Sciences and the National Center for Supercomputing Applications, Universidad de Illinois en Urbana-Champaign.)

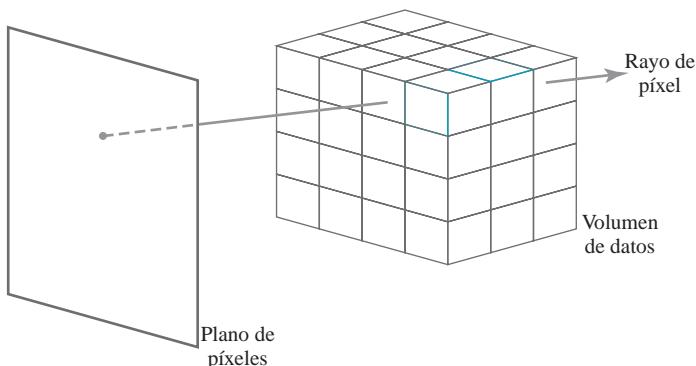


**FIGURA 8.130.** Intersecciones de superficies de nivel con las celdas de la rejilla modeladas con parches de triángulos.

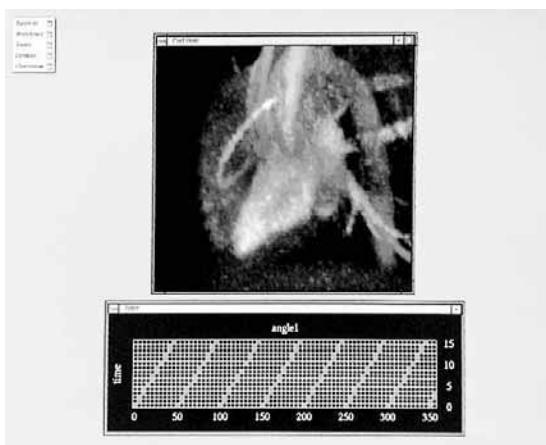
En campos de datos escalares tridimensionales, podemos realizar secciones rectas y visualizar las distribuciones bidimensionales de datos sobre las secciones. Podríamos codificar con color los valores de los datos sobre la sección o podríamos visualizar curvas de nivel. Los paquetes de visualización proporcionan habitualmente una subrutina de corte que permite obtener secciones con cualquier ángulo. La Figura 8.128 muestra una visualización generada mediante un paquete comercial de obtención de secciones.

En lugar de observar secciones rectas bidimensionales, podemos dibujar una o más **superficies de nivel**, que simplemente son gráficos tridimensionales de nivel (Figura 8.129). Cuando se visualizan dos superficies de nivel que se superponen, la superficie exterior se hace transparente a fin de que podamos ver las formas de ambas superficies. La construcción de una superficie de nivel es similar a dibujar líneas de nivel, excepto que ahora disponemos de celdas tridimensionales en la cuadricula y necesitamos comprobar los valores de los datos en los ocho vértices de una celda, para localizar las secciones de una superficie de nivel. La Figura 8.130 muestra algunos ejemplos de intersecciones de superficies de nivel con las celdas de la cuadricula. Las superficies de nivel se modelan habitualmente mediante mallas de triángulos, después se aplican algoritmos de sombreado de superficies para visualizar la forma final.

El **sombreado de volúmenes**, que es a menudo de alguna manera como a una imagen de rayos X, es otro método para visualizar un conjunto de datos tridimensionales. La información del interior a cerca de un conjunto de datos se proyecta sobre una pantalla de visualización empleando los métodos trazado de rayos presentados en la Sección 8.20. En la dirección del rayo procedente de cada píxel de pantalla (Figura 8.131), los valores de los datos interiores se examinan y codifican para su visualización. A menudo, los valores de los datos en los puntos de la cuadricula se promedian de modo que se almacena un valor para cada voxel del espacio de datos. El modo en que los datos se codifican para su visualización depende de la aplicación. Los datos sísmicos, por ejemplo, se examinan a menudo para buscar los valores máximo y mínimo en la dirección de cada rayo. Los valores se pueden entonces codificar con color para proporcionar información sobre el ancho del intervalo y el valor mínimo. En aplicaciones para medicina, los valores de los datos son factores de opa-



**FIGURA 8.131.** Visualización de volumen de una cuadrícula regular y cartesiana de datos que emplea trazado de rayos para examinar los valores de los datos interiores.



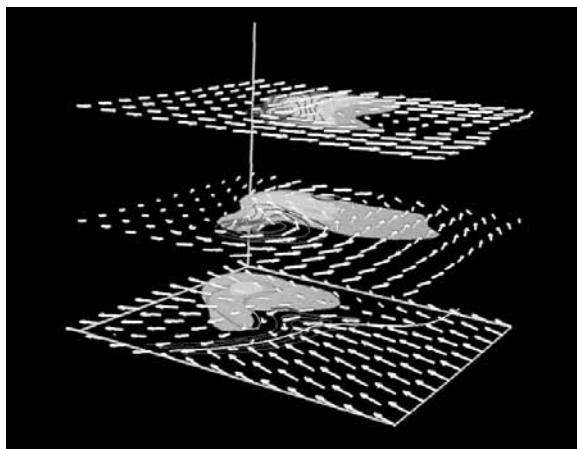
**FIGURA 8.132.** Visualización de un conjunto de datos del corazón de un perro, obtenida mediante el dibujo de la distancia codificada con color al valor máximo de voxel para cada píxel. (Cortesía de Patrick Moran y Clinton Potter, National Center for Supercomputing Applications, Universidad de Illinois en Urbana-Champaign.)

cidad en el rango que varía de 0 a 1 para capas de tejido y hueso. Las capas de hueso son completamente opacas, mientras que el tejido es de algún modo transparente (baja opacidad). En la dirección de cada rayo, los factores de opacidad se acumulan hasta que el total es mayor o igual que 1, o hasta que el rayo salga por la parte posterior de la cuadrícula tridimensional de datos. El valor de opacidad acumulada se codifica a continuación y se visualiza como un píxel en color o en escala de grises. La Figura 8.132 muestra una visualización de un volumen de un conjunto de datos médicos, que describen la estructura del corazón de un perro. En esta visualización de volumen, se mostró un gráfico codificado con color de la distancia al valor máximo de voxel en la dirección de cada rayo píxel.

## Representaciones visuales de campos vectoriales

Una cantidad vectorial  $\mathbf{V}$  en el espacio tridimensional tiene tres valores escalares ( $V_x, V_y, V_z$ ), uno para cada eje de coordenadas, y un vector bidimensional tiene dos componentes ( $V_x, V_y$ ). Otro modo de describir una cantidad vectorial consiste en proporcionar su módulo  $|\mathbf{V}|$  y su dirección como un vector unitario  $\mathbf{u}$ . Como en el caso de los escalares, las cantidades vectoriales pueden ser función de la posición, del tiempo y de otros parámetros. Algunos ejemplos de cantidades físicas vectoriales son la velocidad, la aceleración, la fuerza, la corriente eléctrica y los campos eléctrico, magnético y gravitatorio.

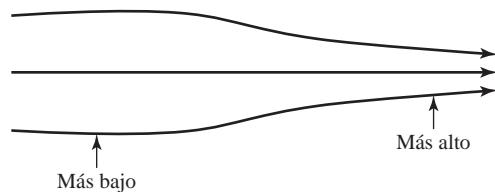
Un modo de visualizar un campo vectorial consiste en dibujar cada punto de datos como una pequeña flecha que muestra la magnitud y la dirección del vector. Este método se utiliza con mayor frecuencia en seccio-



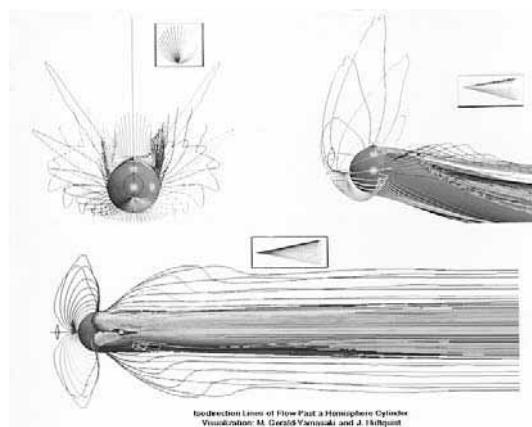
**FIGURA 8.133.** Representación mediante flechas de un campo vectorial sobre secciones rectas. (Cortesía del National Center for Supercomputing Applications, Universidad de Illinois en Urbana-Champaign.)

nes rectas, como en la Figura 8.133, ya que puede ser complicado observar las tendencias de los datos en una región tridimensional que está desordenada con flechas superpuestas. Las magnitudes de los valores vectoriales se pueden representar como variaciones en las longitudes de las flechas, o podríamos visualizar todas las flechas del mismo tamaño pero codificadas con color.

También podemos representar los valores vectoriales mediante el dibujo de *líneas de campo*, que también se denominan *líneas de flujo*. Las líneas de campo se utilizan habitualmente en campos eléctricos, magnéticos y gravitatorios. La magnitud de los valores vectoriales se indica mediante el espaciado de las líneas de campo, y la dirección del campo se representa mediante las tangentes (pendientes) a las líneas de campo, como se muestra en la Figura 8.134. En la Figura 8.135 se muestra un ejemplo de un gráfico de líneas de flujo de un campo vectorial. Las líneas de flujo se pueden visualizar como flechas anchas, particularmente cuando hay presente un efecto remolino o vórtice. Un ejemplo de esto se proporciona en la Figura 8.136, que muestra patrones de flujo de aire con turbulencias dentro de una tormenta. En animaciones de flujo de fluidos, se puede visualizar el comportamiento del campo vectorial mediante el seguimiento de partículas en la dirección del flujo. En la Figura 8.137 se muestra un ejemplo de una visualización de un campo vectorial mediante el empleo tanto de líneas de flujo como de partículas.



**FIGURA 8.134.** Representación mediante líneas de campo de un conjunto de datos vectorial.

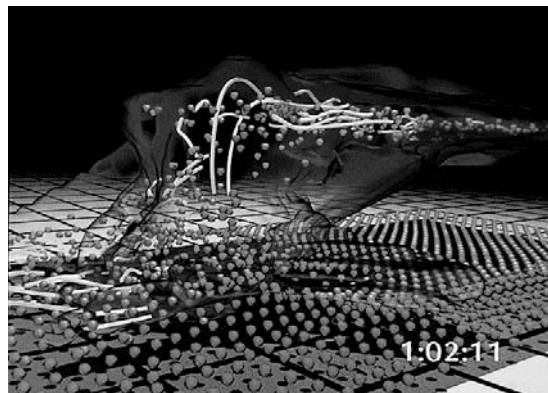


**FIGURA 8.135.** Visualización del flujo de aire alrededor de un cilindro con una tapa semiesférica, que se inclina ligeramente con respecto a la dirección del flujo de aire entrante. (Cortesía de M. Gerald-Yamasaki, J. Hultquist y Sam Uselton, NASA Ames Research Center.)

Reproduction Lines of Flow Past a Hemisphere Cylinder  
Visualization: M. Gerald-Yamasaki and J. Hultquist  
Data: S. X. You, I. B. Schreit and J. L. Steiner



**FIGURA 8.136.** Patrones retorcidos de flujo de aire, visualizados mediante líneas de flujo anchas dentro de un gráfico transparente de nivel de una tormenta. (Cortesía de Bob Wilhelmson, Department of Atmospheric Sciences and the National Center for Supercomputing Applications, Universidad de Illinois en Urbana-Champaign.)



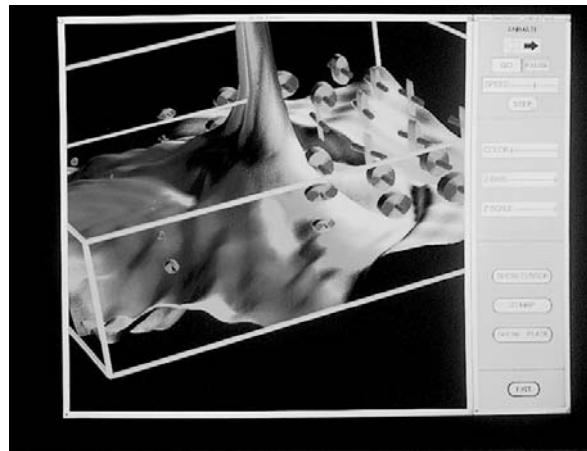
**FIGURA 8.137.** Patrones de flujo de aire, visualizados tanto con líneas de flujo como de movimiento de partículas dentro de un gráfico de una superficie transparente de nivel de una tormenta. Las partículas esféricas que se elevan presentan color naranja y las que caen color azul. (Cortesía de Bob Wilhelmson, Department of Atmospheric Sciences y del National Center for Supercomputing Applications, Universidad de Illinois en Urbana-Champaign.)

A veces, sólo se visualizan los módulos de las cantidades vectoriales. Esto se hace a menudo cuando hay que mostrar múltiples cantidades en un único punto, o cuando las direcciones no varían mucho en alguna región del espacio, o cuando las direcciones de los vectores son menos interesantes.

### Representaciones visuales de campos de tensores

Una cantidad tensorial en un espacio tridimensional tiene nueve componentes y se puede representar mediante una matriz 3 por 3. En realidad, esta representación se utiliza para un *tensor de segundo orden*. Los tensores de orden más elevado se utilizan en algunas aplicaciones, particularmente en estudios de relatividad general. Algunos ejemplos de tensores físicos de segundo orden son la tensión de un material sometido a fuerzas externas, la conductividad (o resistividad) de un conductor eléctrico y el tensor métrico, que proporciona las propiedades de un espacio de coordenadas particular. El tensor presión en coordenadas cartesianas, por ejemplo, se puede representar del siguiente modo:

$$\begin{bmatrix} \sigma_x & \sigma_{xy} & \sigma_{xz} \\ \sigma_{yx} & \sigma_y & \sigma_{yz} \\ \sigma_{zx} & \sigma_{zy} & \sigma_z \end{bmatrix} \quad (8.120)$$



**Figura 8.138.** Representación de los tensores de tensión y presión mediante un disco elíptico y una flecha sobre la superficie de un material en tensión. (Cortesía de Bob Haber, the National Center for Supercomputing Applications, Universidad de Illinois en Urbana-Champaign.)

Las magnitudes tensoriales se encuentran frecuentemente en materiales anisotrópicos, que presentan propiedades diferentes en distintas direcciones. Los elementos  $x$ ,  $xy$  y  $xz$  del tensor de conductividad, por ejemplo, describen las contribuciones de las componentes del campo eléctrico en las direcciones de los ejes  $x$ ,  $y$  y  $z$  a la corriente en la dirección del eje  $x$ . Habitualmente, las magnitudes tensoriales físicas son simétricas, de modo que el tensor sólo tiene seis valores distintos. Por ejemplo, las componentes  $xy$  e  $yx$  del tensor de presión tienen el mismo valor.

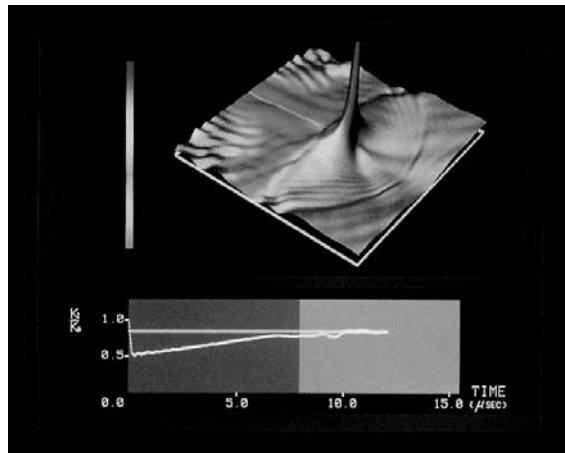
Las técnicas de visualización para representar las seis componentes de una magnitud tensorial simétrica de segundo orden se basan en idear las formas que tienen seis parámetros. Una representación gráfica de este tipo para un tensor se muestra en la Figura 8.138. Los tres elementos de la diagonal del tensor se utilizan para construir el módulo y la dirección de la flecha, y los tres términos situados fuera de la diagonal se utilizan para establecer la forma y el color del disco elíptico.

En lugar de intentar visualizar las seis componentes de una magnitud tensorial simétrica, podemos reducir el tensor a un vector o a un escalar. Empleando una representación vectorial, podemos visualizar simplemente los valores de los elementos de la diagonal del tensor. Y aplicando operaciones de *contracción de tensores*, podemos obtener una representación escalar. Por ejemplo, los tensores de tensión y presión se pueden contraer, para generar una densidad escalar de energía de presión que se puede dibujar en puntos de un material sometido a fuerzas externas (Figura 8.139).

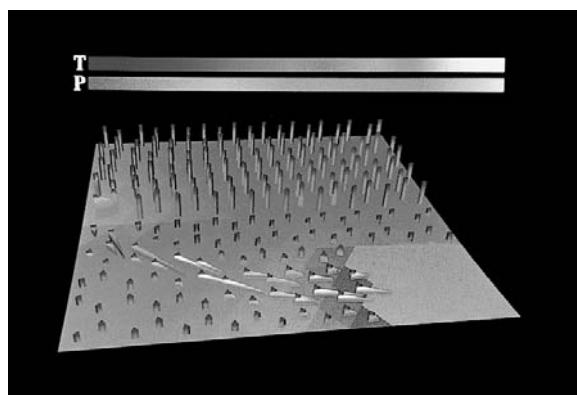
## Representaciones visuales de campos de datos multivariantes

En algunas aplicaciones, podemos querer representar valores de datos múltiples en cada punto de la cuadrícula sobre alguna región del espacio. Estos datos a menudo contienen una mezcla de valores escalares, vectoriales y tensoriales. Como ejemplo, los datos del flujo de un fluido incluyen la velocidad del fluido, la temperatura y los valores de la densidad en cada punto tridimensional. Por tanto, tenemos que visualizar cinco valores en cada punto, y la situación es similar a la de la visualización de un campo tensorial.

Un método para visualizar campos de datos multivariantes consiste en construir objetos gráficos, que a veces se denominan **glifos**, con partes múltiples. Cada parte de un glifo representa una magnitud física particular. El tamaño y el color de cada parte se puede utilizar para visualizar información a cerca de las magnitudes escalares. Para proporcionar información sobre las direcciones en un campo vectorial, podemos utilizar una cuña, un cono o alguna otra forma para apuntar en la parte del glifo que representa el vector. En la Figura 8.140 se muestra un ejemplo de la visualización de un campo de datos multivariantes, que emplea una estructura de glifo en unos puntos seleccionados de una cuadrícula.



**FIGURA 8.139.** Representación de los tensores de tensión y presión mediante un gráfico de la densidad de energía de presión, de una visualización de la propagación de grietas en la superficie de un material en tensión. (Cortesía de Bob Haber, the National Center for Supercomputing Applications, Universidad de Illinois en Urbana-Champaign.)



**FIGURA 8.140.** Un cuadro de una visualización animada de un campo dependiente del tiempo de datos multivariados mediante glifos. La parte con forma de cuña del glifo, señala la dirección de una cantidad vectorial en cada punto. (Cortesía del National Center for Supercomputing Applications, Universidad de Illinois en Urbana-Champaign.)

## 8.28 RESUMEN

Se han desarrollado muchas representaciones para modelar la amplia variedad de objetos y materiales, que podríamos querer visualizar en una escena de gráficos por computadora. En la mayoría de los casos, una representación tridimensional de un objeto se sombra mediante un paquete de software como un *objeto gráfico estándar*, cuyas superficies se muestran como una malla poligonal.

Las funciones para visualizar algunas superficies cuádricas comunes, tales como las esferas y los elipsoides, se encuentran disponibles a menudo en los paquetes gráficos. Las ampliaciones de las cuádricas, llamadas supercuádricas, proporcionan parámetros adicionales para crear una variedad más amplia de formas de objetos. Para describir superficies curvadas flexibles y no rígidas podemos utilizar objetos sin forma para crear formas como combinaciones de abultamientos gaussianos.

Los métodos más ampliamente utilizados en aplicaciones CAD son las representaciones con *splines*, que son funciones polinómicas continuas por tramos. Una curva o una superficie con *splines* se define mediante

**TABLA 8.1.** RESUMEN DE LAS FUNCIONES OpenGL PARA POLIEDROS.

<i>Función</i>	<i>Descripción</i>
glutWireTetrahedron	Muestra una pirámide (tetraedro) triangular con malla de alambre.
glutSolidTetrahedron	Muestra un tetraedro con superficie sombreada.
glutWireCube	Muestra un cubo con malla de alambre.
glutSolidCube	Muestra un cubo con superficie sombreada.
glutWireOctahedron	Muestra un octaedro con malla de alambre.
glutSolidOctahedron	Muestra un octaedro con superficie sombreada.
glutWireDodecahedron	Muestra un dodecaedro con malla de alambre.
glutSolidDodecahedron	Muestra un dodecaedro con superficie sombreada.
glutWireIcosahedron	Muestra un icosaedro con malla de alambre.
glutSolidIcosahedron	Muestra un icosaedro con superficie sombreada.

un conjunto de puntos de control y las condiciones en los límites de las secciones del *spline*. Las líneas que conectan la secuencia de puntos de control forman el grafo de control, y todos los puntos de control se encuentran dentro del armazón convexo de un objeto con *splines*. Las condiciones en los límites se pueden especificar empleando derivadas paramétricas o geométricas, y la mayor parte de las representaciones con *splines* utilizan condiciones paramétricas en los límites. Los *splines* de interpolación unen todos los puntos de control mientras que los *splines* de aproximación no unen todos los puntos de control. Una superficie con *splines* se puede describir mediante el producto cartesiano de dos polinomios. Los polinomios cúbicos se utilizan habitualmente para las representaciones de interpolación, entre los que se incluyen los *splines* de Hermite, cardinales y de Kochanek-Bartels. Los *splines* de Bézier proporcionan un método simple y potente de aproximación para describir líneas y superficies curvadas; sin embargo, el grado del polinomio se determina mediante el número de puntos de control y el control local sobre las formas de las curvas es difícil de lograr. Los *splines* B, entre los que se incluyen los *splines* de Bézier como un caso particular, son una representación de aproximación más versátil, pero requieren la especificación de un vector de nudos. Los *splines* beta son generalizaciones de los *splines* B que se especifican mediante condiciones geométricas en los límites. Los *splines* racionales se formulan como el cociente de dos representaciones con *splines*. Los *splines* racionales se pueden utilizar para describir cuádricas y son invariantes frente a transformaciones de perspectiva de visualización. Un *spline* B racional con un vector de nudos no uniforme se denomina habitualmente NURB. Para determinar los puntos a lo largo de una curva o superficie con *splines*, podemos utilizar cálculos de diferencias hacia delante o métodos de subdivisión.

Entre otras técnicas de diseño se incluyen las representaciones de barrido, los métodos de la geometría constructiva de sólidos, árboles octales y árboles BSP. Una representación de barrido se forma mediante una traslación o una rotación de una forma bidimensional a través de una región del espacio. Los métodos de la geometría constructiva de sólidos combinan dos o más formas tridimensionales empleando las operaciones de conjuntos: unión, diferencia e intersección. Los árboles octales y los árboles BSP utilizan métodos de subdivisión del espacio.

Las representaciones de geometría fractal proporcionan métodos altamente efectivos para describir fenómenos naturales. Podemos utilizar estos métodos para modelar el terreno, los árboles, los arbustos, el agua y las nubes, y para generar patrones gráficos inusuales. Un objeto fractal se puede describir mediante un procedimiento de construcción y una dimensión fractal. Entre los procedimientos de construcción de fractales se incluyen las construcciones geométricas, los métodos de desplazamiento del punto medio, las operaciones

autocuadráticas en el espacio complejo y las transformaciones de inversión. Otros métodos de procesamiento para construir representaciones de objetos que utilizan reglas de transformación son las gramáticas de formas y los graftales.

Los objetos que muestran fluidez, tales como las nubes, el humo, el fuego, el agua y aquello que explota o implota, se pueden modelar mediante sistemas de partículas. Empleando esta técnica de representación, describimos un objeto mediante un conjunto de partículas y las reglas que gobiernan los movimientos de las partículas.

Los métodos de modelado basados en las características físicas se pueden utilizar para describir las características de un objeto flexible, tal como una cuerda, una goma, o la tela. Esta técnica representa un material mediante una cuadrícula de secciones semejantes a los resortes y calcula las deformaciones empleando las fuerzas que actúan sobre el objeto.

**TABLA 8.2. RESUMEN DE FUNCIONES OpenGL PARA SUPERFICIES CUÁDRICAS Y SUPERFICIES CÚBICAS.**

<i>Función</i>	<i>Descripción</i>
glutWireSphere	Muestra una esfera de GLUT alámbrica.
glutSolidSphere	Muestra una esfera de GLUT con superficie sombreada.
glutWireCone	Muestra un cono de GLUT alámbrico.
glutSolidCone	Muestra un cono de GLUT con superficie sombreada.
glutWireTorus	Muestra un toro de GLUT de sección recta circular con malla de alambre.
glutSolidTorus	Muestra un toro de GLUT de sección recta circular con superficie sombreada.
glutWireTeapot	Muestra una tetera de GLUT alámbrica.
glutSolidTeapot	Muestra una tetera de GLUT con superficie sombreada.
gluNewQuadric	Activa el sombreador de cuádricas de GLU para un objeto cuyo nombre se haya definido mediante la declaración: <code>GLUquadricObj *nameOfObject;</code>
gluQuadricDrawStyle	Selecciona un modo de visualización de objeto de GLU con nombre predefinido.
gluSphere	Muestra una esfera de GLU.
gluCylinder	Muestra un cono, un cilindro o un cilindro con tapas de GLU.
gluDisk	Muestra una corona circular plana o un disco de GLU.
gluPartialDisk	Muestra una sección de una corona circular plana o un disco de GLU.
gluDeleteQuadric	Elimina un objeto de cuádrica de GLU.
gluQuadricOrientation	Define las orientaciones de dentro y fuera de un objeto de cuádrica de GLU.
gluQuadricNormals	Especifica cómo se deberían generar los vectores normales a la superficie de un objeto de cuádrica de GLU.
gluQuadricCallback	Especifica una función de atención a errores de un objeto de cuádrica de GLU.

**TABLA 8.3.** RESUMEN DE FUNCIONES DE BÉZIER DE OpenGL.

<i>Función</i>	<i>Descripción</i>
glMap1	Especifica los parámetros de visualización de curvas de Bézier, los valores de los colores, etc., y activa estas subrutinas empleando <code>glEnable</code> .
glEvalCoord1	Calcula un punto en coordenadas de una curva de Bézier.
glMapGrid1	Especifica el número de subdivisiones equiespaciadas entre dos parámetros de una curva de Bézier.
glEvalMesh1	Especifica el modo de visualización y el rango entero de una visualización de una curva de Bézier.
glMap2	Especifica los parámetros de visualización de superficies de Bézier, los valores de los colores, etc., y activa estas subrutinas empleando <code>glEnable</code> .
glEvalCoord2	Calcula un punto en coordenadas de una superficie de Bézier.
glMapGrid2	Especifica una cuadrícula bidimensional con subdivisiones equiespaciadas sobre una superficie de Bézier.
glEvalMesh2	Especifica el modo de visualización y el rango entero de una cuadrícula bidimensional de una superficie de Bézier.

Las técnicas de visualización utilizan los métodos de los gráficos por computadora para analizar conjuntos de datos, entre los que se incluyen los valores escalares, vectoriales y tensoriales en combinaciones variadas. Las representaciones de datos se pueden realizar mediante codificación con color o mediante la visualización de formas de objetos diferentes.

Las caras de la superficie poligonal de un objeto gráfico estándar se pueden especificar en OpenGL empleando las funciones de primitivas de polígonos, triángulos y cuadriláteros. También, las subrutinas de GLUT se encuentran disponibles para mostrar los cinco poliedros regulares. Se pueden visualizar con las funciones de GLUT y GLU esferas, conos y otros objetos con superficies cuádricas, y se proporciona una subrutina de GLUT para la generación de la tetera de Utah con superficies cúbicas. La biblioteca del núcleo de OpenGL contiene funciones para producir *splines* de Bézier, y se proporcionan las funciones de GLU para especificar *splines* B y curvas de recorte de superficies con *splines*. Las Tablas 8.1 a 8.4 resumen las funciones para poliedros, cuádricas, cúbicas y *splines* estudiadas en este capítulo.

## REFERENCIAS

Barr (1981) contiene un estudio detallado de las supercuádricas. Para obtener más información sobre el modelado con objetos sin forma, consulte Blinn (1982). El modelo de metabolas se estudia en Nishimura (1985); el modelo de objetos suaves se estudia en Wyvill, Wyvill y McPheeers (1987).

Entre las fuentes de información sobre representaciones con curvas y superficies paramétricas se incluyen Bézier (1972), Barsky y Beatty (1983), Barsky (1984), Kochanek y Bartels (1984), Huitric y Nahas (1985), Mortenson (1985), Farin (1988), Rogers y Adams (1990), y Piegl y Tiller (1997).

Los algoritmos para aplicaciones con árboles octales y árboles cuaternarios se proporcionan en Doctor y Torberg (1981), Yamaguchi, Kunii, y Fujimura (1984), y Brunet y Navazo (1990). Gordon y Chen (1991) muestran los métodos de los árboles BSP. Y Requicha y Rossignac (1992) estudian los métodos de modelado de sólidos.

**TABLA 8.4.** RESUMEN DE FUNCIONES OpenGL PARA SPLINES B.

<i>Función</i>	<i>Descripción</i>
gluNewNurbsRenderer	Activa el sombreador de GLU para <i>splines</i> B para un objeto cuyo nombre se ha definido mediante la declaración <code>GLUnurbsObj *bsplineName</code> .
gluBeginCurve	Comienza la asignación de valores de los parámetros de una curva específica de una o más secciones con <i>splines</i> B.
gluEndCurve	Señala el fin de las especificaciones de los parámetros de una curva con <i>splines</i> B.
gluNurbsCurve	Especifica los valores de los parámetros de una sección de una curva con nombre con <i>splines</i> B.
gluDeleteNurbsRenderer	Elimina un <i>spline</i> B específico.
gluNurbsProperty	Especifica las opciones de sombreado de un <i>spline</i> B.
gluGetNurbsProperty	Determina el valor actual de una propiedad de un <i>spline</i> B.
gluBeginSurface	Comienza la asignación de valores de los parámetros de una superficie específica de una o más secciones con <i>splines</i> B.
gluEndSurface	Señala el fin de las especificaciones de los parámetros de una superficie con <i>splines</i> B.
gluNurbsSurface	Especifica los valores de los parámetros de una sección de una superficie con nombre con <i>splines</i> B.
gluLoadSamplingMatrices	Especifica las matrices de visionado y de transformación geométrica que se deben utilizar en las subrutinas de muestreo y selección de un <i>spline</i> B.
gluNurbsCallback	Especifica la función de atención a un evento para un <i>spline</i> B.
gluNurbsCallbackData	Especifica los valores de los datos que se deben pasar a la función de atención a un evento.
gluBeginTrim	Comienza la asignación de los valores de los parámetros de una curva de recorte de una superficie con <i>splines</i> B.
gluEndTrim	Señala el fin de las especificaciones de los parámetros de una curva de recorte.
gluPwlCurve	Especifica los valores de los parámetros de una curva de recorte de una superficie con <i>splines</i> B.

Para obtener más información sobre representaciones fractales, consulte Mandelbrot (1977 y 1982), Fournier, Fussel, y Carpenter (1982), Norton (1982), Peitgen y Richter (1986), Peitgen y Saupe (1988), Hart, Sandin, y Kauffman (1989), Koh y Hearn (1992), y Barnsley (1993). En Fournier y Reeves (1986) y en Fowler, Meinhardt y Prusinkiewicz (1992) se proporcionan métodos de modelado de varios fenómenos naturales. Las gramáticas de formas se muestran en Glassner (1992) y los sistemas de partículas se estudian en Reeves (1983). Los métodos basados en las características físicas se abordan en Barzel (1992).

En Hearn y Baker (1991) se proporciona una introducción general a los algoritmos de visualización. Se puede encontrar información adicional sobre técnicas específicas de visualización en Sabin (1985), Lorensen

y Cline (1987), Drebin, Carpenter y Hanrahan (1988), Sabella (1988), Upson y Keeler (1988), Frenkel (1989), Nielson, Shriver y Rosenblum (1990), y Nielson (1993). En Tufte (1990, 1997, y 2001) se proporcionan líneas de actuación para representaciones visuales de información.

Se pueden encontrar técnicas de programación de varias representaciones en Glassner (1990), Arvo (1991), Kirk (1992), Heckbert (1994), y Paeth (1995). Se pueden encontrar ejemplos de programación adicionales de *splines* de Bézier, *splines* B y funciones para curvas de recorte con OpenGL en Woo, Neider, Davis y Shreiner (1999). Kilgard (1996) estudia las funciones de GLUT para visualizar poliedros, superficies cuádricas y la tetera de Utah. Y en Shreiner (2000) se muestra un listado completo de las funciones OpenGL de la biblioteca del núcleo y de GLU.

## EJERCICIOS

---

- 8.1 Establezca un algoritmo para convertir una esfera en una representación mediante una malla poligonal.
- 8.2 Establezca un algoritmo para convertir un elipsoide en una representación mediante una malla poligonal.
- 8.3 Establezca un algoritmo para convertir un cilindro en una representación mediante una malla poligonal.
- 8.4 Establezca un algoritmo para convertir un superelipsoide en una representación mediante una malla poligonal.
- 8.5 Establezca un algoritmo para convertir una metabola en una representación mediante una malla poligonal.
- 8.6 Escriba una subrutina para visualizar una curva bidimensional con *splines* cardinales, que utilice un conjunto de puntos de control del plano *xy* como entrada.
- 8.7 Escriba una subrutina para visualizar una curva bidimensional de Kochanek-Bartels, que utilice un conjunto de puntos de control del plano *xy* como entrada.
- 8.8 ¿Cuáles son las funciones de combinación de las curvas de Bézier en el caso de tener tres puntos de control especificados en el plano *xy*? Dibuje cada función e identifique los valores mínimo y máximo de las funciones de combinación.
- 8.9 ¿Cuáles son las funciones de combinación de las curvas de Bézier en el caso de tener tres puntos de control especificados en el plano *xy*? Dibuje cada función e identifique los valores mínimo y máximo de las funciones de combinación.
- 8.10 Modifique el programa de ejemplo de la Sección 8.10 para visualizar una curva cúbica de Bézier, utilizando como entrada un conjunto de cuatro puntos de control del plano *xy*.
- 8.11 Modifique el ejemplo de la Sección 8.10 para visualizar un curva de Bézier de grado  $n - 1$ , utilizando como entrada un conjunto de  $n$  puntos de control del plano *xy*.
- 8.12 Complete el ejemplo de programación con OpenGL de la Sección 8.18 para visualizar cualquier curva cúbica de Bézier, utilizando como entrada un conjunto de cuatro puntos de control del plano *xy*.
- 8.13 Complete el ejemplo de programación con OpenGL de la Sección 8.18 para visualizar cualquier curva espacial cúbica de Bézier, utilizando como entrada un conjunto de cuatro puntos de control del plano *xy*. Utilice una proyección ortogonal para visualizar la curva y los parámetros de visualización como entrada.
- 8.14 Escriba una subrutina que se pueda utilizar para diseñar formas de curvas bidimensionales de Bézier que posean continuidad por tramos de primer orden. El número y la posición de los puntos de control de cada sección de la curva se deben especificar como entrada.
- 8.15 Escriba una subrutina que se pueda utilizar para diseñar formas de curvas bidimensionales de Bézier que posean continuidad por tramos de segundo orden. El número y la posición de los puntos de control de cada sección de la curva se deben especificar como entrada.
- 8.16 Modifique el programa de ejemplo de la Sección 8.10 para visualizar cualquier curva cúbica de Bézier, utilizando como entrada un conjunto de cuatro puntos de control del plano *xy*. Emplee el método de la subdivisión para calcular los puntos de la curva.

- 8.17 Modifique el programa de ejemplo de la Sección 8.10 para visualizar cualquier curva cúbica de Bézier, utilizando como entrada un conjunto de cuatro puntos de control del plano  $xy$ . Emplee diferencias hacia adelante para calcular los puntos de la curva.
- 8.18 ¿Cuáles son las funciones de combinación de una curva con *splines* B bidimensional, uniforme, periódica y con  $d = 5$ ?
- 8.19 ¿Cuáles son las funciones de combinación de una curva con *splines* B bidimensional, uniforme, periódica y con  $d = 6$ ?
- 8.20 Modifique el programa de ejemplo de la Sección 8.10 para visualizar una curva con *spline* B bidimensional, uniforme y periódica, utilizando como entrada un conjunto de puntos de control del plano  $xy$ . Emplee diferencias hacia adelante para calcular los puntos de la curva.
- 8.21 Modifique el programa del ejemplo anterior para visualizar la curva con *splines* B empleando funciones OpenGL.
- 8.22 Escriba una subrutina para visualizar cualquier cónica en el plano  $xy$  utilizando una representación mediante un *spline* racional de Bézier.
- 8.23 Escriba una subrutina para visualizar cualquier cónica en el plano  $xy$  utilizando una representación mediante un *spline* B racional.
- 8.24 Desarrolle un algoritmo para calcular el vector normal a una superficie de Bézier en un punto  $\mathbf{P}(u, v)$ .
- 8.25 Obtenga las expresiones para calcular las diferencias hacia delante para una curva cuadrática.
- 8.26 Obtenga las expresiones para calcular las diferencias hacia delante para una curva cúbica.
- 8.27 Establezca los procedimientos para generar la descripción de un objeto tridimensional a partir de la entrada de los parámetros que definen el objeto en función de un barrido de traslación de una forma bidimensional.
- 8.28 Establezca los procedimientos para generar la descripción de un objeto tridimensional a partir de la entrada de los parámetros que definen el objeto en función de un barrido de rotación de una forma bidimensional.
- 8.29 Idee un algoritmo para generar objetos sólidos como combinaciones de formas primitivas tridimensionales, tales como un cubo y una esfera, empleando los métodos de la geometría constructiva de sólidos.
- 8.30 Modifique el algoritmo del ejercicio anterior de manera que las formas primitivas se definan con estructuras con árboles octales.
- 8.31 Desarrolle un algoritmo para codificar una escena bidimensional como una representación mediante un árbol cuaternario.
- 8.32 Desarrolle un algoritmo para transformar una representación mediante árboles cuaternarios en píxeles del búfer de imagen.
- 8.33 Escriba una subrutina para convertir una descripción mediante una malla poligonal de un objeto tridimensional en un árbol octal.
- 8.34 Empleando un método aleatorio de desplazamiento del punto medio, escriba una subrutina para crear un contorno de una montaña, comenzando por una línea horizontal del plano  $xy$ .
- 8.35 Escriba una subrutina para calcular las alturas sobre un plano de tierra empleando el método de desplazamiento aleatorio del punto medio, a partir de un conjunto de alturas en las esquinas del plano de tierra.
- 8.36 Escriba un programa para visualizar un copo fractal de nieve (curva de Koch) para un número de iteraciones dado.
- 8.37 Escriba un programa para generar una curva fractal con un número de iteraciones concreto, empleando uno de los generadores de la Figura 8.73 o la Figura 8.74. ¿Cuál es la dimensión fractal de la curva?
- 8.38 Escriba un programa para generar curvas fractales empleando la función autocuadrática  $f(z) = z^2 + \lambda$ , donde la constante  $\lambda$  se especifica como entrada.
- 8.39 Escriba un programa que genere curvas fractales usando la función autocuadrática  $f(z) = i(z^2 + 1)$ , donde  $i = \sqrt{-1}$ .
- 8.40 Modifique el ejemplo de programación de la Sección 8.23 para utilizar niveles de color adicionales en la visualización de los límites de las regiones alrededor del conjunto de Mandelbrot.

- 8.41** Modifique el programa del ejercicio anterior para permitir que los colores y el número de niveles de color se proporcionen como entradas.
- 8.42** Modifique el programa del ejercicio anterior para seleccionar y visualizar cualquier región frontera rectangular (el área de ampliación) alrededor del conjunto de Mandelbrot.
- 8.43** Escriba una subrutina para implementar la inversión de puntos, Ecuación 8.118, para un círculo y un conjunto de puntos específicos.
- 8.44** Idee un conjunto de reglas de sustitución geométricas para alterar la forma de un triángulo equilátero.
- 8.45** Escriba un programa para el ejercicio anterior que muestre las etapas de la conversión del triángulo.
- 8.46** Escriba un programa para modelar y visualizar una esfera del plano  $xy$  que explota, utilizando un sistema de partículas.
- 8.47** Modifique el programa del ejercicio anterior para explotar un petardo (cilindro).
- 8.48** Idee una subrutina para modelar un trozo pequeño rectangular de tela como una cuadrícula de muelles idénticos.
- 8.49** Escriba una subrutina para visualizar un conjunto bidimensional de datos escalares empleando una representación mediante pseudocolor.
- 8.50** Escriba una subrutina para visualizar un conjunto bidimensional de datos empleando líneas de nivel.
- 8.51** Escriba una subrutina para visualizar un conjunto bidimensional de datos vectoriales, empleando una representación con flechas para los valores vectoriales. Utilice una flecha de tamaño fijo con diferentes codificaciones de color.



# Métodos de detección de superficies visibles



Un paisaje infográfico que muestra los árboles visibles dispuestos entorno a un claro del bosque.  
(Cortesía de Thomson Digital Image, Inc.)

<b>9.1</b>	Clasificación de los algoritmos de detección de superficies visibles	<b>9.9</b>	Métodos de árboles octales
<b>9.2</b>	Detección de caras posteriores	<b>9.10</b>	Método de proyección de rayos
<b>9.3</b>	Método del búfer de profundidad	<b>9.11</b>	Comparación de los métodos de detección de visibilidad
<b>9.4</b>	Método del búfer A	<b>9.12</b>	Superficies curvas
<b>9.5</b>	Método de la línea de exploración	<b>9.13</b>	Métodos de visibilidad para imágenes alámbricas
<b>9.6</b>	Método de ordenación de la profundidad	<b>9.14</b>	Funciones OpenGL de detección de visibilidad
<b>9.7</b>	Método del árbol BSP	<b>9.15</b>	Resumen
<b>9.8</b>	Método de la subdivisión de áreas		

Uno de los problemas principales en la generación de imágenes gráficas realistas consiste en determinar qué cosas son visibles dentro de una escena desde una posición de visualización seleccionada. Son diversas las técnicas que podemos utilizar para llevar a cabo esta tarea y se han desarrollado numerosos algoritmos para la eficiente identificación y visualización de objetos visibles en distintos tipos de aplicaciones. Algunos métodos requieren más memoria, otros consumen un mayor tiempo de procesamiento y algunos sólo pueden aplicarse a tipos especiales de objeto. Qué método elijamos para una aplicación concreta puede depender de factores tales como la complejidad de la escena, el tipo de los objetos que haya que mostrar, el equipo gráfico disponible y si se necesitan generar imágenes estáticas o animadas. Estos diversos algoritmos se denominan métodos de **detección de superficies visibles**. En ocasiones, también se llamaba métodos de **eliminación de superficies ocultas**, aunque puede que existan sutiles diferencias entre la identificación de superficies visibles y la eliminación de superficies ocultas. Con una imagen alámbrica, por ejemplo, puede que no queramos eliminar las superficies ocultas, sino sólo mostrarlas con contornos punteados o con algún otro tipo de identificación, con el fin de retener la información acerca de la forma del objeto.

## 9.1 CLASIFICACIÓN DE LOS ALGORITMOS DE DETECCIÓN DE SUPERFICIES VISIBLES

---

Podemos clasificar los algoritmos de detección de superficies visibles en sentido amplio dependiendo de si tratan con las definiciones de los objetos o con sus imágenes proyectadas. Estas dos técnicas se denominan métodos del **espacio de objetos** y métodos del **espacio de imagen**, respectivamente. Un método del espacio de objetos compara los objetos y las partes de los objetos entre sí, dentro de la definición de la escena, para determinar qué superficies debemos etiquetar como visibles en su conjunto. En un algoritmo del espacio de imagen, la visibilidad se decide punto a punto en cada posición de píxel del plano de proyección. La mayoría de los algoritmos para detección de superficies visibles utilizan métodos del espacio de imagen, aunque los métodos del espacio de objetos pueden usarse de manera efectiva para localizar las superficies visibles en algunos casos. Por ejemplo, los algoritmos de visualización de líneas utilizan generalmente métodos del espacio de objetos para identificar las líneas visibles en las imágenes alámbricas, pero muchos algoritmos de detección de superficies visibles en el espacio de imágenes pueden adaptarse fácilmente para la detección de líneas visibles.

Aunque existen importantes diferencias en los enfoques básicos adoptados por los diversos algoritmos de detección de superficies visibles, la mayoría de ellos utilizan técnicas de ordenación y coherencia para mejorar la velocidad. La ordenación se usa para facilitar las comparaciones de profundidad, ordenando las superficies individuales de una escena de acuerdo con su distancia con respecto al plano de visualización. Los métodos de coherencia se emplean para aprovechar las regularidades de una escena. Cabe esperar que una línea de barrido individual contenga intervalos (recorridos) con intensidades de píxel constantes, y los patrones de las líneas de barrido a menudo cambian muy poco de una línea a la siguiente. Las imágenes de las secuencias de animación sólo contienen cambios en la vecindad de los objetos en movimiento. Asimismo, a menudo pueden establecerse relaciones constantes entre los objetos de una escena.

## 9.2 DETECCIÓN DE CARAS POSTERIORES

---

Un método rápido y simple en el espacio de objetos para localizar las **caras posteriores** de un poliedro se basa en los tests frontal-posterior que hemos presentado en la Sección 3.15. Un punto  $(x, y, z)$  está detrás de una superficie poligonal si

$$Ax + By + Cz + D < 0 \quad (9.1)$$

donde  $A, B, C$  y  $D$  son los parámetros del plano correspondiente al polígono. Cuando este punto se encuentre a lo largo de la línea de visión que da a la superficie, tenemos que estar mirando a la parte posterior del polígono. Por tanto, podemos utilizar la posición de visualización para detectar las caras posteriores.

Podemos simplificar el test de caras posteriores considerando la dirección del vector normal  $\mathbf{N}$  para una superficie poligonal. Si  $\mathbf{V}_{\text{view}}$  es un vector en la dirección de visualización procedente de nuestra posición de cámara, como se muestra en la Figura 9.1, un polígono será una cara posterior si,

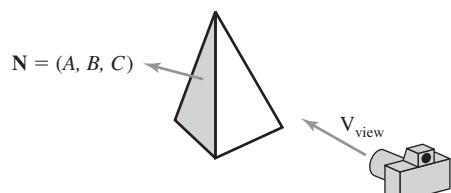
$$\mathbf{V}_{\text{view}} \cdot \mathbf{N} > 0 \quad (9.2)$$

Además, si las descripciones de los objetos han sido convertidas a coordenadas de proyección y nuestra dirección de visualización es paralela al eje  $z_v$ , sólo será necesario tener en cuenta la componente  $z$  del vector normal  $\mathbf{N}$ .

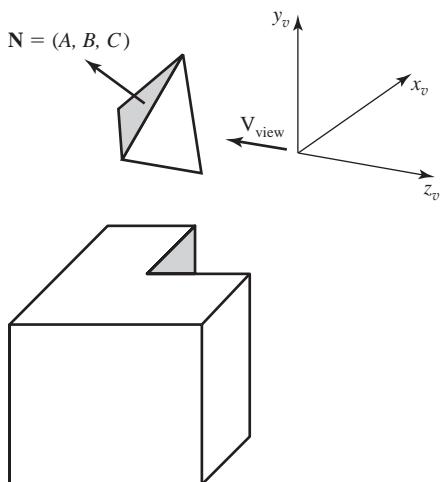
En un sistema de visualización que cumpla con la regla de la mano derecha y que tenga la dirección de visualización definida según el eje  $z_v$  negativo (Figura 9.2), un polígono será una cara posterior si la componente  $z$ ,  $C$  de su vector normal  $\mathbf{N}$  satisface la condición  $C < 0$ . Asimismo, no podremos ver ninguna cara cuya normal tenga componente  $z$  de valor  $C = 0$ , ya que nuestra dirección de visualización será tangente a dicho polígono. Así, en general, podemos etiquetar cualquier polígono como una cara posterior si su vector normal tiene una componente  $z$  cuyo valor satisface la desigualdad:

$$C \leq 0 \quad (9.2)$$

Pueden utilizarse métodos similares en los paquetes que emplean un sistema de visualización que cumpla con la regla de la mano izquierda. En estos paquetes, los parámetros del plano  $A, B, C$  y  $D$  pueden calcularse a partir de las coordenadas de los vértices del polígono especificados en sentido de las agujas del reloj (en lugar de en el sentido contrario a las agujas del reloj que se emplea en los sistemas que cumplen con la regla de la mano derecha). La desigualdad 9.1 continuará entonces siendo válida para los puntos situados detrás del



**FIGURA 9.1.** Un vector normal de superficie  $\mathbf{N}$  y el vector de dirección de visualización  $\mathbf{V}_{\text{view}}$ .



**FIGURA 9.2.** Una superficie poligonal con parámetro del plano  $C < 0$  en un sistema de coordenadas de visualización que cumpla con la regla de la mano derecha será una cara posterior cuando la dirección de visualización esté definida según el eje  $z_v$  negativo.

**FIGURA 9.3.** Vista de un poliedro cóncavo con una cara parcialmente oculta por otras caras del objeto.

polígono. Asimismo, las caras posteriores tendrán vectores normales que se alejan de la posición de visualización y que pueden identificarse mediante la desigualdad  $C \geq 0$  cuando la dirección de visualización se define según el eje  $z_v$  positivo.

Examinando el parámetro  $C$  para las diferentes superficies planas que describen un objeto, podemos identificar inmediatamente todas las caras posteriores. Para un único poliedro convexo, como la pirámide de la Figura 9.2, este test identifica todas las superficies ocultas de la escena, ya que cada superficie será completamente visible o completamente oculta. Asimismo, si una escena sólo contiene poliedros convexos no solapados, de nuevo todas las superficies ocultas podrán ser identificadas con el método de la cara posterior.

Para otros objetos, como el poliedro cóncavo de la Figura 9.3, es necesario efectuar más comprobaciones para determinar si hay caras adicionales que están parcial o totalmente oscurecidas por otras caras. Una escena general cualquiera contendrá objetos solapados a lo largo de la línea de visión, por lo que necesitaremos determinar si los objetos tapados están parcial o completamente ocultos por otros objetos. En general, la eliminación de caras posteriores permite eliminar aproximadamente la mitad de las superficies poligonales de una escena, con lo que nos ahorraremos tener que aplicar tests adicionales de visibilidad.

## 9.3 MÉTODO DEL BÚFER DE PROFUNDIDAD

Una técnica del espacio de imagen comúnmente utilizada para la detección de superficies visibles es el **método del búfer de profundidad**, que compara los valores de profundidad de las superficies en una escena para cada posición de píxel sobre el plano de proyección. Cada superficie de la escena se procesa por separado, procesando una posición de píxel de la superficie cada vez. El algoritmo se suele aplicar únicamente a aquellas escenas que sólo contienen superficies poligonales, porque los valores de profundidad pueden calcularse muy rápidamente y el método resulta fácil de implementar. Pero también podríamos aplicar los mismos procedimientos a superficies no planas. Esta técnica de detección de visibilidad también se denomina frecuentemente como *método del búfer z*, ya que la profundidad del objeto se suele medir a lo largo del eje  $z$  de un sistema de visualización.

La Figura 9.4 muestra tres superficies situadas a distancias diferentes según la línea de proyección ortográfica que va del punto  $(x, y)$  al plano de visualización. Estas superficies pueden procesarse en cualquier orden. A medida que se procesa cada superficie, su profundidad con respecto al plano de visualización se compara con las superficies previamente procesadas. Si una superficie está más próxima que todas las anteriormente procesadas, se calcula y almacena su color de superficie, junto con su profundidad. Las superficies visibles de una escena estarán representadas por el conjunto de colores de superficie que estén almacenados

después de haber completado el procesamiento de todas las superficies. La implementación del algoritmo del búfer de profundidad se suele realizar en coordenadas normalizadas, de modo que los valores de profundidad van desde 0 en el plano de recorte próximo (el plano de visualización) a 1.0 en el plano de recorte lejano.

Como el nombre de este método indica, se requieren dos zonas de búfer. Se utiliza un búfer de profundidad para almacenar los valores de profundidad de cada posición  $(x, y)$  a medida que se procesan las superficies, y en el búfer de imagen se almacenan los valores de color de las superficies para cada posición de píxel. Inicialmente, todas las posiciones en el búfer de profundidad tienen asignado el valor 1.0 (profundidad máxima) y el búfer de imagen (búfer de refresco) se inicializa con el color de fondo. A continuación se procesa cada una de las superficies enumeradas en las tablas de polígonos, de línea en línea, calculando el valor de profundidad en cada posición de píxel  $(x, y)$ . Esta profundidad calculada se compara con el valor previamente almacenado en el búfer de profundidad para dicha posición de píxel. Si la profundidad calculada es menor que el valor almacenado en el búfer de profundidad, se almacena el nuevo valor de profundidad y el color de superficie para dicha posición se calcula y se almacena en la correspondiente dirección de píxel dentro del búfer de imagen.

Los pasos de procesamiento para el método del búfer de profundidad se resumen en el siguiente algoritmo, suponiendo que los valores de profundidad estén normalizados en el rango que va de 0.0 a 1.0, con el plano de visualización a profundidad = 0. También podemos aplicar este algoritmo para cualquier otro rango de profundidades, y algunos paquetes gráficos permiten al usuario especificar el rango de profundidades sobre el que hay que aplicar el algoritmo del búfer de profundidad.

### Algoritmo del búfer de profundidad

1. Inicializar el búfer de profundidad y el búfer de imagen de modo que para todas las posiciones de búfer  $(x, y)$ ,

```
depthBuff (x, y) = 1.0, frameBuff (x, y) = backgndColor
```

2. Procesar cada polígono de una escena consecutivamente.

- Para cada posición de píxel  $(x, y)$  proyectada de un polígono, calcular la profundidad  $z$  (si no se conoce ya).
- Si  $z < \text{depthBuff} (x, y)$ , calcular el color de superficie para dicha posición y hacer

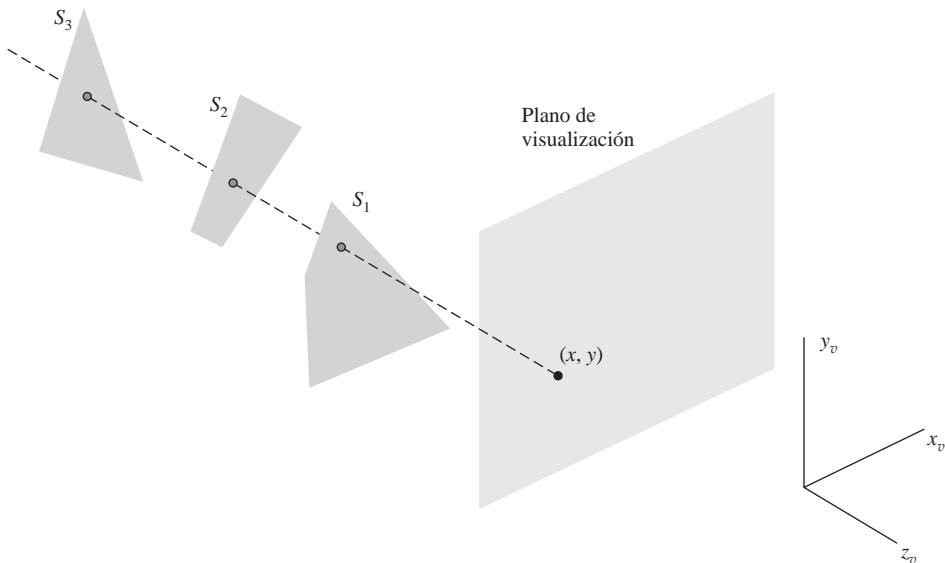
```
depthBuff (x, y) = z, frameBuff (x, y) = surfColor (x, y)
```

Después de que todas las superficies hayan sido procesadas, el búfer de profundidad contendrá los valores de profundidad de las superficies visibles y el búfer de imagen contendrá los correspondientes valores de color de dicha superficie.

Dados los valores de profundidad para los vértices de cualquier polígono en una escena, podemos calcular la profundidad en cualquier otro punto del plano que contiene al polígono. En la posición de superficie  $(x, y)$ , la profundidad se calcula a partir de la ecuación del plano:

$$z = \frac{-Ax - By - D}{C} \quad (9.4)$$

Para cualquier línea de barrido (Figura 9.5) las posiciones  $x$  horizontales adyacentes para línea difieren en  $\pm 1$  y los valores verticales  $y$  en líneas de barrido adyacentes difieren en  $\pm 1$ . Si la profundidad de la posición  $(x, y)$  es  $z$ , entonces la profundidad  $z'$  de la siguiente posición  $(x + 1, y)$  de la línea de barrido se obtendrá a partir de la Ecuación 9.4 como:



**FIGURA 9.4.** Tres superficies que se solapan para la posición de píxel  $(x, y)$  del plano de visualización. La superficie visible  $S_1$ , tiene el menor de los valores de profundidad.

$$z' = \frac{-A(x+1) - By - D}{C} \quad (9.5)$$

o

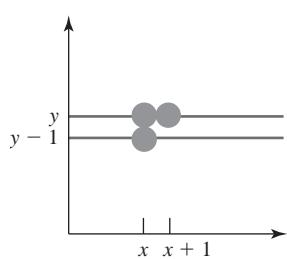
$$z' = z - \frac{A}{C} \quad (9.6)$$

El cociente  $-A/C$  es constante para cada superficie, por lo que los sucesivos valores de profundidad en una línea de barrido se obtienen a partir de los valores anteriores mediante una única suma.

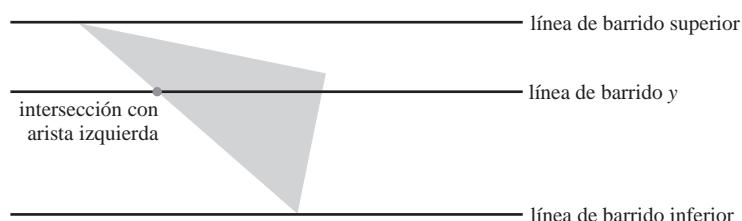
Procesando las posiciones de píxel de izquierda a derecha en cada línea de barrido, comenzamos calculando la profundidad en una arista del polígono situada a la izquierda que intersecte dicha línea de barrido (Figura 9.6). Para cada posición sucesiva a lo largo de la línea de barrido, calculamos entonces el valor de profundidad utilizando la Ecuación 9.6.

$$x' = x - \frac{1}{m}$$

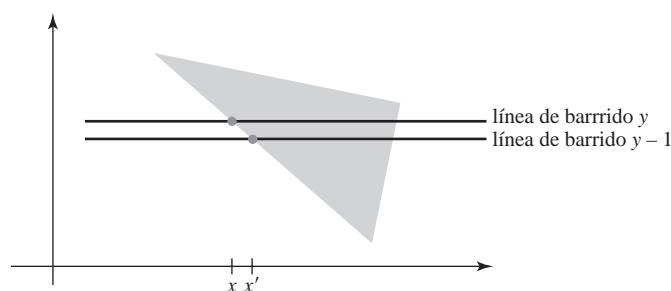
donde  $m$  es la pendiente de la arista (Figura 9.7). Los valores de profundidad a medida que se desciende por esta arista se pueden obtener recursivamente mediante,



**Figura 9.5.** A partir de la posición  $(x, y)$  en una línea de barrido, la siguiente posición de la línea tiene las coordenadas  $(x + 1, y)$  y la posición situada inmediatamente debajo en la siguiente línea de barrido tiene coordenadas  $(x, y - 1)$ .



**FIGURA 9.6.** Líneas de barrido que intersectan una cara poligonal.



**FIGURA 9.7.** Posiciones de intersección en líneas de barrido sucesivas a lo largo de una arista izquierda de un polígono.

$$z' = z + \frac{A/m + B}{C} \quad (9.7)$$

Si el procesamiento se efectúa descendiendo por una arista vertical, la pendiente es infinita y los cálculos recursivos se reducen a:

$$z' = z + \frac{B}{C}$$

Una técnica alternativa consiste en utilizar un método del punto medio o un algoritmo de tipo Bresenham para determinar los valores  $x$  iniciales a lo largo de las aristas, para cada línea de barrido. Asimismo, este método puede aplicarse a superficies curvas determinando los valores de profundidad y de color en cada punto de proyección de la superficie.

Para las superficies poligonales, el método del búfer de profundidad es muy fácil de implementar y no requiere ninguna ordenación de las superficies de la escena, aunque lo que sí hace falta es disponer de un segundo búfer, además del búfer de refresco. Un sistema con una resolución de 1280 por 1024, por ejemplo, requeriría más de 1.3 millones de posiciones en el búfer de profundidad, debiendo cada posición contener los bits suficientes como para representar el número de incrementos de profundidad necesarios. Una forma de reducir los requisitos de almacenamiento consiste en procesar una sección de la escena cada vez, utilizando un búfer de profundidad menor. Después de procesada cada sección de visualización, el búfer se reutiliza para la siguiente sección.

Además, el algoritmo básico del búfer de profundidad realiza a menudo cálculos innecesarios. Los objetos se procesan en orden arbitrario, de modo que puede calcularse un color para un punto de la superficie que luego será sustituido por el de otra superficie más próxima. Para aliviar parcialmente este problema, algunos paquetes gráficos proporcionan opciones que permiten al usuario ajustar el rango de profundidades para comprobación de superficies. Esto permite, por ejemplo, excluir los objetos distantes de las comprobaciones de profundidad. Utilizando esta opción, podríamos incluso excluir objetos que se encuentren muy próximos al plano de proyección. Normalmente, los sistemas infográficos sofisticados incluyen implementaciones hardware del algoritmo de búfer de profundidad.

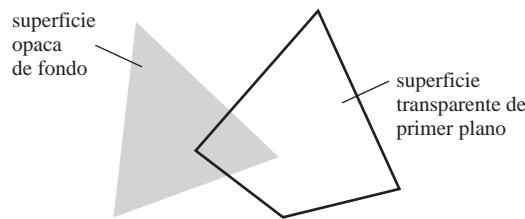
## 9.4 MÉTODO DEL BÚFER A

Una extensión de los conceptos del búfer de profundidad es el algoritmo de **búfer A** (letra situada en el otro extremo del alfabeto con respecto al «búfer z», donde  $z$  representa la profundidad). Esta extensión del búfer de profundidad es un método de detección de visibilidad, promediado de área y antialiasing desarrollado en Lucasfilm Studios para su inclusión en el sistema de representación de superficies denominado REYES (un acrónimo de «Renders Everything You Ever Saw», que podría traducirse por «capaz de representar cualquier cosa que hayas visto»). La región de búfer para este algoritmo se denomina *búfer de acumulación*, porque se utiliza para almacenar diversos datos de la superficie, además de los valores de profundidad.

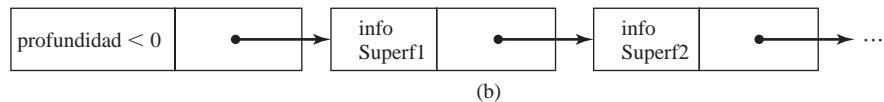
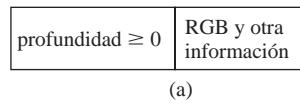
Una desventaja del método del búfer de profundidad es que identifica únicamente una superficie visible en cada posición de píxel. En otras palabras, sólo es capaz de manejar superficies opacas y no puede acumular valores de color para más de una superficie, tal como hace falta si se quieren representar superficies transparentes (Figura 9.8). El método del búfer A extiende el algoritmo del búfer de profundidad para que cada posición del búfer pueda hacer referencia a una lista enlazada de superficies. Esto permite calcular un color de píxel como combinación de diferentes colores de superficie, para aplicar efectos de transparencia o de *antialiasing*.

- Cada posición del búfer A tiene dos campos:
- Campo de profundidad: almacena un número real (positivo, negativo o cero).
- Campo de datos de la superficie: almacena datos de la superficie o un puntero.

Si el campo de profundidad es no negativo, el número almacenado en dicha posición es la profundidad de una superficie que se solapa con la correspondiente área de píxel. Entonces, el campo de datos de superficie almacena diversa información sobre la superficie, como el color existente en dicha posición y el porcentaje de recubrimiento del píxel, como se ilustra en la Figura 9.9(a). Si el campo de profundidad para una posición del búfer A es negativo, esto indica múltiples contribuciones de superficies al color del píxel. El campo de color almacena entonces un puntero a una lista enlazada de datos de superficie, como en la Figura 9.9(b). La información de superficie que se almacena en el búfer A incluye,



**FIGURA 9.8.** La visualización de una superficie opaca a través de una superficie transparente requiere múltiples entradas de color y la aplicación de operaciones de mezcla de color.



**FIGURA 9.9.** Dos posibles organizaciones para la información de superficie en la representación de una posición de píxel en el búfer A. Cuando sólo hay una única superficie solapada en el pixel, la profundidad de la superficie, su color y otras informaciones se almacenan como en (a). Cuando hay más de una superficie solapada, se almacena una lista enlazada de datos de superficie, como en (b).

- componentes de intensidad RGB
- parámetro de opacidad (porcentaje de transparencia)
- profundidad
- porcentaje de recubrimiento del área
- identificador de la superficie
- otros parámetros de representación de la superficie

El esquema de detección de visibilidad mediante búfer A puede implementarse utilizando métodos similares a los del algoritmo del búfer de profundidad. Las líneas de barrido se procesan para determinar cuánta parte de cada superficie cubre cada posición de píxel en las líneas de barrido individuales. Las superficies se subdividen en mallas poligonales y se recortan de acuerdo con los contornos del píxel. Utilizando los factores de opacidad y el porcentaje de recubrimiento de la superficie, los algoritmos de representación calculan el color de cada píxel como una media de las contribuciones de todas las superficies solapadas.

## 9.5 MÉTODO DE LA LÍNEA DE BARRIDO

---

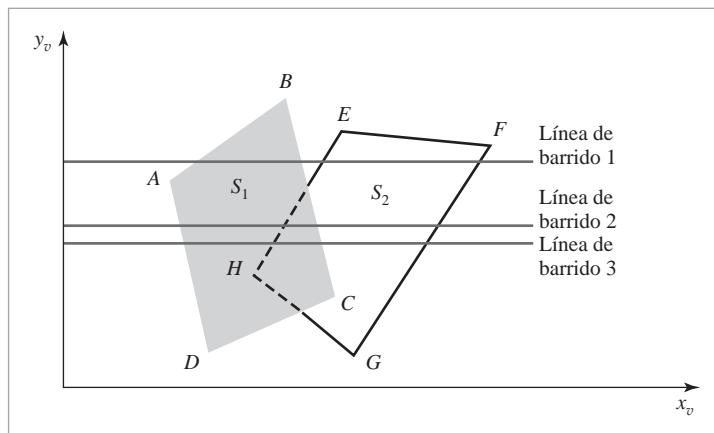
Este método del espacio de imagen para la identificación de superficies visibles calcula y compara los valores de profundidad a lo largo de las diversas líneas de barrido de una escena. A medida que se procesa cada línea de barrido, se examinan todas las proyecciones de superficies poligonales que intersectan dicha línea para determinar cuáles son visibles. A lo largo de cada línea de barrido, se realizan cálculos de profundidad para determinar qué superficie está más próxima al plano de visualización en cada posición de píxel. Una vez determinada la superficie visible para un píxel, se introduce el color de superficie correspondiente a dicha posición en el búfer de imagen.

Las superficies se procesan utilizando la información almacenada en la tabla de polígonos (Sección 3.15). La tabla de aristas contiene las coordenadas de los extremos de cada línea en la escena, la inversa de la pendiente de cada línea y punteros a la tabla de caras de la superficie con el fin de identificar las superficies delimitadas por cada línea. La tabla de caras de la superficie contiene los coeficientes del plano, las propiedades del material de la superficie, otros datos de la superficie y posiblemente punteros a la tabla de aristas. Para facilitar la búsqueda de las superficies que cruzan una determinada línea de barrido, se forma una lista de aristas activas para cada línea de barrido a medida que ésta es procesada. La lista de aristas activas contiene únicamente aquellas aristas que cruzan la línea de barrido actual, ordenadas en sentido ascendente según la coordenada  $x$ . Además, se define un indicador para cada superficie que se configura como «on» u «off» para indicar si una posición de una línea de barrido está dentro o fuera de la superficie. Las posiciones de píxel de cada línea de barrido se procesan de izquierda a derecha. En la intersección de la izquierda con la proyección de un polígono convexo, se pone a «on» el indicador de la superficie, poniéndose luego a «off» en el punto de intersección derecho a lo largo de la línea de barrido. Para un polígono cóncavo, las intersecciones de la línea de barrido pueden ordenarse de derecha a izquierda, poniendo el indicador de superficie a «on» entre cada par de intersecciones.

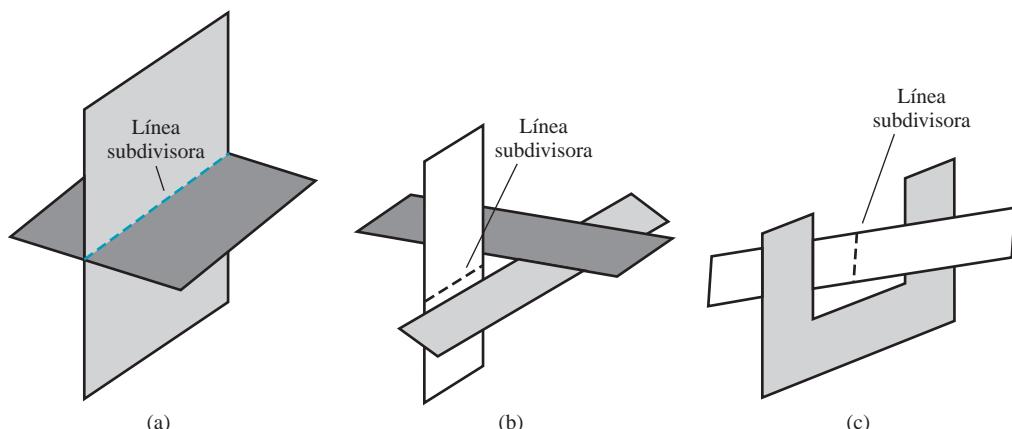
La Figura 9.10 ilustra el método de la línea de barrido para la localización de las partes visibles de las superficies para cada posición de píxel de una línea de barrido. La lista activa para la línea de barrido 1 contiene información extraída de la tabla de aristas y correspondiente a las aristas AB, BC, EH y FG. Para las posiciones a lo largo de esta línea de barrido entre las aristas AB y BC, sólo estará activado el indicador de la superficie  $S_1$ . Por tanto, no hacen falta cálculos de profundidad y los valores de color se determinan a partir de las propiedades y de las condiciones de iluminación de la superficie  $S_1$ . De forma similar, entre las aristas EH y FG sólo estará activado el indicador de la superficie  $S_2$ . No hay ninguna otra posición de píxel a lo largo de la línea de barrido 1 que intersecte ninguna superficie, por lo que el color de dichos píxeles será el color de fondo, que puede cargarse en el búfer de imagen como parte de la rutina de inicialización.

Para las líneas de barrido 2 y 3 de la Figura 9.10, la lista de aristas activas contiene las aristas AD, EH, BC y FG. A lo largo de la línea de barrido 2, entre las aristas AD y EH sólo está activado el indicador de la superficie  $S_1$ , pero entre las aristas EH y BC, están activados los indicadores de ambas superficies. Por tanto, es necesario realizar un cálculo de profundidad cuando nos encontramos la vista EH, utilizando los coeficientes de los respectivos planos de las dos superficies. Para este ejemplo, asumimos que la profundidad de la superficie  $S_1$  es menor que la de  $S_2$ , por lo que se asignarán los valores de color de la superficie  $S_1$  a todos los píxeles de la línea de barrido hasta encontrar la arista BC. Entonces, el indicador de superficie de  $S_1$  se desactiva, almacenándose los colores correspondientes a la superficie  $S_2$  hasta alcanzar la arista FG. Ya no hacen falta más cálculos de profundidad, porque asumimos que la superficie  $S_2$  permanece detrás de  $S_1$  una vez que hemos establecido la relación de profundidades en la arista EH.

Podemos aprovechar la coherencia entre líneas de barrido al pasar de una línea de barrido a la siguiente. En la Figura 9.10, la línea de barrido 3 tiene la misma lista activa que la línea de barrido 2. Puesto que no se han producido cambios en las intersecciones de las líneas, vuelve a ser innecesario realizar cálculos de profundidad entre las aristas EH y BC. Las dos superficies deben estar en la misma orientación que se ha determinado en la línea de barrido 2, por lo que pueden introducirse los colores de la superficie  $S_1$  sin necesidad de cálculos adicionales de profundidad.



**FIGURA 9.10.** Líneas de barrido que cruzan la proyección de dos superficies  $S_1$  y  $S_2$  sobre el plano de visualización. Las líneas punteadas indican los contornos de secciones ocultas de una superficie.



**FIGURA 9.11.** Superficies solapadas que se intersectan o que se superponen de forma cíclica y que se ocultan alternativamente unas a otras.

Con este método de la línea de barrido puede procesarse cualquier número de superficies poligonales solapadas. Los indicadores correspondientes a las superficies se activan «on» para indicar si una posición está dentro o fuera y sólo se realizan cálculos de profundidad en las aristas de las superficies solapadas. Este procedimiento funciona correctamente sólo si las superficies no se cortan y si no se solapan cíclicamente de alguna forma (Figura 9.11). Si en una escena se produce algún tipo de solapamiento cíclico, podemos dividir la superficie para eliminar dichos solapamientos. Las líneas punteadas de la figura indican dónde podrían subdividirse los planos para formar dos superficies diferentes, con el fin de eliminar los solapamientos cílicos.

## 9.6 MÉTODO DE ORDENACIÓN DE LA PROFUNDIDAD

---

Utilizando operaciones tanto en el espacio de imagen como en el espacio de objetos, el método de **ordenación de la profundidad** lleva a cabo las siguientes funciones básicas:

- (1) Se ordenan las superficies en orden decreciente de profundidades.
- (2) Se digitalizan las superficies por orden, comenzando por la superficie de mayor profundidad.

Las operaciones de ordenación se llevan a cabo tanto en el espacio de imagen como en el espacio de objetos, y la digitalización de las superficies de los polígonos se realiza en el espacio de imagen.

Este método de detección de visibilidad se denomina a menudo **algoritmo del pintor**. Al dibujar una acuarela o un óleo, el artista pinta primero los colores de fondo. A continuación, añade los objetos más distantes y luego los más próximos. En el paso final, se pinta el primer plano sobre el fondo y sobre los objetos más distantes. Cada capa de color cubre la capa anterior. Utilizando una técnica similar, primero ordenamos las superficies de acuerdo con su distancia respecto al plano de visualización. Los valores de color de la superficie más lejana pueden entonces introducirse en el búfer de refresco. Si procesamos cada superficie sucesiva por turno (en orden de profundidad decreciente), estaremos «pintando» la superficie en el búfer de imagen sobre los colores de las superficies previamente procesadas.

El almacenamiento de los colores de las superficies de los polígonos en el búfer de imagen de acuerdo con la profundidad se lleva a cabo en varios pasos. Suponiendo que estemos visualizando la escena según la dirección  $z$ , las superficies se ordenan en la primera pasada de acuerdo con el valor de  $z$  más pequeño de cada superficie. La superficie  $S$  situada al final de la lista (es decir, la superficie con mayor profundidad) se compara entonces con las otras superficies de la lista para ver si hay solapamientos de profundidad. Si no es así,  $S$  es la superficie más distante y se procede a digitalizarla. La Figura 9.12 muestra dos superficies que se solapan en el plano  $xy$ , pero que no tienen solapamiento de profundidad. Este proceso se repite a continuación para la siguiente superficie de la lista. Mientras que no haya solapamientos, se va procesando cada superficie por orden de profundidad hasta que se hayan digitalizado todas. Si se detecta un solapamiento de superficie en cualquier punto de la lista, será necesario efectuar algunas comparaciones adicionales para determinar si hay que reordenar alguna de las superficies.

Habrá que realizar los siguientes tests para cada superficie que tenga un solapamiento de profundidad con  $S$ . Si alguno de estos tests se cumple, no será necesario reordenar  $S$  y la superficie que esté siendo comprobada. Los tests se enumeran en orden de dificultad creciente:

- (1) Los rectángulos de contorno (extensiones de coordenadas) en las direcciones  $xy$  de las dos superficies no se solapan.
- (2) La superficie  $S$  está completamente detrás de la superficie solapada, en relación con la posición de visualización.
- (3) La superficie solapada está completamente delante de  $S$  en relación con la posición de visualización.
- (4) Las proyecciones de las aristas de contorno de las dos superficies sobre el plano de visualización no se solapan.

Estos test se realizan en el orden indicado y se salta a la siguiente superficie solapada en cuanto se detecte que alguno de los tests es cierto. Si todas las superficies solapadas pasan al menos uno de estos tests,  $S$  será

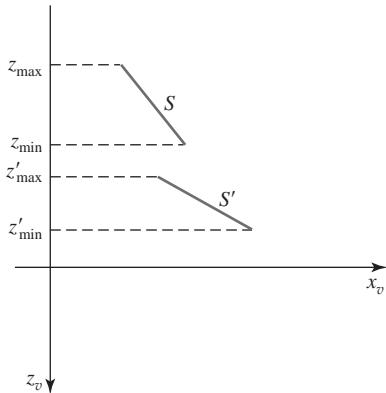
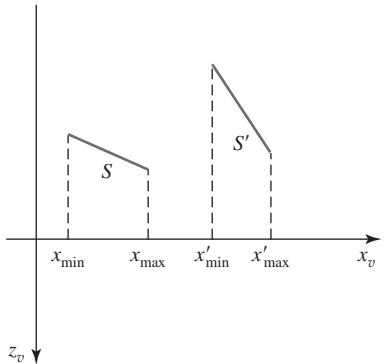
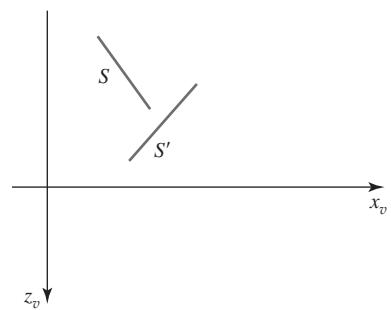


FIGURA 9.12. Dos superficies sin solapamiento de profundidad.

FIGURA 9.13. Dos superficies con solapamiento de profundidad pero que no se solapan en la dirección  $x$ .FIGURA 9.14. La superficie  $S$  está completamente detrás de la superficie solapada  $S'$ .

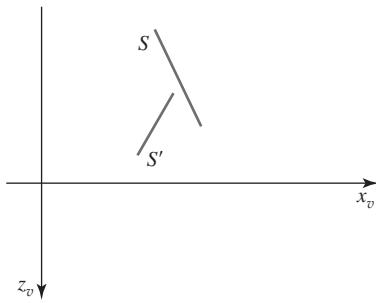
la superficie más distante, no siendo necesario efectuar ninguna reordenación y pudiendo por tanto digitalizar  $S$ .

El test 1 se realiza en dos partes. Primero se comprueba el solapamiento en la dirección  $x$  y luego en la dirección  $y$ . Si no hay solapamiento de las superficies en ninguna de estas direcciones, los dos planos no pueden ocultarse el uno al otro. En la Figura 9.13 se muestra un ejemplo de dos superficies que se solapan en la dirección  $z$  pero no en la dirección  $x$ .

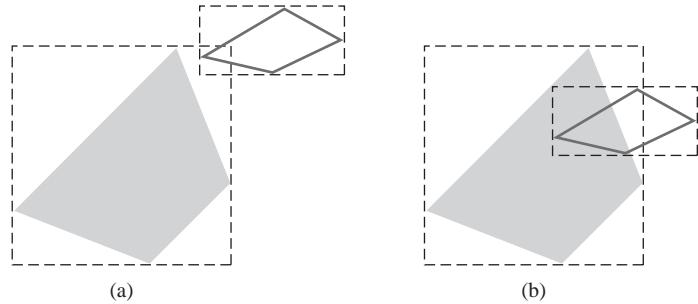
Podemos realizar los tests 2 y 3 utilizando los tests de polígonos posterior-frontal. En otras palabras, podemos sustituir las coordenadas de todos los vértices de  $S$  en la ecuación del plano de la superficie solapada y verificar el signo del resultado. Si las ecuaciones del plano están especificadas de modo que la parte frontal de la superficie apunte hacia la posición de visualización, entonces  $S$  estará detrás de  $S'$  si todos los vértices de  $S$  están en la parte posterior de  $S'$  (Figura 9.14). De forma similar,  $S'$  estará completamente delante de  $S$  si todos los vértices de  $S'$  se encuentran delante de  $S$ . La Figura 9.15 muestra una superficie solapada  $S'$  que está completamente delante de  $S$ , aunque la superficie  $S$  no está completamente detrás de  $S'$  (el test 2 no da un resultado verdadero).

Si fallaran los tests 1 a 3, realizaremos el test 4 para determinar si se solapan las proyecciones de las dos superficies. Como se ilustra en la Figura 9.16, dos superficies pueden o no intersectarse aún cuando sus extensiones de coordenadas se solapen.

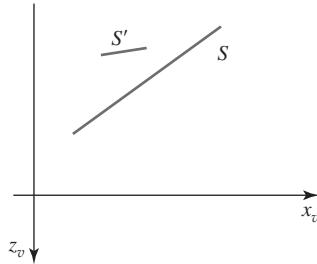
Si los cuatro tests fallan para una superficie solapada  $S'$ , intercambiaremos las superficies  $S$  y  $S'$  en la lista ordenada. En la Figura 9.17 se proporciona un ejemplo de dos superficies que serían reordenadas según este procedimiento. En este punto, todavía no sabemos a ciencia cierta si hemos encontrado la superficie más



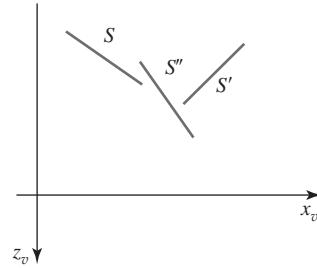
**FIGURA 9.15.** La superficie solapada  $S'$  está completamente delante de la superficie  $S$ , pero ésta no está completamente detrás de  $S'$ .



**FIGURA 9.16.** Dos superficies poligonales con rectángulos de contorno no solapados en el plano  $xy$ .



**FIGURA 9.17.** La superficie  $S$  llega hasta una profundidad mayor, pero tapa completamente a la superficie  $S'$ .



**FIGURA 9.18.** Tres superficies que han sido introducidas en la lista ordenada de superficies con el orden  $S$ ,  $S''$ ,  $S'$  y que deben reordenarse como  $S'$ ,  $S''$ ,  $S$ .

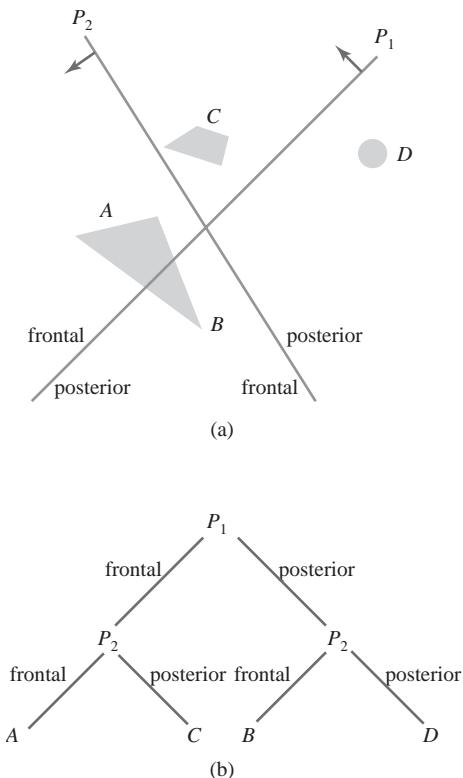
alejada del plano de visualización. La Figura 9.18 ilustra una situación en la que primero intercambiaríamos  $S$  y  $S''$ . Pero como  $S''$  oculta parte de  $S'$ , será necesario intercambiar  $S''$  y  $S'$  para que las tres superficies queden en el orden correcto de profundidad. Por tanto, necesitamos repetir el proceso de comprobaciones para cada superficie que se reordene en la lista.

Resulta perfectamente posible que el algoritmo que acabamos de esbozar entre en un bucle infinito si hay dos o más superficies que se ocultan alternativamente la una a la otra, como en la Figura 9.11. En dicho caso, el algoritmo se dedicaría a reordenar continuamente las superficies solapadas. Para evitar tales bucles, podemos marcar toda superficie que haya sido reordenada a una posición de profundidad mayor, de modo que dicha superficie no pueda volver a ser desplazada. Si se hace un intento de reordenar la superficie una segunda vez, la dividiremos en dos partes para eliminar los solapamientos cíclicos. La superficie original se sustituye entonces por las dos nuevas superficies y el procesamiento continúa como antes.

## 9.7 MÉTODO DEL ÁRBOL BSP

Un árbol de **particionamiento del espacio binario** (BSP, binary space-partitioning) es un método eficiente para determinar la visibilidad de los objetos pintando las superficies en el búfer de imagen desde atrás hacia adelante, como en el algoritmo del pintor. El árbol BSP resulta particularmente útil cuando el punto de referencia de visualización cambia pero los objetos de la escena se encuentran en posiciones fijas.

Aplicar un árbol BSP a las comprobaciones de visibilidad implica identificar las superficies que se encuentren detrás o delante del plano de particionamiento en cada paso de subdivisión del espacio, con respecto a la



**FIGURA 9.19.** Una región del espacio (a) se partitiona con dos planos  $P_1$  y  $P_2$  para formar la representación en árbol BSP que se muestra en (b).

dirección de visualización. La Figura 9.19 ilustra el concepto básico de este algoritmo. Con el plano  $P_1$ , primero particionamos el espacio en dos conjuntos de objetos. Uno de los conjuntos de objetos se encuentra detrás del plano  $P_1$  en relación con la dirección de visualización, mientras que el otro conjunto se encuentra delante de  $P_1$ . Puesto que hay un objeto intersectado por el plano  $P_1$ , dividimos dicho objeto en dos objetos separados, etiquetados como  $A$  y  $B$ . Los objetos  $A$  y  $C$  están delante de  $P_1$ , mientras que los objetos  $B$  y  $D$  se encuentran detrás de  $P_1$ . A continuación, particionamos de nuevo el espacio en el plano  $P_2$  y construimos la representación en árbol binario que se muestra en la Figura 9.19(b). En este árbol, los objetos se representan como nodos terminales, ocupando los objetos frontales las ramas izquierdas y los objetos posteriores las ramas derechas.

Para los objetos descritos mediante caras poligonales, podemos hacer que los planos de particionamiento coincidan con planos de las superficies poligonales. Entonces se utilizan las ecuaciones de los polígonos para identificar los polígonos frontales y posteriores y el árbol se construye utilizando el plano de particionamiento para cada cara poligonal. Todo polígono intersectado por un plano de particionamiento será dividido en dos partes. Cuando el árbol BSP se complete, procesaremos el árbol seleccionando primero los nodos de la derecha y luego los nodos de la izquierda. Así, las superficies se procesan para su visualización comenzando por las frontales y siguiendo por las posteriores, por lo que los objetos de primer plano se pintan sobre los objetos de fondo. En algunos sistemas se utilizan implementaciones rápidas en hardware para construir y procesar árboles BSP.

## 9.8 MÉTODO DE LA SUBDIVISIÓN DE ÁREAS

Esta técnica de eliminación de caras ocultas es esencialmente un método en el espacio de imagen, pero pueden utilizarse operaciones del espacio de objetos para realizar una ordenación de las superficies según su pro-

fundidad. El **método de la subdivisión de áreas** aprovecha la coherencia de las áreas de una escena, localizando las áreas de proyección que representan parte de una misma superficie. Aplicamos este método dividiendo sucesivamente el área total del plano de visualización en rectángulos de tamaño más pequeño, hasta que cada área rectangular: (1) sólo contenga la proyección de una parte de una única superficie visible, (2) no contenga proyecciones de ninguna superficie o (3) el área se haya reducido al tamaño de un píxel.

Para implementar este método, tenemos que definir tests que permitan identificar rápidamente el área como parte de una misma superficie o que nos digan que el área es demasiado compleja como para analizarla fácilmente. Comenzando con el área total, aplicamos los tests para determinar si debemos subdividir dicha área en rectángulos más pequeños. Si los tests indican que la vista es lo suficientemente compleja, la subdividimos. A continuación, aplicamos los tests a cada una de las áreas más pequeñas, subdividiéndolas si los tests indican que la condición de visibilidad de una única superficie sigue siendo incierta. Continuamos con este proceso hasta que se pueda analizar fácilmente las subdivisiones como pertenecientes a una única superficie o hasta que hayamos alcanzado el límite de resolución. Una forma fácil de hacer esto consiste en dividir sucesivamente el área en cuatro partes iguales en cada caso, como se muestra en la Figura 9.20. Esta técnica es similar a la que se emplea para construir un árbol cuádrico. Un área de visualización con una resolución en píxeles de 1024 por 1024 podría subdividirse diez veces de esta forma antes de que una subárea se redujera al tamaño de un único píxel.

Hay cuatro posibles relaciones que una superficie puede tener con una de las áreas del plano de visualización subdividido. Podemos describir estas posiciones relativas de la superficie utilizando las siguientes clasificaciones (Figura 9.21).

**Superficie circundante:** una superficie que encierra completamente el área.

**Superficie solapada:** una superficie que está parcialmente dentro y parcialmente fuera del área.

**Superficie interior:** una superficie que está completamente dentro del área.

**Superficie exterior:** una superficie que está completamente fuera del área.

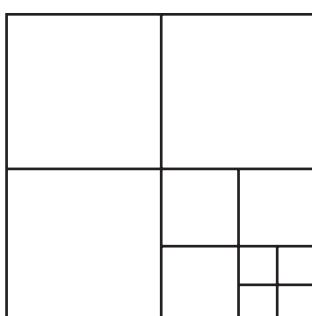
Las pruebas para determinar la visibilidad de superficies dentro de un área rectangular pueden enunciarse en términos de las cuatro clasificaciones de superficies ilustradas en la Figura 9.21. No será necesaria realizar ninguna subdivisión adicional de un área especificada si se cumple alguna de las siguientes condiciones.

**Condición 1:** un área no tiene superficies interiores, solapadas o circundantes (todas las superficies están fuera del área).

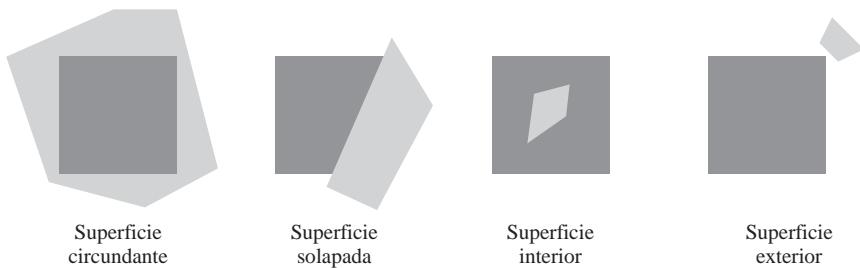
**Condición 2:** un área sólo tiene una superficie interior, solapada o circundante.

**Condición 3:** un área tiene una superficie circundante que oculta todas las demás superficies que caen dentro de los límites del área.

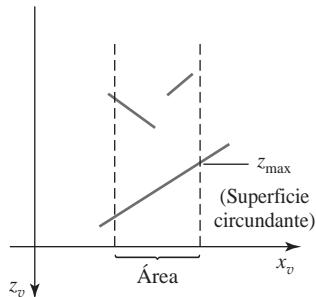
Inicialmente, podemos comparar las extensiones de coordenadas de cada superficie con el contorno del área. Esto nos permitirá identificar las superficies interiores y circundantes, pero las superficies solapadas y exteriores requieren usualmente tests de intersección. Si un único rectángulo de contorno intersecta el área de alguna forma, se utilizan comprobaciones adicionales para determinar si la superficie es circundante, solapa-



**FIGURA 9.20.** División de un área cuadrada en cuadrantes del mismo tamaño en cada paso.



**FIGURA 9.21.** Posibles relaciones entre las superficies poligonales y una sección rectangular del plano de visualización.



**FIGURA 9.22.** Dentro de un área especificada, una superficie circundante con una profundidad máxima  $z_{\max}$  oculta todas las demás superficies que tienen una profundidad mínima mayor que  $z_{\max}$ .

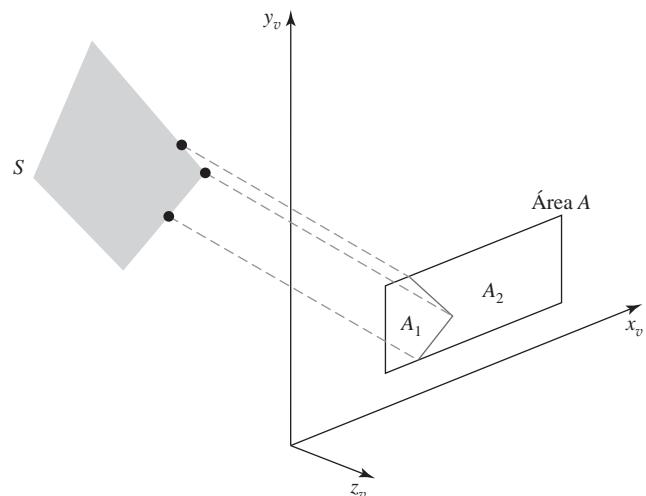
da o exterior. Una vez identificada una única superficie interior, solapada o circundante. Se almacenan los valores de color de la superficie en el búfer de imagen.

Un método para comprobar la condición 3 consiste en ordenar las superficies de acuerdo con su profundidad mínima con respecto al plano de visualización. Para cada superficie circundante, calculamos entonces la profundidad máxima dentro del área que estamos considerando. Si la profundidad máxima de una de estas superficies circundantes está más cerca del plano de visualización que la profundidad mínima de todas las otras superficies dentro del área, la condición 3 se satisfará. La Figura 9.22 ilustra esta situación.

Otro método para comprobar la condición 3 que no requiere una ordenación según las profundidades consiste en utilizar las ecuaciones de los planos para calcular los valores de profundidad en los cuatro vértices del área para todas las superficies circundantes, solapadas e interiores. Si los cuatro valores de profundidad para una de las superficies circundantes son menores que las profundidades calculadas para las otras superficies, se satisfará la condición 3. Entonces, puede mostrarse ese área con los colores correspondientes a dicha superficie circundante.

En algunas situaciones, los dos métodos anteriores de comprobación pueden no identificar correctamente una superficie circundante que oculte a todas las otras superficies. Pueden llevarse a cabo comprobaciones adicionales para identificar esa única superficie que cubre el área, pero resulta más rápido subdividir el área que continuar realizando comprobaciones más complejas. Una vez identificada una superficie como exterior o circundante para un área, seguirá siendo exterior o circundante para todas las subdivisiones de dicha área. Además, podemos esperar que se eliminen algunas superficies interiores y solapadas a medida que continúa el proceso de subdivisión, por lo que las áreas serán más fáciles de analizar. En el caso límite, cuando se produce una subdivisión del tamaño de un píxel, simplemente se calcula la profundidad de cada superficie relevante en dicho punto y se le asigna al píxel el color de la superficie más próxima.

Como variación del proceso de subdivisión básico, podríamos subdividir las áreas según los contornos de la superficie, en lugar de dividirlas por la mitad. Si se han ordenado las superficies de acuerdo con su profundidad mínima, podemos utilizar la superficie con valor de profundidad más pequeño para subdividir un área determinada. La Figura 9.23 ilustra este método de subdivisión de áreas. Se utiliza la proyección del contorno de la superficie  $S$  para partitionar el área original en las subdivisiones  $A_1$  y  $A_2$ . Entonces, la superficie  $S$  será una superficie circundante para  $A_1$  y pueden comprobarse las condiciones de visibilidad 2 y 3 para deter-



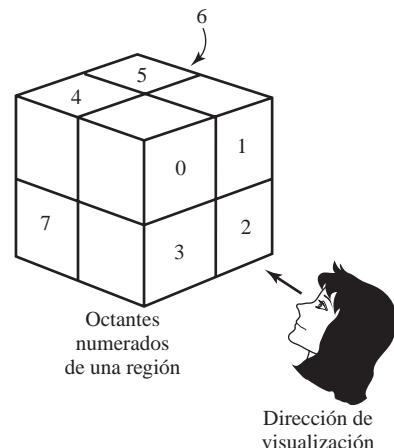
**FIGURA 9.23.** El área  $A$  se subdivide en  $A_1$  y  $A_2$  utilizando el contorno de la superficie  $S$  sobre el plano de visualización.

minar si es necesario efectuar subdivisiones adicionales. En general, se necesitan menos subdivisiones utilizando esta técnica, pero hace falta una mayor cantidad de procesamiento para subdividir las áreas y para analizar la relación de las superficies con los contornos de subdivisión.

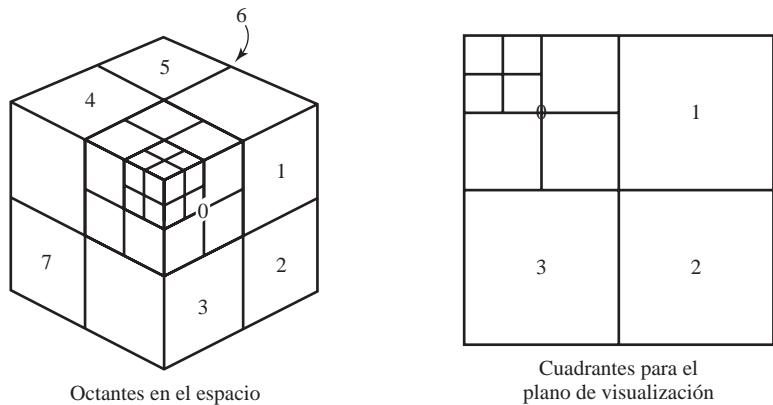
## 9.9 MÉTODOS DE ÁRBOLES OCTALES

Cuando se utiliza una representación en árbol octal para el volumen de visualización, la identificación de las superficies visibles se lleva a cabo explorando los nodos del árbol octal en orden de parte frontal a parte trasera. En la Figura 9.24, el primer plano de una escena está contenido en los octantes 0, 1, 2 y 3. Las superficies en la parte frontal de estos octantes son visibles para el observador. Las superficies situadas en la parte posterior de los octantes frontales o en los octantes posteriores (4, 5, 6 y 7) pueden estar ocultas por las superficies frontales.

Podemos procesar los nodos del árbol octal de la Figura 9.24 en el orden 0, 1, 2, 3, 4, 5, 6, 7. Esto da como resultado un recorrido del árbol octal que se realiza primero según el orden de profundidad, visitándose los nodos de los cuatro suboctantes frontales del octante 0 antes que los nodos de los cuatro suboctantes posteriores. El recorrido del árbol octal continúa en este orden para cada subdivisión de los octantes.



**FIGURA 9.24.** Los objetos de los octantes 0, 1, 2 y 3 ocultan a los objetos de los octantes posteriores (4, 5, 6, 7) cuando la dirección de visualización es como se muestra.



**FIGURA 9.25.** División en octantes para una región del espacio y el correspondiente plano de cuadrantes.

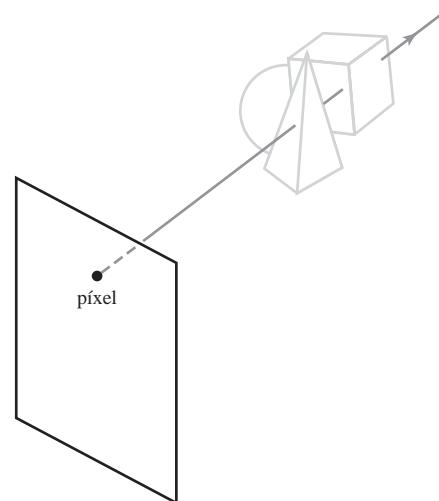
Cuando se encuentra un valor de color en un nodo de árbol octal, dicho color se guarda en el árbol cuádrico únicamente si no se ha almacenado previamente ningún valor para la misma área. De esta forma, sólo se almacenarán los colores frontales. Los nodos que tienen el valor “void” se ignoran. Cualquier nodo que esté completamente oculto se eliminará de los ulteriores procesamientos, de modo que no se accederá a sus subárboles. La Figura 9.25 muestra los octantes en una región del espacio y los correspondientes cuadrantes en el plano de visualización. Las contribuciones al cuadrante 0 vienen de los cuadrantes 0 y 4. Los valores de color del cuadrante 1 se obtienen a partir de las superficies de los octantes 1 y 5 y los valores de cada uno de los otros dos cuadrantes se generan a partir de las parejas de octantes alineadas con dichos cuadrantes.

Las comprobaciones de visibilidad mediante árbol octal se llevan a cabo mediante procesamiento recursivo de los nodos del árbol y mediante la creación de una representación en árbol cuádrico para las superficies visibles. Para la mayoría de los casos, es necesario tener en cuenta tanto el octante frontal como el posterior a la hora de determinar los valores de color correctos para un cuadrante. Pero si el octante frontal está homogéneamente relleno con un cierto color, no es necesario procesar el octante posterior. Para regiones heterogéneas, se invoca a un procedimiento recursivo, pasándole como nuevos argumentos el hijo del octante heterogéneo y un nuevo nodo del árbol cuádrico recién creado. Si el frontal está vacío, sólo será necesario procesar el hijo del octante posterior. En caso contrario, se realizan dos llamadas recursivas, una para el octante posterior y otra para el octante frontal.

Puede obtenerse diferentes vistas de objetos representados como árboles octales aplicando transformaciones a la representación en árbol que hagan que se reoriente el objeto de acuerdo con la vista seleccionada. Los octantes pueden entonces renombrarse de modo que la representación en árbol octal esté siempre organizada con los octantes 0, 1, 2 y 3 en la cara frontal.

## 9.10 MÉTODO DE PROYECCIÓN DE RAYOS

Si consideramos la línea de visión que atraviesa una escena partiendo de una posición de píxel en el plano de visualización, como en la Figura 9.26, podemos determinar qué objetos de una escena intersectan dicha línea (si es que hay alguna). Después de calcular todas las intersecciones entre el rayo y las superficies, identificaremos la superficie visible como aquella cuyo punto de intersección esté más próximo al píxel. Este esquema de detección de visibilidad utiliza procedimientos de *proyección de rayos* que ya hemos presentado en la Sección 8.20. La proyección de rayos, como herramienta de detección de visibilidad, está basada en métodos de óptica geométrica que trazan los trayectos de los rayos luminosos. Puesto que hay un número infinito de rayos luminosos en una escena y sólo nos interesan aquellos que pasan a través de las posiciones de los píxe-



**FIGURA 9.26.** Un rayo trazado a lo largo de la línea de visión que atraviesa una escena partiendo de una posición de píxel.

les, podemos trazar los trayectos de los rayos luminosos hacia atrás a partir de los píxeles, atravesando la escena. La técnica del proyección de rayos es un método eficiente de detección de visibilidad para escenas que tengan superficies curvadas, particularmente esferas.

Podemos pensar en la proyección de rayos como en una variante del método del búfer de profundidad (Sección 9.3). En el algoritmo del búfer de profundidad, procesamos las superficies de una en una y calculamos los valores de profundidad para todos los puntos de proyección de la superficie. Las profundidades de superficie calculadas se comparan entonces con las profundidades previamente almacenadas para determinar qué superficie es visible en cada píxel. En el trazado de rayos, procesamos los píxeles de uno en uno y calculamos las profundidades para todas las superficies a lo largo del trayecto de proyección que va hasta dicho píxel.

La proyección de rayos es un caso especial de los algoritmos de *trazado de rayos* (Sección 10.11) que trazan múltiples trayectos de rayos para recopilar las contribuciones de refracción y reflexión globales debidas a múltiples objetos de la escena. Con la proyección de rayos, lo único que hacemos es seguir un rayo desde cada píxel hasta el objeto más cercano. Se han desarrollado técnicas muy eficientes de cálculo de intersecciones entre rayos y superficies para objetos comunes, particularmente esferas, y hablaremos en detalle de estos métodos de cálculo de intersecciones en la Sección 10.11.

## 9.11 COMPARACIÓN DE LOS MÉTODOS DE DETECCIÓN DE VISIBILIDAD

---

La efectividad de un método de detección de superficies visibles depende de las características de cada aplicación concreta. Si las superficies de una escena están ampliamente distribuidas a lo largo de la dirección de visualización, por lo que hay muy poco solapamiento en profundidad, lo más eficiente suele ser utilizar un algoritmo de ordenación de profundidad o de árbol BSP. Cuando hay pocos solapamientos de las proyecciones de las superficies sobre el plano de visualización, los algoritmos de líneas de barrido o de subdivisión de áreas constituyen una forma rápida de localizar las superficies visibles.

Como regla general, el algoritmo de ordenación de profundidades o el método del árbol BSP constituyen técnicas altamente efectivas para aquellas escenas que sólo tengan unas pocas superficies. Esto se debe a que dichas escenas suelen tener pocas superficies que se solapen en profundidad. El método de la línea de barrido también funciona bien cuando una escena contiene un pequeño número de superficies. Podemos utilizar los métodos de la línea de barrido, de ordenación de profundidades o del árbol BSP para identificar las superficies visibles de manera efectiva en escenas que tengan hasta unos pocos miles de superficies poligonales. Con escenas que contengan un número mayor de superficies, los métodos más adecuados son el del búfer de

profundidad o el del árbol octal. El método del búfer de profundidad tiene un tiempo de procesamiento prácticamente constante e independiente del número de superficies de una escena. Esto se debe a que el tamaño de las áreas de superficie decrece a medida que el número de superficies de la escena se incrementa. Por tanto, el método del búfer de profundidad tiene un rendimiento relativamente bajo para escenas simples y relativamente alto para escenas complejas. Los árboles BSP son útiles cuando hay que generar múltiples vistas utilizando diferentes puntos de referencia de visualización. Si una escena contiene superficies curvas, podemos utilizar los métodos del árbol octal o de proyección de rayos para identificar las partes visibles de la escena.

Cuando se utilizan representaciones en árbol octal en un sistema, el proceso de detección de visibilidad es rápido y simple. Sólo se utilizan sumas y restas enteras en el proceso y no hay necesidad de realizar ordenaciones ni cálculos de intersecciones. Otra ventaja de los árboles octales es que almacenan más información que simplemente la geometría de la superficie. Tenemos disponible toda la región sólida de un objeto para la visualización, lo que hace que la representación de un árbol octal sea útil para obtener secciones transversales de objetos tridimensionales.

Resulta posible combinar e implementar los diferentes métodos de detección de superficies visibles en diversas formas. Además, los algoritmos de detección de visibilidad se suelen implementar en hardware, utilizando sistemas especiales con procesamiento paralelo para incrementar la eficiencia de estos métodos. Los sistemas hardware especiales se utilizan cuando la velocidad de procesamiento es una consideración de especial importancia, como en el caso de la generación de imágenes animadas para simuladores de vuelo.

## 9.12 SUPERFICIES CURVAS

---

Los métodos más eficientes para determinar la visibilidad de objetos con superficies curvas son la proyección de rayos y los métodos basados en árbol octal. Con la proyección de rayos, calculamos las intersecciones entre los rayos y las superficies y localizamos la distancia de intersección más pequeña a lo largo del trayecto del rayo. Con los árboles octales, simplemente exploramos los nodos de adelante hacia atrás para localizar los valores de color de superficie. Una vez definida una representación en un árbol octal a partir de las definiciones de entrada de los objetos, todas las superficies visibles se identifican con el mismo tipo de procesamiento. No es necesario realizar ningún tipo especial de consideración para diferentes tipos de superficies, ya sean curvas o de cualquier otra clase.

Una superficie curva también puede aproximarse mediante una malla poligonal, y entonces podemos utilizar algunos de los métodos de identificación de superficies visibles previamente expuestos. Pero para algunos objetos, como las esferas, puede que sea más eficiente, además de más preciso utilizar el método de proyección de rayos y las ecuaciones que describen la superficie curva.

### Representación de superficies curvas

Podemos representar una superficie como una ecuación implícita de la forma  $f(x, y, z) = 0$  o con una representación paramétrica (Apéndice A). Las superficies de tipo *spline*, por ejemplo, se suelen describir mediante ecuaciones paramétricas. En algunos casos, resulta útil obtener una ecuación explícita de la superficie, como por ejemplo una ecuación que nos de la altura con respecto a un plano de tierra  $xy$ :

$$z = f(x, y)$$

Muchos objetos de interés, como las esferas, elipsoides, cilindros y conos tienen representaciones mediante ecuaciones cuadráticas. Estas superficies se suelen utilizar comúnmente para modelar estructuras moleculares, cojinete, anillos y ejes.

Los algoritmos de línea de barrido y de proyección de rayos requieren a menudo técnicas de aproximación numérica para resolver la ecuación de la superficie en el punto de intersección con una línea de barrido o con un rayo de un píxel. Se han desarrollado diversas técnicas, incluyendo cálculos en paralelo e implementaciones hardware de gran velocidad, para resolver las ecuaciones de intersección con superficies curvas para los objetos más comúnmente utilizados.

## Diagramas de contorno de superficies

Para muchas aplicaciones en Matemáticas, Física, Ingeniería y otros campos, resulta útil mostrar una función de superficie mediante un conjunto de líneas de contorno que muestren la forma de la superficie. La superficie puede describirse mediante una ecuación o mediante tablas de datos, como por ejemplo datos topográficos sobre las elevaciones del terreno o datos de densidad de población. Con una representación funcional explícita, podemos dibujar las líneas de contorno de las superficies visibles y eliminar aquellas secciones de contorno que están ocultas por las partes visibles de la superficie.

Para obtener un diagrama  $xy$  de una superficie de una función, podemos escribir la representación de la superficie en la forma:

$$y = f(x, z) \quad (9.8)$$

Entonces, podemos dibujar una curva en el plano  $xy$  para los valores de  $z$  que caigan dentro del rango seleccionado, utilizando un intervalo especificado  $\Delta z$ . Comenzando con el valor mayor de  $z$ , dibujamos las curvas desde «atrás» hacia «adelante» y eliminamos las secciones ocultas. Las secciones curvas se dibujan en pantalla mapeando un rango  $xy$  de la función sobre un rango de píxeles  $xy$  de la pantalla. Entonces, tomamos incrementos unitarios en  $x$  y determinamos el correspondiente valor  $y$  para cada valor de  $x$  aplicando la Ecuación 9.8 para un valor dado de  $z$ .

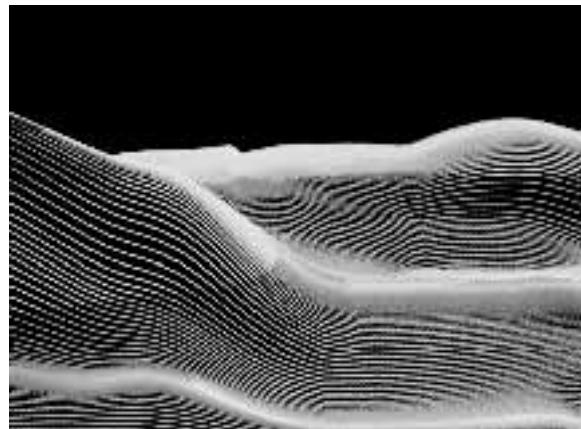
Una forma de identificar las secciones de curvas visibles de las superficies consiste en mantener una lista de valores  $y_{\min}$  y  $y_{\max}$  previamente calculados para las coordenadas  $x$  de la pantalla. Al pasar de una posición de píxel  $x$  a la siguiente, comparamos el valor  $y$  calculado con el rango almacenado  $y_{\min}$  y  $y_{\max}$  para el siguiente píxel. Si  $y_{\min} \leq y \leq y_{\max}$ , dicho punto de la superficie no es visible y no lo dibujaremos. Pero si el valor  $y$  calculado cae fuera de los límites de  $y$  almacenados para dicho píxel, el punto será visible. Entonces, dibujamos el punto  $y$  y asignamos los nuevos límites para dicho píxel. Pueden utilizarse procedimientos similares para proyectar la gráfica de contorno sobre el plano  $xz$  o  $yz$ . La Figura 9.27 muestra un ejemplo de diagrama de contorno de superficie, con líneas de contorno para las que se ha empleado una codificación de colores.

Podemos aplicar los mismos métodos a un conjunto discreto de puntos de datos determinando las líneas isosuperficiales. Por ejemplo, si tenemos un conjunto discreto de valores  $z$  para una cuadrícula  $n_x$  por  $n_y$  de valores  $xy$ , podemos determinar el trayecto correspondiente a una línea de  $z$  constante sobre la superficie utilizando los métodos de contorno explicados en la Sección 8.27. Cada línea de contorno seleccionada puede entonces proyectarse sobre un plano de visualización y mostrarse mediante segmentos lineales. De nuevo, las líneas pueden dibujarse sobre la pantalla en orden de profundidad delante-detrás, y eliminaremos las secciones de contorno que pasen por detrás de otras líneas de contorno previamente dibujadas (visibles).

## 9.13 MÉTODOS DE VISIBILIDAD PARA IMÁGENES ALÁMBRICAS

---

Las escenas no suelen contener secciones de línea aisladas, a menos que estemos mostrando un grafo, un diagrama o un esquema de una red. Pero a menudo surge la necesidad de ver una escena tridimensional en esbozo, con el fin de hacerse una idea rápida de las características de los objetos. La forma más rápida de generar una vista alámbrica de una escena consiste en mostrar los bordes de todos los objetos. Sin embargo, puede resultar difícil determinar la posición frontal o posterior de los objetos en ese tipo de imagen. Una solución a este problema consiste en aplicar técnicas de regulación de la intensidad del color según la profundidad, de modo que la intensidad mostrada de una línea esté en función de la distancia con respecto al observador. Alternativamente, podemos aplicar tests de visibilidad, de modo que las secciones de línea oculta se eliminan o se muestren con unas propiedades distintas de las de las aristas visibles. Los procedimientos para determinar la visibilidad de las aristas de los objetos se denominan **métodos de visibilidad alámbrica**. También se llaman **métodos de detección de líneas visibles** o **métodos de detección de líneas ocultas**. Además también pueden usarse algunos de los métodos de determinación de superficies visibles analizados en la sección anterior para comprobar la visibilidad de las aristas.



**FIGURA 9.27.** Una gráfica de contorno de superficie con codificación de colores. (Cortesía de Los Alamos National Laboratory.)

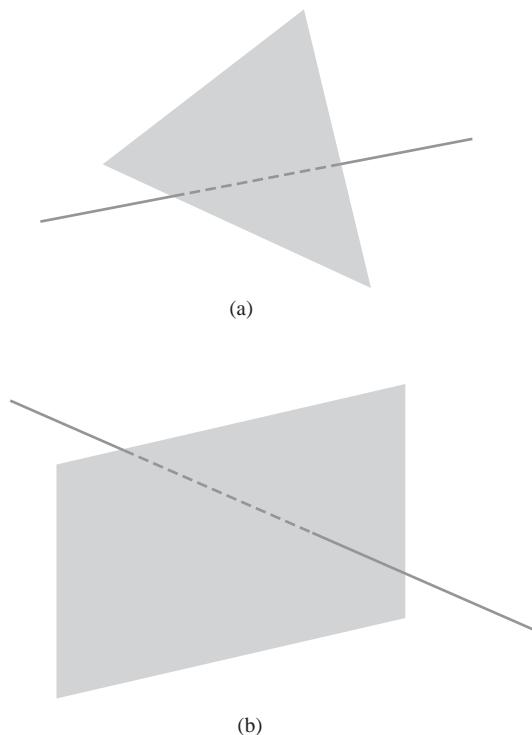
### Algoritmos de visibilidad de superficies para representaciones alámbricas

Una técnica directa para identificar las secciones de línea visibles consiste en comparar las posiciones de las aristas con las posiciones de las superficies en una escena. Este proceso implica la utilización de los mismos métodos que se usan en los algoritmos de recorte de líneas. Es decir, comprobamos la posición de los extremos de la línea con respecto al contorno de un área especificada pero en el caso de la comprobación de visibilidad también necesitamos comparar los valores de profundidad de la arista y de la superficie. Cuando ambos extremos proyectados de un segmento de línea caen dentro del área proyectada de una superficie, comparamos la profundidad de los extremos con la de la superficie en dichas posiciones ( $x, y$ ). Si ambos extremos están detrás de la superficie, se trata de una arista oculta. Si ambos extremos están delante de la superficie, la arista será visible con respecto a esa superficie. En caso contrario, debemos calcular los puntos de intersección y determinar los valores de profundidad en dichos puntos de intersección. Si la arista tiene una mayor facilidad que la superficie en las intersecciones correspondientes al perímetro, parte de la arista estará oculta por la superficie, como en la Figura 9.28(a). Otra posibilidad es que una arista tenga una mayor profundidad en la intersección con una de las líneas de contorno y una menor profundidad que la superficie en la intersección con otra línea de contorno (suponiendo que las superficies sean convexas). En dicho caso, tendremos que determinar si la arista penetra por el interior de la superficie, como en la Figura 9.28(b). Una vez identificada una sección oculta de una arista, podemos eliminarla, mostrarla en forma de línea punteada o utilizar alguna otra característica para distinguirla de las secciones visibles.

Algunos de los métodos de detección de superficies visibles pueden adaptarse fácilmente a las pruebas de visibilidad para visualización alámbrica de las aristas de los objetos. Utilizando un método de la cara posterior, podríamos identificar todas las superficies traseras de un objeto y mostrar únicamente los contornos de las superficies visibles. Con la ordenación de profundidad, podemos pintar las superficies en el búfer de refresco de modo que los interiores de las superficies estén en el color de fondo mientras que el contorno se dibuja en el color de primer plano. Procesando las superficies desde atrás hacia adelante, las líneas ocultas serán borradas por las superficies más próximas. Un método de subdivisión de áreas puede adaptarse para la eliminación de líneas ocultas mostrando únicamente los contornos de las superficies visibles. Finalmente, los métodos de líneas de barrido pueden utilizarse para mostrar las posiciones de intersección de las líneas de barrido con los contornos de las superficies visibles.

### Algoritmo de variación de intensidad con la profundidad para representaciones alámbricas

Otro método para mostrar la información de visibilidad consiste en variar el brillo de los objetos de una escena en función de su distancia con respecto a la posición de visualización. Este **método de variación de la intensidad con la profundidad** se suele aplicar utilizando la función lineal:



**FIGURA 9.28.** Secciones de línea ocultas (discontinuas) de una línea (a) que tiene mayor profundidad que una superficie y una línea (b) que está parcialmente detrás de una superficie y parcialmente delante de la misma.

$$f_{\text{depth}}(d) = \frac{d_{\max} - d}{d_{\max} - d_{\min}} \quad (9.9)$$

donde  $d$  es la distancia de un punto con respecto a la posición de visualización. Los valores de profundidad mínima y máxima,  $d_{\min}$  y  $d_{\max}$ , pueden especificarse con los valores que sean convenientes para cada especificación concreta. O bien, las profundidades mínima y máxima pueden ajustarse al rango de profundidad normalizado:  $d_{\min} = 0.0$  y  $d_{\max} = 1.0$ . A medida que se procesa cada posición de píxel, su color se multiplica por  $f_{\text{depth}}(d)$ . Así, los puntos más próximos se mostrarán con intensidades mayores y los puntos de profundidad máxima tendrán una intensidad igual a cero.

La función de variación de la intensidad puede implementarse con varias opciones. En algunas bibliotecas gráficas, hay disponible una función general de atmósfera (Sección 10.3) que puede combinar la variación de intensidad con otros efectos atmosféricos con el fin de simular humo o niebla, por ejemplo. Así, el color de un objeto podría verse modificado por la función de variación de intensidad con la distancia y luego combinarse con el color atmosférico.

## 9.14 FUNCIONES OpenGL DE DETECCIÓN DE VISIBILIDAD

Podemos aplicar a nuestras escenas tanto el método de eliminación de caras ocultas como las pruebas de visibilidad basadas en búfer  $z$  utilizando funciones incluidas en la biblioteca básica de OpenGL. Además, podemos utilizar funciones OpenGL para construir una imagen alámbrica de una escena en la que se hayan eliminado las líneas ocultas, o bien podemos mostrar las escenas con mecanismos de variación de la intensidad en función de la profundidad.

## Funciones OpenGL de eliminación de polígonos

La eliminación de caras posteriores se lleva a cabo mediante las funciones,

```
glEnable (GL_CULL_FACE);
glCullFace (mode);
```

donde al parámetro *mode* se le asigna el valor `GL_BACK`. De hecho, podemos utilizar esta función para eliminar en su lugar las caras frontales, o podríamos incluso eliminar tanto las caras frontales como las posteriores. Por ejemplo, si nuestra posición de visualización se encuentra dentro de un edificio, entonces lo que queremos es ver las caras posteriores (el interior de las habitaciones). En este caso, podríamos asignar al parámetro *mode* el valor `GL_FRONT` o podríamos cambiar la definición de qué cara de los polígonos es la frontal utilizando la función `glFrontFace` que se explica en la Sección 4.14. Entonces, si la posición de visualización se desplaza al exterior del edificio, podemos eliminar las caras posteriores de la imagen. Asimismo, en algunas aplicaciones, puede que sólo queramos ver otras primitivas dentro de la escena, como los conjuntos de puntos y los segmentos de líneas individuales. En este caso, para eliminar todas las superficies poligonales de una escena, asignaríamos al parámetro *mode* la constante simbólica OpenGL `GL_FRONT_AND_BACK`.

De manera predeterminada, el parámetro *mode* en la función `glCullFace` tiene el valor `GL_BACK`. Por tanto, si activamos la eliminación de caras posteriores mediante la función `glEnable` sin invocar explícitamente la función `glCullFace`, se eliminarán las caras posteriores de la escena. La rutina de eliminación se desactiva mediante,

```
glDisable (GL_CULL_FACE);
```

## Funciones OpenGL de gestión del búfer de profundidad

Para usar las rutinas OpenGL de detección de visibilidad mediante búfer de profundidad, primero necesitaremos modificar la función de inicialización GLUT para que el modo de visualización incluya una solicitud de configuración del búfer de profundidad, además de la del búfer de refresco. Podemos hacer esto, por ejemplo, con la instrucción,

```
glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
```

Los valores del búfer de profundidad pueden entonces inicializarse mediante,

```
glClear (GL_DEPTH_BUFFER_BIT);
```

Normalmente, el búfer de profundidad se inicializa con la misma instrucción que inicializa el búfer de refresco al color de fondo. Pero es necesario borrar el búfer de profundidad cada vez que queramos mostrar una nueva imagen. En OpenGL, los valores de profundidad están normalizados en el rango que va de 0 a 1.0, por lo que la inicialización indicada asignaría a todos los valores del búfer de profundidad el valor máximo 1.0 de manera predeterminada.

Las rutinas OpenGL de detección de visibilidad basada en búfer de profundidad se activan mediante la siguiente función:

```
glEnable (GL_DEPTH_TEST);
```

y se desactivan mediante,

```
glDisable (GL_DEPTH_TEST);
```

También podemos aplicar las comprobaciones de visibilidad basadas en búfer de profundidad utilizando algún otro valor inicial para la profundidad máxima, y este valor inicial se selecciona mediante la función OpenGL,

```
glClearDepth (maxDepth);
```

Al parámetro `maxDepth` se le puede asignar cualquier valor entre 0 y 1.0. Para cargar este valor de inicialización en el búfer de profundidad, debemos a continuación invocar la función `glClear (GL_DEPTH_BUFFER_BIT)`. En caso contrario, el búfer de profundidad se inicializará con el valor predeterminado (1.0). Puesto que los cálculos de color de superficie y otros tipos de procesamiento no se llevan a cabo para los objetos que se encuentren más allá de la profundidad máxima especificada, esta función puede usarse para acelerar las rutinas de búfer de profundidad cuando una escena contenga muchos objetos distantes que estén por detrás de los objetos de primer plano.

Las coordenadas de proyección en OpenGL están normalizadas en el rango que va de -1.0 a 1.0, y los valores de profundidad entre los planos de recorte próximo y lejano se normalizan al rango 0.0 a 1.0. El valor 0.0 se corresponde con el plano de recorte próximo (el plano de proyección), mientras que el valor 1.0 se corresponde con el plano de recorte lejano. Como opción, podemos ajustar estos valores de normalización con,

```
glDepthRange (nearNormDepth, farNormDepth);
```

De manera predeterminada `nearNormDepth = 0.0` y `farNormDepth = 1.0`, pero con la función `glDepthRange` podemos asignar a estos dos parámetros los valores que deseemos dentro del rango que va de 0.0 a 1.0, incluyendo valores para los que `nearNormDepth > farNormDepth`. Utilizando la función `glDepthRange`, podemos restringir las comprobaciones de búfer en profundidad a cualquier región del volumen de visualización, e incluso podemos invertir las posiciones de los planos próximo y lejano.

Otra opción disponible en OpenGL es la condición de prueba que hay que utilizar para las rutinas del búfer de profundidad. Especificamos una condición de prueba mediante la siguiente función:

```
glDepthFunc (testCondition);
```

Podemos asignar al parámetro `testCondition` cualquiera de las siguientes ocho constantes simbólicas: `GL_LESS`, `GL_GREATER`, `GL_EQUAL`, `GL_NOTEQUAL`, `GL_LEQUAL`, `GL_GEQUAL`, `GL_NEVER` (no se procesa ningún punto), `GL_ALWAYS` (se procesan todos los puntos). Estas diferentes pruebas pueden ser útiles en diversas aplicaciones, para reducir los cálculos en el procesamiento relacionado con el búfer de profundidad. El valor predeterminado para el parámetro `testCondition` es `GL_LESS`, por lo que un valor de profundidad se procesará si tiene un valor que sea inferior al que esté actualmente almacenado en el búfer de profundidad para dicha posición de píxel.

También podemos establecer el estado del búfer de profundidad, configurándolo como de sólo lectura o de lectura-escritura. Esto se realiza mediante la función:

```
glDepthMask (writeStatus);
```

Cuando `writeStatus = GL_TRUE` (el valor predeterminado), podemos tanto leer como escribir en el búfer de profundidad. Con `writeStatus = GL_FALSE`, el modo de escritura en el búfer de profundidad estará desactivado y sólo podremos extraer valores para compararlos durante los tests de profundidad. Esta característica resulta útil cuando queremos utilizar el mismo fondo complicado para mostrar imágenes de diferentes objetos de primer plano. Después de almacenar el fondo en el búfer de profundidad, desactivamos el modo de escritura y procesamos los objetos de primer plano. Esto nos permite generar una serie de imágenes con diferentes objetos de primer plano o con un objeto en diferentes posiciones para una secuencia de animación. Así, sólo se guardan los valores de profundidad correspondientes al fondo. Otra aplicación de la función `glDepthMask` es para mostrar efectos de transparencia (Sección 10.20). En este caso, sólo queremos guardar las profundidades de los objetos opacos para las pruebas de visibilidad, y no las profundidades de las posiciones correspondientes a superficies transparentes. De modo que se desactivaría la escritura para el búfer de profundidad cada vez que se procesara una superficie transparente. Hay disponibles comandos similares para configurar el estado de escritura para otros búferes (color, índice y patrón de contorno).

## Métodos OpenGL para visibilidad de superficies en representaciones alámbricas

En OpenGL, podemos obtener una visualización alámbrica de un objeto gráfico estándar solicitando que sólo se generen sus aristas. Podemos hacer esto utilizando la función de modo poligonal (Sección 4.14), por ejemplo:

```
glPolygonMode (GL_FRONT_AND_BACK, GL_LINE);
```

pero esto haría que se mostraran tanto las aristas visibles como las ocultas.

Para eliminar las líneas ocultas en una imagen alámbrica, podemos emplear el método de desplazamiento de profundidad descrito en la Sección 4.14. Es decir, primero especificamos la versión alámbrica del objeto utilizando el color de primer plano y luego especificamos una versión de relleno interior utilizando un desplazamiento de profundidad y el color de fondo para el relleno interior. El desplazamiento de profundidad garantiza que el relleno de color de fondo no interfiera con la visualización de las aristas visibles. Como ejemplo, el siguiente segmento de código genera una imagen alámbrica de un objeto utilizando un color blanco en primer plano y un color negro de fondo.

```
 glEnable (GL_DEPTH_TEST);
 glPolygonMode (GL_FRONT_AND_BACK, GL_LINE);
 glColor3f (1.0, 1.0, 1.0);
 /* Invocar la rutina de descripción del objeto. */

 glPolygonMode (GL_FRONT_AND_BACK, GL_FILL);
 glEnable (GL_POLYGON_OFFSET_FILL);
 glPolygonOffset (1.0, 1.0);
 glColor3f (0.0, 0.0, 0.0);
 /* Invocar de nuevo la rutina de descripción del objeto. */

 glDisable (GL_POLYGON_OFFSET_FILL);
```

## Función OpenGL para variación de la intensidad con la profundidad

Podemos variar el brillo de un objeto en función de su distancia a la posición de visualización mediante,

```
 glEnable (GL_FOG);
 glFogi (GL_FOG_MODE, GL_LINEAR);
```

Esto aplica la función de profundidad lineal de la Ecuación 9.9 a los colores de los objetos utilizando  $d_{\min} = 0.0$  y  $d_{\max} = 1.0$ . Pero podemos establecer diferentes valores para  $d_{\min}$  y  $d_{\max}$  mediante las siguientes llamadas a función:

```
 glFogf (GL_FOG_START, minDepth);
 glFogf (GL_FOG_END, maxDepth);
```

En estas dos funciones, a los parámetros `minDepth` y `maxDepth` se les asignan valores en coma flotante, aunque también pueden emplearse valores enteros si cambiamos el sufijo de la función a `i`.

Además, podemos utilizar la función `glFog` para establecer un color atmosférico que se combine con el color de un objeto después de aplicar la función lineal de variación de la intensidad con la profundidad. También se pueden modelar otros efectos atmosféricos, y hablaremos de dichas funciones en la Sección 10.20.

## 9.15 RESUMEN

El test más simple de visibilidad es el algoritmo de detección de caras posteriores, que es rápido y efectivo como mecanismo inicial de filtro para eliminar muchos polígonos de los posteriores tests de visibilidad. Para un único poliedro convexo, la detección de caras posteriores elimina todas las superficies ocultas, pero en general la detección de caras posteriores no puede identificar todas las superficies ocultas completamente.

Un método comúnmente utilizado para identificar todas las superficies visibles de una escena es el algoritmo de búfer de profundidad. Cuando se aplica a objetos gráficos estándar, este procedimiento es altamente eficiente, aunque requiere disponer del suficiente espacio de almacenamiento extra. Hacen falta dos búferes: uno para almacenar los colores de los píxeles y otro para almacenar los valores de profundidad correspondientes a las posiciones de los píxeles. Se utilizan métodos de línea de barrido rápidos y de carácter incremental para procesar cada polígono de una escena con el fin de calcular las profundidades de la superficie. A medida que se procesa cada superficie, se actualizan los dos búferes. Una extensión de la técnica de búfer de profundidad es el método de búfer A, que proporciona información adicional para mostrar superficies transparentes y a las que se les pueden aplicar técnicas de antialiasing.

Se han desarrollado varios otros métodos de detección de visibilidad. El método de líneas de barrido procesa todas las superficies de una vez para cada línea de barrido. Con el método de ordenación de profundidad (el algoritmo del pintor), los objetos se “pintan” en el búfer de refresco de acuerdo con sus distancias con respecto a la posición de visualización. Entre los esquemas de subdivisión para la identificación de las partes visibles de una escena podemos citar el método del árbol BSP, la subdivisión de áreas y las representaciones basadas en árboles octales. Las superficies visibles también pueden detectarse utilizando métodos de proyección de rayos, que producen líneas desde el plano de los píxeles hacia la escena para determinar las posiciones de intersección con los objetos a lo largo de estas líneas proyectadas. Los métodos de proyección de rayos son un caso particular de los algoritmos de trazado de rayos, que permiten mostrar los sistemas con efectos globales de iluminación.

Los métodos de detección de visibilidad también se utilizan para mostrar diagramas lineales tridimensionales. Con las superficies curvas, podemos mostrar gráficas de contorno. Para la visualización alámbrica de poliedros, lo que hacemos es buscar las diversas secciones de aristas de las superficies de una escena que son visibles desde la posición de visualización.

**TABLA 9.1. RESUMEN DE FUNCIONES OpenGL DE DETECCIÓN DE VISIBILIDAD.**

Función	Descripción
<code>glCullFace</code>	Especifica los planos frontal o posterior de los polígonos para las operaciones de eliminación de caras, cuando se activa este mecanismo mediante <code> glEnable (GL_CULL_FACE)</code> .
<code>glutInitDisplayMode</code>	Especifica las operaciones de búfer de profundidad utilizando el argumento <code>GLUT_DEPTH</code> .
<code>glClear (GL_DEPTH_BUFFER_BIT)</code>	Inicializa los valores del búfer de profundidad con el valor predeterminado (1.0) o con un valor especificado por la función <code> glClearDepth</code> .
<code>glClearDepth</code>	Especifica un valor inicial del búfer de profundidad.
<code>glEnable (GL_DEPTH_TEST)</code>	Activa las operaciones de comprobación de profundidad.
<code>glDepthRange</code>	Especifica un rango para la normalización de los valores de profundidad.
<code>glDepthFunc</code>	Especifica una condición de comprobación de profundidad.
<code>glDepthMask</code>	Establece el estado de escritura del búfer de profundidad.
<code>glPolygonOffset</code>	Especifica un desplazamiento para eliminar líneas ocultas en una imagen alámbrica cuando se aplica un color de fondo de relleno.
<code>glFog</code>	Especifica las operaciones lineales de variación de la intensidad con la profundidad y los valores de profundidad mínima y máxima que hay que utilizar en dichos cálculos.

Podemos implementar cualquier esquema deseado de detección de visibilidad en un programa de aplicación creando nuestras propias rutinas, pero las bibliotecas gráficas suelen proporcionar funciones únicamente para la eliminación de caras posteriores y para la incrementación del método del búfer de profundidad. En los sistemas infográficos de alta gama, las rutinas de búfer de profundidad están implementadas en hardware.

En la biblioteca básica de OpenGL hay disponibles funciones para eliminación de polígonos posteriores y para determinación de visibilidad basada en búfer de profundidad. Con las rutinas de eliminación de polígonos posteriores, podemos eliminar las caras posteriores de objetos gráficos estándar, sus caras frontales o ambas. Con las rutinas de búfer de profundidad, podemos establecer el rango para las comprobaciones de profundidad y el tipo de comprobación de profundidad que haya que realizar. Las imágenes alámbricas se obtienen utilizando las operaciones OpenGL de modo poligonal de desplazamiento de polígonos. Y también pueden generarse escenas OpenGL utilizando efectos de variación de la intensidad de acuerdo con la profundidad. En la Tabla 9.1 se resumen las funciones OpenGL de comprobación de visibilidad. La función de modo poligonal y otras operaciones relacionadas se resumen al final del Capítulo 4.

## REFERENCIAS

---

Entre las fuentes adicionales de información sobre algoritmos de visibilidad podemos citar Elber y Cohen (1990), Franklin y Kankanhalli (1990), Segal (1990) y Naylor, Amanatides y Thibault (1990). Los métodos de búfer A se presentan en Cook, Carpenter y Catmull (1987), Haeberli y Akeley (1990) y Shilling y Strasser (1993). Puede encontrar un resumen de los métodos de dibujo de contornos en Earnshaw (1985).

Si quiere aprender más sobre técnicas de programación para pruebas de visibilidad puede consultar Glassner (1990), Arvo (1991), Kirk (1992), Heckbert (1994) y Paeth (1995). Woo, Neider, Davis y Shreiner (1999) proporcionan explicaciones adicionales sobre las funciones OpenGL de detección de visibilidad y puede encontrar un listado completo de las funciones OpenGL disponibles en la biblioteca básica y en GLU en Shreiner (2000).

## EJERCICIOS

---

- 9.1 Defina un procedimiento de detección de caras posteriores que permita identificar todas las caras visibles de cualquier poliedro convexo de entrada que tenga superficies con diferentes colores. El poliedro debe definirse en un sistema de visualización que cumpla la regla de la mano derecha y la dirección de visualización es uno de los parámetros de entrada que debe especificar el usuario.
- 9.2 Implemente el procedimiento del ejercicio anterior utilizando una proyección paralela ortográfica para ver las superficies visibles del poliedro convexo proporcionado como entrada. Suponga que todas las partes del objeto se encuentran delante del plano de visualización.
- 9.3 Implemente el procedimiento del Ejercicio 9.1 utilizando una proyección de perspectiva para ver las caras visibles del poliedro convexo proporcionado como entrada. Suponga que todas las partes del objeto se encuentran delante del plano de visualización.
- 9.4 Escriba un programa para generar una animación de un poliedro convexo. Hay que rotar incrementalmente el objeto alrededor de un eje que pase por el objeto y sea paralelo al plano de visualización. Suponga que el objeto cae completamente delante del plano de visualización. Utilice una proyección ortográfica paralela para mapear las vistas sucesivamente sobre el plano de visualización.
- 9.5 Escriba una rutina para implementar el método del búfer de profundidad para la visualización de las caras visibles de cualquier poliedro que suministre como entrada. La matriz para el búfer de profundidad puede tener cualquier tamaño que sea conveniente para su sistema, como por ejemplo 500 por 500. ¿Cómo pueden determinarse los requisitos de almacenamiento para el búfer de profundidad a partir de las definiciones de los objetos que hay que mostrar?

- 9.6 Modifique el procedimiento del ejercicio anterior para mostrar las caras visibles de una escena que contenga cualquier número de poliedros. Defina métodos eficientes para almacenar y procesar los diversos objetos de la escena.
- 9.7 Modifique el procedimiento del ejercicio anterior para implementar el algoritmo del búfer A para la visualización de una escena que contenga tanto superficies opacas como transparentes.
- 9.8 Amplíe el procedimiento desarrollado en el ejercicio anterior para incluir técnicas de antialiasing.
- 9.9 Desarrolle un programa para implementar el algoritmo de línea de barrido para la visualización de las superficies visibles de un poliedro dado. Utilice las tablas de polígonos para almacenar la definición del objeto y emplee técnicas de coherencia para evaluar los puntos a lo largo de una línea de barrido y entre unas líneas de barrido y otras.
- 9.10 Escriba un programa para implementar el algoritmo de línea de barrido para una escena que contenga diversos poliedros. Utilice tablas de polígonos para almacenar la definición de los objetos y emplee técnicas de coherencia para evaluar los puntos a lo largo de una línea de barrido y entre una línea de barrido y la siguiente.
- 9.11 Diseñe un programa para mostrar las superficies visibles de un poliedro convexo utilizando el algoritmo del pintor, es decir, ordene las superficies según su profundidad y píntelas comenzando desde atrás.
- 9.12 Escriba un programa que utilice el método de ordenación de profundidad para mostrar las superficies visibles de cualquier objeto dado definido mediante una serie de caras planas.
- 9.13 Diseñe un programa de ordenación de profundidad para mostrar las superficies visibles en una escena que contenga varios poliedros.
- 9.14 Escriba un programa para mostrar las superficies visibles de un poliedro convexo utilizando el método del árbol BSP.
- 9.15 Proporcione ejemplos de situaciones en las que los dos métodos expuestos para la condición 3 del algoritmo de subdivisión de áreas no permitan identificar correctamente una superficie circundante que oculte todas las demás superficies.
- 9.16 Desarrolle un algoritmo que compruebe si una superficie plana dada es circundante, solapada, interior o exterior con respecto a una determinada área rectangular.
- 9.17 Diseñe un algoritmo para generar una representación en árbol cuádrico para las superficies visibles de un objeto, aplicando los tests de subdivisión de área para determinar los valores de los elementos del árbol cuádrico.
- 9.18 Diseñe un algoritmo para almacenar una representación en árbol cuádrico de un objeto dentro de un búfer de imagen.
- 9.19 Defina un procedimiento para mostrar las superficies visibles de un objeto descrito mediante una representación en árbol octal.
- 9.20 Diseñe un algoritmo para visualizar una única esfera utilizando el método de proyección de rayos.
- 9.21 Explique cómo pueden incorporarse los métodos de antialiasing en los diversos algoritmos de eliminación de superficies ocultas.
- 9.22 Escriba una rutina para generar un diagrama de contorno de una superficie, dada la función de superficie  $f(x, y)$ .
- 9.23 Desarrolle un algoritmo para detectar las secciones de linea visible en una escena comparando cada línea de la escena con cada faceta poligonal de una superficie.
- 9.24 Explique cómo pueden generarse imágenes alámbricas con los diversos métodos de detección de superficies visibles expuestos en este capítulo.
- 9.25 Diseñe un procedimiento para generar una imagen alámbrica de un poliedro en la que se muestren las aristas ocultas del objeto como líneas punteadas.
- 9.26 Escriba un programa para mostrar un poliedro en el que se eliminen determinadas caras seleccionadas, utilizando las funciones de eliminación de polígonos de OpenGL. A cada cara del polígono hay que asignarle un color diferente y es el usuario quien puede seleccionar una cara para su eliminación. Asimismo, el usuario debe proporcionar también como valores de entrada la posición de visualización y los demás parámetros de visualización.

- 9.27 Modifique el programa del ejercicio anterior para poder ver el poliedro desde cualquier posición, utilizando rutinas de búfer de profundidad en lugar de rutinas de eliminación de polígonos.
- 9.28 Modifique el programa del ejercicio anterior para poder especificar también como entrada el rango de profundidades y la condición de test de profundidad.
- 9.29 Genere una imagen alámbrica de un poliedro utilizando las funciones `glPolygonMode` y `glPolygonOffset` expuestas en la Sección 9.14.
- 9.30 Modifique el programa del ejercicio anterior para mostrar el poliedro utilizando la función `glFogi` de variación de la intensidad con la profundidad.
- 9.31 Modifique el programa del ejercicio anterior para mostrar varios poliedros que estén distribuidos a diversas profundidades. El rango de variación de la intensidad con la profundidad debe configurarse según los datos de entrada proporcionados por el usuario.

# Modelos de iluminación y métodos de representación superficial



Una escena de la película de animación por computadora *Final Fantasy: The Spirits Within*, donde se muestran los efectos de iluminación utilizados para simular la explosión de un espíritu.  
(Cortesía de Square Pictures, Inc. © 2001 FFFP. Todos los derechos reservados.)

- |   |   |
|---|---|
| <ul style="list-style-type: none"><li>10.1 Fuentes luminosas</li><li>10.2 Efectos de iluminación superficial</li><li>10.3 Modelos básicos de iluminación</li><li>10.4 Superficies transparentes</li><li>10.5 Efectos atmosféricos</li><li>10.6 Sombras</li><li>10.7 Parámetros de la cámara</li><li>10.8 Visualización de la intensidad de la luz</li><li>10.9 Patrones de semitono y técnicas de aleatorización</li><li>10.10 Métodos de representación de polígonos</li><li>10.11 Métodos de trazado de rayos</li></ul> | <ul style="list-style-type: none"><li>10.12 Modelo de iluminación de radiosidad</li><li>10.13 Mapeado de entorno</li><li>10.14 Mapeado de fotones</li><li>10.15 Adición de detalles a las superficies</li><li>10.16 Modelado de los detalles superficiales mediante polígonos</li><li>10.17 Mapeado de texturas</li><li>10.18 Mapeado de relieve</li><li>10.19 Mapeado del sistema de referencia</li><li>10.20 Funciones OpenGL de iluminación y representación de superficies</li><li>10.21 Funciones de texturas OpenGL</li><li>10.22 Resumen</li></ul> |
|---|---|

Pueden obtenerse imágenes realistas de una escena generando proyecciones en perspectiva de los objetos y aplicando efectos de iluminación naturales a las superficies visibles. Se utiliza un **modelo de iluminación**, también denominado **modelo de sombreado**, para calcular el color de cada posición iluminada en la superficie de un objeto. Un **método de representación superficial** utiliza los cálculos de color del modelo de iluminación para determinar los colores de los píxeles para todas las posiciones proyectadas de una escena. El modelo de iluminación puede aplicarse a cada posición de proyección, o bien puede llevarse a cabo la representación de la superficie interpolando los colores de las superficies a partir de un pequeño conjunto de cálculos relativos al modelo de iluminación. Los algoritmos del espacio de imagen basados en las líneas de barrido utilizan normalmente esquemas de interpolación, mientras que los algoritmos de trazado de rayos pueden invocar el modelo de iluminación para cada posición de píxel. Algunas veces, se denomina procedimientos de representación de superficie a los métodos de sombreado que calculan los colores de las superficies utilizando el modelo de sombreado, pero esto puede llevar a cierta confusión entre ambos términos. Para evitar posibles malas interpretaciones debido al uso de terminología similar, denominaremos *modelo de iluminación* al modelo utilizado para calcular la intensidad lumínosa en cada punto de una superficie, y emplearemos el término *representación superficial* para referirnos a un procedimiento mediante el cual se aplica un modelo de iluminación con el fin de obtener los colores de píxel para todas las posiciones proyectadas de la superficie.

Entre otras cosas, el fotorrealismo en los gráficos por computadora requiere dos elementos: representaciones precisas de las propiedades de las superficies y una buena descripción física de los efectos de iluminación en la escena. Estos efectos de iluminación de las superficies incluyen la reflexión de la luz, la transparencia, las texturas de las superficies y las sombras.

En general, el modelado de los efectos de iluminación que podemos observar sobre un objeto es un proceso muy complejo, en el que intervienen principios tanto de la física como de la psicología. Fundamentalmente, los efectos de iluminación se describen mediante modelos que tienen en cuenta la interacción de la energía electromagnética con las superficies de los objetos de la escena. Una vez que los rayos luminosos alcanzan nuestros ojos, se ponen en marcha determinados procesos de percepción que son los que dictan lo que realmente «vemos». Los modelos físicos de iluminación tienen en cuenta una serie de factores, como las propiedades de los materiales, las posiciones de los objetos en relación con las fuentes de ilumina-

ción y con otros objetos y las características de las fuentes luminosas. Los objetos pueden estar compuestos de materiales opacos, o bien pueden ser más o menos transparentes. Además, pueden tener superficies brillantes o mates, y exhibir diversos patrones de textura superficial. Pueden utilizarse fuentes luminosas, de formas, colores y posiciones variables para iluminar una escena. Dados los parámetros de las propiedades ópticas de las superficies, dadas las posiciones relativas de las superficies dentro de una escena, dados el color y las posiciones de las fuentes luminosas, dadas las características de dichas fuentes y dadas la posición y la orientación del plano de visualización, se utilizan los modelos de iluminación para calcular la intensidad de la luz proyectada desde una posición concreta de la superficie en una dirección de visualización especificada.

Los modelos de iluminación en infografía son a menudo aproximaciones de la leyes físicas que describen los efectos de iluminación de una superficie. Para reducir los cálculos, la mayoría de los paquetes utilizan modelos empíricos basados en cálculos de fotometría simplificados. Otros modelos más precisos, como el algoritmo de radiosidad, calculan las intensidades luminosas considerando la propagación de la energía radiante entre las fuentes luminosas y las diversas superficies de una escena. En las siguientes secciones, vamos primero a echar un vistazo a los modelos de iluminación básicos que más a menudo se utilizan en los sistemas infográficos, para pasar después a analizar otros métodos más precisos, pero más complejos, de determinación de la apariencia de las superficies iluminadas. Exploraremos también los diversos algoritmos de representación superficial que pueden utilizarse para aplicar los modelos de iluminación, con el fin de obtener imágenes de calidad de escenas naturales.

## 10.1 FUENTES LUMINOSAS

---

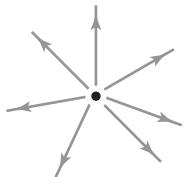
Cualquier objeto que emita energía radiante es una **fuente luminosa** que contribuye a los efectos de iluminación que afectan a otros objetos de la escena. Podemos modelar fuentes luminosas con diversas formas y características, y la mayoría de los emisores sirven únicamente como fuente de iluminación de una escena. Sin embargo, en algunas aplicaciones, puede que nos interese crear un objeto que sea a la vez una fuente luminosa y un reflector de luz. Por ejemplo, un globo de plástico que rodee a una bombilla emite luz, pero también los rayos luminosos procedentes de otras fuentes se reflejan en la superficie del globo. También podríamos modelar el globo como una superficie semitransparente dispuesta en torno a una fuente luminosa, pero para algunos objetos, como por ejemplo un panel fluorescente de gran tamaño, puede que sea más conveniente describir la superficie simplemente como una combinación de un emisor y un reflector.

Las fuentes luminosas pueden definirse con diversas propiedades. Podemos definir su posición, el color de la luz emitida, la dirección de emisión y la forma de la fuente. Si la fuente es también una superficie reflectora de la luz, necesitaremos indicar sus propiedades de reflectividad. Además, podemos definir una fuente luminosa que emita diferentes colores en diferentes direcciones. Por ejemplo, se podría especificar una fuente que emitiera luz roja por uno de sus lados y luz verde por el otro.

En la mayoría de las aplicaciones, y particularmente en los gráficos en tiempo real, se utiliza un modelo simple de fuentes luminosas para evitar complicar demasiado los cálculos. Las propiedades de emisión de luz se definen utilizando un único valor para cada uno de los componentes de color RGB, que se corresponde con la «intensidad» de dicha componente de color. Los parámetros de color y los modelos de fuentes de iluminación se analizan con más detalle en el Capítulo 12.

### Fuentes luminosas puntuales

El modelo más simple para un objeto que emite energía radiante es la **fuente luminosa puntual** de un único color, el cual se especifica mediante las tres componentes RGB. Podemos definir una fuente puntual para una escena indicando su posición y el color de la luz emitida. Como se muestra en la Figura 10.1, los rayos luminosos se generan según una serie de trayectorias radialmente divergentes a partir de esa única fuente puntual monocromática. Este modelo de fuente luminosa constituye una aproximación razonable para aquellas fuentes cuyas dimensiones sean pequeñas comparadas con el tamaño de los objetos de la escena. También pode-



**FIGURA 10.1.** Trayectorias divergentes de los rayos a partir de una fuente luminosa puntual.

mos simular fuentes de mayor tamaño mediante emisores puntuales si dichas fuentes no están demasiado próximas a la escena. Utilizamos la posición de una fuente puntual dentro de un modelo de imagen para determinar qué objetos de la escena se ven iluminados por dicha fuente y para calcular la dirección de los rayos luminosos cuando éstos inciden sobre una posición seleccionada de la superficie del objeto.

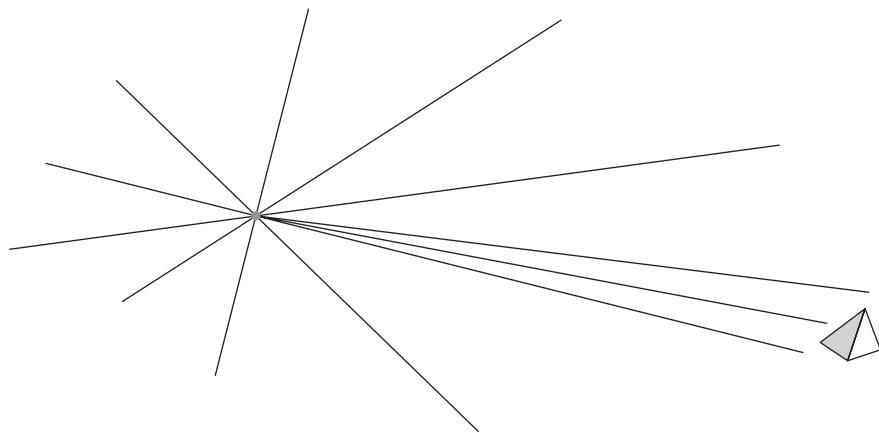
### Fuentes luminosas infinitamente distantes

Una fuente luminosa de gran tamaño, como por ejemplo el Sol, pero que esté muy lejos de una escena puede también aproximarse como un emisor puntual, aunque en este caso la variación que existe en sus efectos direccionales es muy pequeña. Por contraste con una fuente luminosa situada en mitad de una escena, que ilumina los objetos situados en todas las direcciones con respecto a la fuente, las cuentas remotas iluminan la escena desde una única dirección. El trayecto del rayo luminoso que va desde una fuente distante hasta cualquier posición de la escena es prácticamente constante, como se ilustra en la Figura 10.2.

Podemos simular una fuente luminosa infinitamente distante asignándola un valor de color y una dirección fija para los rayos luminosos que emanen de la fuente. En los cálculos de iluminación sólo hace falta conocer el color de la fuente luminosa y el vector correspondiente a la dirección de emisión, siendo completamente irrelevante cuál sea la posición de la fuente.

### Atenuación radial de la intensidad

A medida que la energía radiante de una fuente luminosa viaja a través del espacio, su amplitud a cualquier distancia  $d_i$  de la fuente se atenúa según el factor  $1/d_i^2$ . Esto significa que una superficie próxima a la fuente luminosa recibe una intensidad de luz incidente mayor que otra superficie más distante. Por tanto, para producir efectos de iluminación realistas, tenemos que tener en cuenta esta atenuación de la intensidad. En caso contrario, todas las superficies serían iluminadas con la misma intensidad por las fuentes luminosas y podrían obtenerse, como resultado efectos deseables en las imágenes. Por ejemplo, si dos superficies con los mismos



**FIGURA 10.2.** Los rayos luminosos procedentes de una fuente infinitamente distante iluminan a los objetos según una serie de trayectos prácticamente paralelos.

parámetros ópticos se proyectan sobre posiciones que se solapan, serían indistinguibles la una de la otra. Como consecuencia, independientemente de sus distancias relativas con respecto a la fuente luminosa, las dos superficies parecerían ser una sola.

En la práctica, sin embargo, utilizar un factor de atenuación de  $1/d_l^2$  con una fuente puntual no siempre produce imágenes realistas. El factor  $1/d_l^2$  tiende a producir una variación excesiva de la intensidad para objetos que se encuentren próximos a la fuente luminosa, y muy poca variación cuando  $d_l$  es grande. Esto se debe a que las fuentes luminosas reales no son puntos infinitesimales, e iluminar una escena con emisores puntuales es sólo una aproximación simple de los verdaderos efectos de iluminación. Para generar imágenes más realistas utilizando fuentes puntuales, podemos atenuar las intensidades luminosas con una función cuadrática inversa de  $d_l$  que incluya un término lineal:

$$f_{\text{radatten}}(d_l) = \frac{1}{a_0 + a_1 d_l + a_2 d_l^2} \quad (10.1)$$

Los valores numéricos de los coeficientes  $a_0$ ,  $a_1$  y  $a_2$  pueden entonces ajustarse para producir unos efectos de atenuación óptimos. Por ejemplo, podemos asignar un gran valor a  $a_0$  cuando  $d_l$  es muy pequeña con el fin de prevenir que  $f_{\text{radatten}}(d_l)$  se haga demasiado grande. Como opción adicional, a menudo disponible en muchos paquetes de gráficos, puede asignarse un conjunto diferente de valores a los coeficientes de atenuación de cada fuente luminosa puntual de la escena.

No podemos aplicar la Ecuación 10.1 de cálculo de la atenuación de la intensidad a una fuente puntual que esté situada en el «infinito», porque la distancia a la fuente luminosa es indeterminada. Asimismo, todos los puntos de la escena están a una distancia prácticamente igual de las fuentes muy lejanas. Con el fin de tener en cuenta tanto las fuentes luminosas remotas como las locales, podemos expresar la función de atenuación de la intensidad como:

$$f_{l,\text{radatten}} = \begin{cases} 1.0, & \text{si la fuente está en el infinito} \\ \frac{1}{a_0 + a_1 d_l + a_2 d_l^2}, & \text{si la fuente es local} \end{cases} \quad (10.2)$$

## Fuentes de luz direccionales y efectos de foco

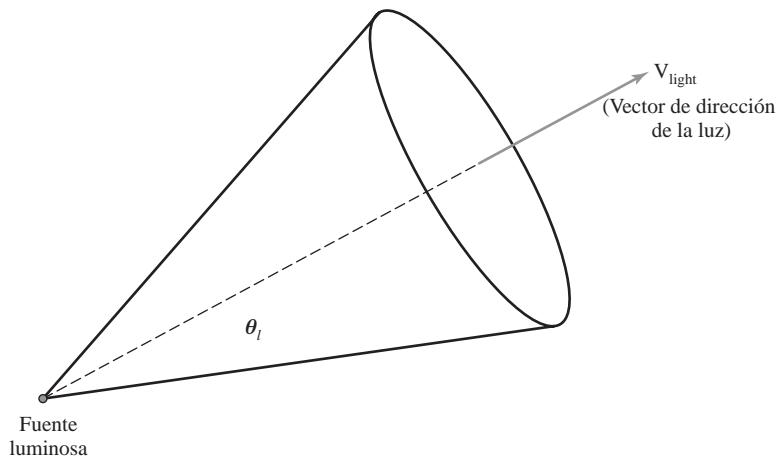
Una fuente luminosa local puede modificarse fácilmente para que actúe como un foco, generando un haz luminoso direccional. Si un objeto está fuera de los límites direccionales de la fuente luminosa, lo excluiremos de los cálculos de iluminación correspondientes a dicha fuente. Una forma de definir una fuente direccional luminosa consiste en asignarla un vector de dirección y un límite angular  $\theta_l$  medido con respecto a dicho vector de dirección, además de definir la posición en color de la fuente. Esto especifica una región cónica del espacio en la que el vector de la fuente luminosa está dirigido según el eje del cono (Figura 10.3). De esta forma, podríamos modelar una fuente luminosa puntual multicolor utilizando múltiples vectores de dirección y un color de emisión diferente para cada una de esas direcciones.

Si denominamos  $\mathbf{V}_{\text{light}}$  al vector unitario que define la dirección de la fuente luminosa y  $\mathbf{V}_{\text{obj}}$  al vector unitario que apunta desde la posición de la fuente hasta la posición de un objeto, tendremos que:

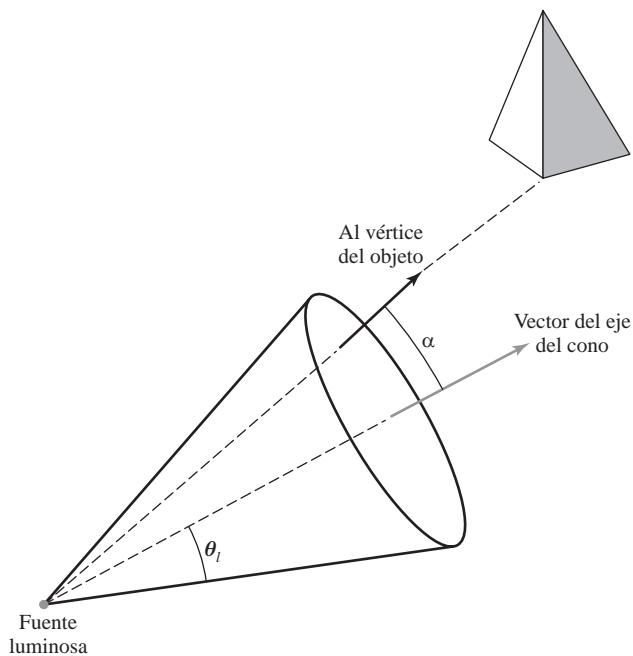
$$\mathbf{V}_{\text{obj}} \cdot \mathbf{V}_{\text{light}} = \cos \alpha \quad (10.3)$$

donde el ángulo  $\alpha$  es la distancia angular del objeto con respecto al vector que indica la dirección de la fuente.

Si restringimos la extensión angular de cualquier cono luminoso de modo que  $0^\circ < \theta_l \leq 90^\circ$ , entonces el objeto estará dentro del rango de iluminación del foco si  $\cos \alpha \geq \cos \theta_l$ , como se muestra en la Figura 10.4. Por el contrario, si  $\mathbf{V}_{\text{obj}} \cdot \mathbf{V}_{\text{light}} < \cos \theta_l$ , el objeto estará fuera del cono de luz.



**FIGURA 10.3.** Una fuente luminosa puntual direccional. El vector unitario de dirección de la luz define el eje de un cono luminoso y el ángulo  $\theta_l$  define la extensión angular del cono circular.

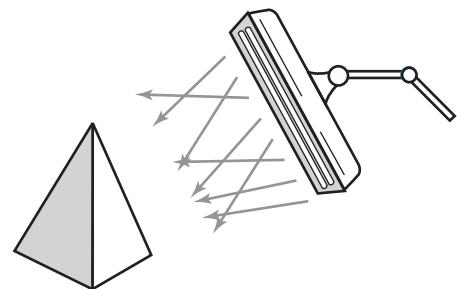


**FIGURA 10.4.** Un objeto iluminado por una fuente luminosa puntual direccional.

### Atenuación angular de la intensidad

Para una fuente luminosa direccional, podemos atenuar angularmente la intensidad de la luz de la fuente, además de atenuarla radialmente con respecto a la posición de la fuente puntual. Esto nos permite simular un cono de luz que sea más intenso a lo largo del eje del cono, decreciendo la intensidad a medida que nos alejamos de dicho eje. Una función comúnmente utilizada para la atenuación angular de la intensidad de una fuente luminosa direccional es:

$$f_{\text{angatten}}(\phi) = \cos^{a_l} \phi, \quad 0^\circ \leq \phi \leq \theta \quad (10.4)$$



**FIGURA 10.5.** Un objeto iluminado por una fuente luminosa de gran tamaño situada muy cerca.



**FIGURA 10.6.** Efectos de iluminación de estudio producidos con el modelo de Warn, utilizando cinco fuentes luminosas complejas con el fin de iluminar un Chevrolet Camaro. (Cortesía de David R. Warn, General Motors Research Laboratories.)

donde al exponente de atenuación  $a_l$  se le asigna algún valor positivo y el ángulo  $\phi$  se mide con respecto al eje del cono. A lo largo del eje del cono,  $\phi = 0$  y  $f_{\text{angatten}}(\phi) = 1.0$ . Cuanto mayor sea el valor del exponente de atenuación  $a_l$ , más pequeño será el valor de la función de atenuación angular de la intensidad para un cierto valor del ángulo  $\phi > 0$ .

Hay varios casos especiales que es necesario considerar en la implementación de la función angular de atenuación. No existirá atenuación angular si la fuente luminosa no es direccional (es decir, si no es un foco). Asimismo, ningún objeto será iluminado por la fuente luminosa si está situado fuera del cono del foco. Para determinar el factor de atenuación angular a lo largo de una línea que vaya desde la posición de la fuente hasta una posición de una superficie en una escena, podemos calcular el coseno del ángulo de dirección con respecto al eje del cono utilizando el producto escalar de la Ecuación 10.3. Si llamamos  $\mathbf{V}_{\text{light}}$  al vector unitario que indica la dirección de la fuente luminosa (a lo largo del eje del cono) y  $\mathbf{V}_{\text{obj}}$  al vector unitario correspondiente a la línea que une la fuente luminosa con la posición de un objeto y si suponemos que  $0^\circ < \theta_l \leq 90^\circ$ , podemos expresar la ecuación general de la atenuación angular como:

$$f_{l,\text{angatten}} = \begin{cases} 1.0, & \text{si la fuente no es un foco} \\ 0.0, & \text{si } \mathbf{V}_{\text{obj}} \cdot \mathbf{V}_{\text{light}} = \cos \alpha < \cos \theta_l \\ & (\text{el objeto está fuera del cono del foco}) \\ (\mathbf{V}_{\text{obj}} \cdot \mathbf{V}_{\text{light}})^{a_l}, & \text{en caso contrario} \end{cases} \quad (10.5)$$

### Fuentes luminosas complejas y el modelo de Warn

Cuando necesitemos incluir una fuente luminosa de gran tamaño en una posición próxima a los objetos de una escena, como por ejemplo la gran lámpara de neón de la Figura 10.5. Podemos aproximarla mediante una superficie emisora de luz. Una forma de hacer esto consiste en modelar la superficie luminosa como una cu-

drícula de emisores puntuales direccionales. Podemos definir la dirección de las fuentes puntuales de modo que los objetos situados detrás de la superficie emisora de luz no sean iluminados. Y también podemos incluir otros controles para restringir la dirección de la luz emitida en las proximidades de los bordes de la fuente.

El **modelo de Warn** proporciona un método para producir efectos de iluminación de estudio utilizando conjuntos de emisores puntuales con diversos parámetros que pretenden simular las pantallas, controles de foco y demás aparatos utilizados por los fotógrafos profesionales. Los efectos de foco se consiguen mediante los conos de iluminación que antes hemos presentado, mientras que las pantallas proporcionan controles direccionales adicionales. Por ejemplo, pueden especificarse dos pantallas para cada una de las direcciones  $x$ ,  $y$  y  $z$  con el fin de restringir aún más el trayecto de los rayos luminosos emitidos. Esta simulación de fuentes luminosas está implementada en algunos paquetes gráficos, y la Figura 10.6 ilustra los efectos de iluminación que pueden conseguirse mediante el modelo de Warn.

## 10.2 EFECTOS DE ILUMINACIÓN SUPERFICIAL

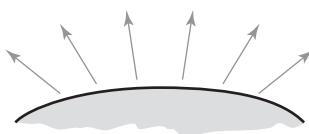
---

Un modelo de iluminación calcula los efectos luminosos sobre una superficie utilizando las distintas propiedades ópticas que se hayan asignado a dicha superficie. Esas propiedades incluyen el grado de transparencia, los coeficientes de reflexión del color y diversos parámetros relativos a la textura de la superficie.

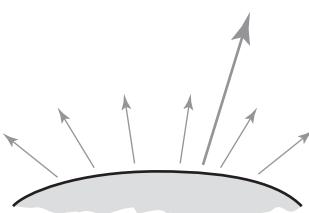
Cuando la luz incide sobre una superficie opaca, parte de la misma se refleja y parte se absorbe. La cantidad de luz incidente reflejada por la superficie dependerá del tipo de material. Los materiales brillantes reflejan un mayor porcentaje de la luz incidente, mientras que las superficies mates absorben más la luz. Para una superficie transparente, parte de la luz incidente también se transmite a través del material.

Las superficies rugosas o granulosas tienden a dispersar la luz reflejada en todas las direcciones. Esta luz dispersada se denomina **reflexión difusa**. Una superficie mate muy rugosa produce principalmente reflexiones difusas, de modo que la superficie parece igualmente brillante desde cualquier ángulo. La Figura 10.7 ilustra la dispersión difusa de la luz en una superficie. Lo que denominamos el color de un objeto es el color de la reflexión difusa cuando el objeto se ilumina con luz blanca, que está compuesta de una combinación de todos los colores. Un objeto azul, refleja la componente azul de la luz blanca y absorbe todas las demás componentes de color. Si el objeto azul se contempla bajo una luz roja, parecerá negro, ya que toda la luz incidente será absorbida.

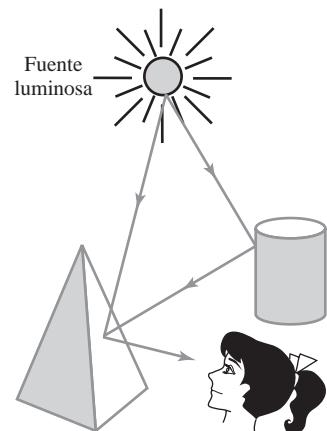
Además de la dispersión difusa de la luz, parte de la luz reflejada se concentra en lo que se denomina un resalte, llamándose a este fenómeno **reflexión especular**. Este efecto de resalte es más pronunciado en las superficies brillantes que en las mates. Podemos ver la reflexión especular cuando observamos una superficie brillante iluminada, como por ejemplo un trozo de metal pulido, una manzana o la frente de una persona, pero sólo podemos percibir esa reflexión especular cuando contemplamos la superficie desde una dirección concreta. En la Figura 10.8 se muestra una representación del fenómeno de la reflexión especular.



**FIGURA 10.7.** Reflexiones difusas en una superficie.



**FIGURA 10.8.** Reflexión especular superpuesta sobre los vectores de reflexión difusa.



**FIGURA 10.9.** Los efectos de iluminación superficial se producen mediante una combinación de la iluminación procedente de fuentes luminosas y de las reflexiones producidas en otras superficies.

Otro factor que hay que considerar en los modelos de iluminación es la **luz de fondo** o **luz ambiental** de la escena. Una superficie que no esté directamente expuesta a una fuente luminosa puede seguir siendo visible debido a la luz reflejada en los objetos cercanos que sí están iluminados. Así, la luz ambiente de una escena es el efecto de iluminación producido por la luz reflejada en las diversas superficies de la escena. La Figura 10.9 ilustra este efecto de iluminación de fondo. La luz total reflejada por una superficie es la suma de las contribuciones de las fuentes luminosas y de la luz reflejada por otros objetos iluminados.

## 10.3 MODELOS BÁSICOS DE ILUMINACIÓN

Los modelos más precisos de iluminación superficial calculan los resultados de las interacciones entre la energía radiante incidente y la composición material de un objeto. Para simplificar los cálculos de iluminación superficial, podemos utilizar representaciones aproximadas de los procesos físicos que producen los efectos de iluminación expuestos en la sección anterior. El modelo empírico descrito en esta sección produce unos resultados razonablemente buenos y es el que se implementa en la mayoría de los sistemas gráficos.

Los objetos emisores de luz en un modelo básico de iluminación suelen estar limitados, generalmente, a fuentes puntuales. Sin embargo, muchos paquetes gráficos proporcionan funciones adicionales para incluir fuentes direccionales (focos) y fuentes luminosas complejas.

### Luz ambiente

En nuestro modelo básico de iluminación, podemos incorporar la luz de fondo definiendo un nivel de brillo general para la escena. Esto produce una iluminación ambiente uniforme que es igual para todos los objetos y que aproxima las reflexiones difusas globales correspondientes a las diversas superficies iluminadas.

Suponiendo que estemos describiendo únicamente efectos de iluminación monocromáticos, como por ejemplo escalas de grises, designaríamos el nivel de la luz ambiente en una escena mediante un parámetro de intensidad  $I_a$ . Cada superficie de la escena se verá entonces iluminada por esta luz de fondo. Las reflexiones producidas por la iluminación mediante la luz ambiente son sólo una forma de reflexión difusa, y son independientes de la dirección de visualización y de la orientación espacial de las superficies. Sin embargo, la cantidad de luz ambiente incidente que se refleje dependerá de las propiedades ópticas de las superficies, que determinan qué parte de la energía incidente se refleja y qué parte se absorbe.

### Reflexión difusa

Podemos modelar las reflexiones difusas de una superficie asumiendo que la luz incidente se dispersa con igual intensidad en todas las direcciones, independientemente de la posición de visualización. Tales superfi-

cies se denominan **reflectores difusos ideales**. También se les denomina **reflectores lambertianos**, porque la energía luminosa radiante reflejada por cualquier punto de la superficie se calcula mediante la **ley del coseno de Lambert**. Esta ley establece que la cantidad de energía radiante procedente de cualquier pequeña área de superficie  $dA$  en una dirección  $\phi_N$  relativa a la normal a la superficie es proporcional a  $\cos \phi_N$  (Figura 10.10). La intensidad de la luz en esta dirección puede calcularse dividiendo la magnitud de la energía radiante por unidad de tiempo entre la proyección de ese área superficial en la dirección de radiación:

$$\begin{aligned} \text{Intensidad} &= \frac{\text{energía radiante por unidad de tiempo}}{\text{área proyectada}} \\ &\propto \frac{\cos \phi_N}{dA \cos \phi_N} \\ &= \text{constante} \end{aligned} \quad (10.6)$$

Así, para la reflexión lambertiana, la intensidad de la luz es la misma en todas las direcciones de visualización.

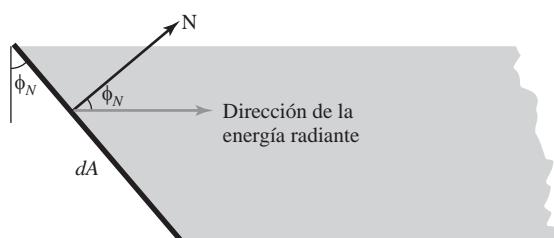
Suponiendo que haya que tratar a todas las superficies como un reflector difuso ideal (lambertiano), podemos definir un parámetro  $k_d$  para cada superficie que determine la fracción de la luz incidente que hay que dispersar en forma de reflexiones difusas. Este parámetro se denomina **coeficiente de reflexión difusa o reflectividad difusa**. La reflexión difusa en todas las direcciones se da entonces una constante cuyo valor es igual a la intensidad de la luz incidente multiplicada por el coeficiente de reflexión difusa. Para una fuente luminosa monocromática, al parámetro  $k_d$  se le asigna un valor constante en el intervalo 0.0 a 1.0, de acuerdo con las propiedades de reflexión que queramos que la superficie tenga. Para una superficie altamente reflectante, asignaremos a  $k_d$  un valor próximo a 1.0. Esto produce una superficie más brillante, en la que la intensidad de la luz reflejada estará más próxima a la de la luz incidente. Si queremos simular una superficie que absorba la mayor parte de la luz incidente, asignaremos a la reflectividad un valor próximo a 0.0.

Para los efectos de iluminación de fondo, podemos asumir que todas las superficies están completamente iluminadas por la luz ambiente  $I_a$  que hayamos asignado a la escena. Por tanto, la contribución de la luz ambiente a la reflexión difusa en cualquier punto de una superficie es simplemente:

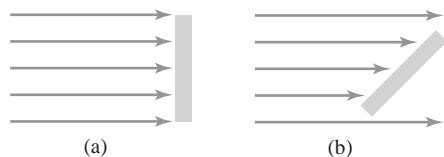
$$I_{\text{ambdiff}} = k_d I_a \quad (10.7)$$

La luz ambiente sola, sin embargo, produce un sombreado plano y poco interesante de las superficies (Figura 10.23(b)), por lo que raramente se representan las escenas utilizando únicamente luz ambiente. Al menos se suele incluir una fuente luminosa en la escena, a menudo definida como fuente puntual situada en la posición de visualización.

Cuando se ilumina una superficie mediante una fuente luminosa de intensidad  $I_p$ , la cantidad de luz incidente dependerá de la orientación de la superficie en relación con la dirección de la fuente luminosa. Una superficie que esté orientada en perpendicular a la dirección de iluminación recibirá más luz de la fuente que otra superficie que forme un ángulo oblicuo con la dirección de la luz incidente. Este efecto de iluminación puede observarse sobre una hoja de papel blanco que se coloque en paralelo a una ventana iluminada por el



**FIGURA 10.10.** La energía radiante de un elemento de área superficial  $dA$  en la dirección  $\phi_N$  relativa a la normal a la superficie es proporcional a  $\cos \phi_N$ .



**FIGURA 10.11.** Una superficie perpendicular a la dirección de la luz incidente (a) estará más iluminada que una superficie de igual tamaño que forme un ángulo oblicuo (b) con la dirección de la luz.

Sol. Al girar lentamente la hoja de papel con respecto a la dirección de la ventana, el brillo de la superficie disminuye. La Figura 10.11 ilustra este efecto, mostrando un haz de rayos luminosos que inciden sobre dos superficies planas de igual área que tienen diferente orientación espacial en relación con la dirección de iluminación de una fuente distante (rayos entrantes paralelos).

En la Figura 10.11 podemos ver que el número de rayos luminosos que intersectan un elemento de superficie es proporcional al área de la proyección de la superficie en perpendicular a la dirección de la luz incidente. Si denominamos  $\theta$  al **ángulo de incidencia** entre los rayos luminosos y la normal de la superficie (Figura 10.12), entonces el área proyectada de un elemento de superficie en perpendicular a la dirección de la luz será proporcional a  $\cos \theta$ . Por tanto, podemos modelar la cantidad de luz incidente sobre una superficie para una fuente de intensidad  $I_l$  mediante la fórmula:

$$I_{l,\text{incident}} = I_l \cos \theta \quad (10.8)$$

Utilizando la Ecuación 10.8, podemos modelar las reflexiones difusas para una fuente luminosa con intensidad  $I_l$  utilizando la fórmula:

$$\begin{aligned} I_{l,\text{diff}} &= k_d I_{l,\text{incident}} \\ &= k_d I_l \cos \theta \end{aligned} \quad (10.9)$$

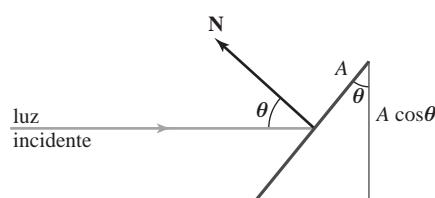
Cuando la luz que incide desde la fuente es perpendicular a la superficie en un punto concreto,  $\theta = 90^\circ$  y  $I_{l,\text{diff}} = k_d I_l$ . A medida que se incrementa el ángulo de incidencia, decrece la iluminación debida a esa fuente luminosa. Además, una superficie se verá iluminada por una fuente puntual únicamente si el ángulo de incidencia está comprendido entre  $0^\circ$  y  $90^\circ$  (es decir, si  $\cos \theta$  está en el intervalo que va de 0.0 a 1.0). Cuando  $\cos \theta < 0.0$ , la fuente luminosa estará situada detrás de la superficie.

En cualquier posición de la superficie, podemos designar al vector unitario normal como  $\mathbf{N}$  y al vector unitario en la dirección de una fuente puntual como  $\mathbf{L}$ , como en la Figura 10.13. Entonces,  $\cos \theta = \mathbf{N} \cdot \mathbf{L}$  y la ecuación de reflexión difusa para iluminación mediante una única fuente puntual en una determinada posición de la superficie puede expresarse en la forma:

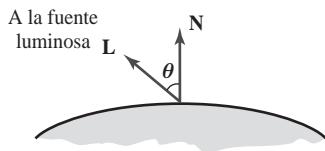
$$I_{l,\text{diff}} = \begin{cases} k_d I_l (\mathbf{N} \cdot \mathbf{L}), & \text{si } \mathbf{N} \cdot \mathbf{L} > 0 \\ 0.0, & \text{si } \mathbf{N} \cdot \mathbf{L} \leq 0 \end{cases} \quad (10.10)$$

El vector unitario  $\mathbf{L}$  en dirección a una fuente luminosa puntual cercana se calcula utilizando las coordenadas del punto de la superficie y de la fuente luminosa:

$$\mathbf{L} = \frac{\mathbf{P}_{\text{source}} - \mathbf{P}_{\text{surf}}}{|\mathbf{P}_{\text{source}} - \mathbf{P}_{\text{surf}}|} \quad (10.11)$$



**FIGURA 10.12.** Un área iluminada  $A$  proyectada en perpendicular al trayecto de los rayos de luz incidentes. Esta proyección perpendicular tiene un área igual a  $A \cos \theta$ .



**FIGURA 10.13.** Ángulo de incidencia  $\theta$  entre el vector unitario  $\mathbf{L}$  en dirección de la fuente luminosa y el vector unitario  $\mathbf{N}$  normal a la superficie en una determinada posición.

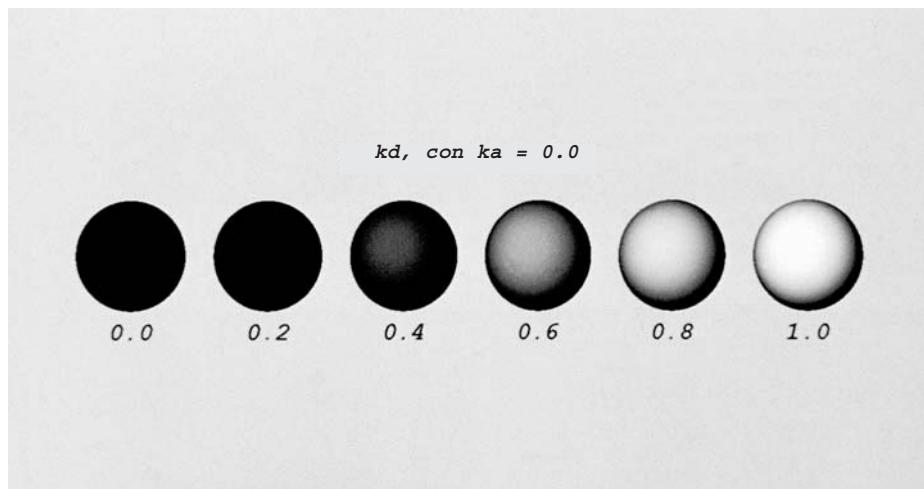
Sin embargo, una fuente luminosa situada en el «infinito» no tiene posición asignada, sino únicamente una dirección de propagación. En dicho caso, utilizaremos como vector de dirección  $\mathbf{L}$  el negado del vector que define la dirección de emisión de la fuente luminosa.

La Figura 10.14 ilustra la aplicación de la Ecuación 10.10 a una serie de posiciones sobre la superficie de una esfera, utilizando valores seleccionados del parámetro  $k_d$  comprendidos entre 0 y 1. Para  $k_d = 0$ , no se refleja nada de luz y la superficie del objeto parece negra. Los valores crecientes de  $k_d$  incrementan la intensidad de las reflexiones difusas, produciendo todos de gris cada vez más claros. A cada posición de píxel proyectada de la superficie se le asigna un valor de intensidad que se calcula mediante la ecuación de reflexión difusa. Las representaciones superficiales de esta figura ilustran la iluminación mediante una única fuente puntual, sin ningún efecto de iluminación adicional. Esto es lo que cabría esperar ver si apuntáramos con una linterna muy pequeña hacia un objeto en una habitación completamente oscura. En las escenas generales, sin embargo, existirán reflexiones superficiales debidas a luz ambiente, además de los efectos de iluminación producidos por la fuente luminosa.

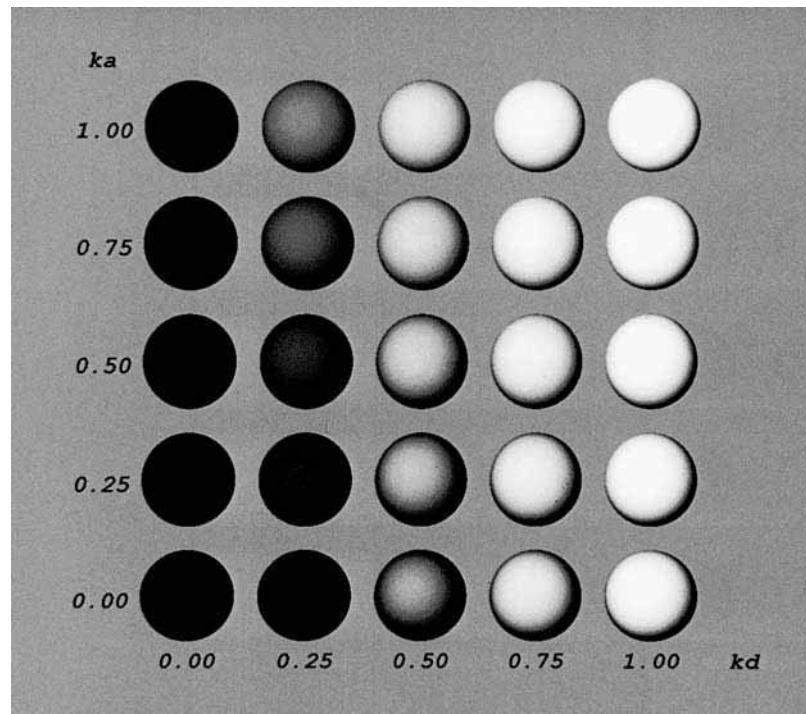
Podemos combinar los cálculos de intensidad debidos a la luz ambiente y a las fuentes puntuales con el fin de obtener una expresión para la reflexión difusa total en cada posición de una superficie. Además, muchos paquetes gráficos incluyen un **coeficiente de reflexión ambiente**  $k_a$  que puede asignarse a cada superficie para modificar la intensidad  $I_a$  de la luz ambiente. Esto simplemente nos proporciona un parámetro adicional para ajustar los efectos de iluminación de nuestro modelo empírico. Utilizando el modelo  $k_a$ , podemos escribir la ecuación total de reflexión difusa para una única fuente puntual en la forma

$$I_{\text{diff}} = \begin{cases} k_a I_a + k_d I_i(\mathbf{N} \cdot \mathbf{L}), & \text{si } \mathbf{N} \cdot \mathbf{L} > 0 \\ k_a I_a, & \text{si } \mathbf{N} \cdot \mathbf{L} \leq 0 \end{cases} \quad (10.12)$$

donde  $k_a$  y  $k_d$  dependen de las propiedades de los materiales de la superficie y tienen valores comprendidos en el rango que va de 0 a 1.0 para efectos de iluminación monocromáticos. La Figura 10.15 muestra una



**FIGURA 10.14.** Reflexiones difusas en una superficie esférica iluminada mediante una fuente luminosa puntual de color blanco, con valores del coeficiente de reflectividad difusa en el intervalo  $0 \leq k_d \leq 1$ .



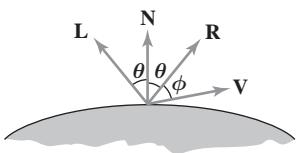
**FIGURA 10.15.** Reflexiones difusas en una superficie esférica iluminada con una luz ambiente de color gris oscuro y una fuente puntual de color blanco, utilizando cinco valores para  $k_a$  y  $k_d$  comprendidos entre 0.0 y 1.0.

esfera con intensidades superficiales calculadas según la Ecuación 10.12, para valores de los parámetros  $k_a$  y  $k_d$  comprendidos entre 0 y 1.0.

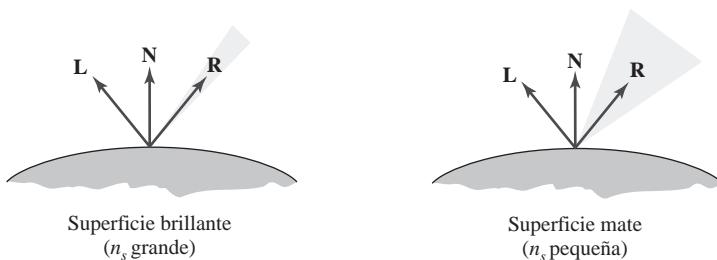
### Reflexión especular y modelo de Phong

El resalte o reflexión especular que podemos ver en las superficies brillantes es el resultado de una reflexión total, o casi total, de la luz incidente en una región concentrada alrededor del **ángulo de reflexión especular**. La Figura 10.16 muestra la dirección de reflexión especular para una determinada posición de una superficie iluminada. El ángulo de reflexión especular es igual al ángulo de la luz incidente, midiendo ambos ángulos en lados opuestos del vector unitario  $\mathbf{N}$  normal a la superficie. En esta figura,  $\mathbf{R}$  representa el vector unitario en la dirección de la reflexión especular ideal,  $\mathbf{L}$  es el vector unitario dirigido hacia la fuente luminosa puntual y  $\mathbf{V}$  es el vector unitario que apunta hacia el observador desde la posición seleccionada de la superficie. El ángulo  $\phi$  es el ángulo de visualización relativo a la dirección de reflexión especular  $\mathbf{R}$ . Para un reflector ideal (un espejo perfecto), la luz incidente se refleja sólo en la dirección de reflexión especular, y sólo podríamos ver la luz reflejada cuando los vectores  $\mathbf{V}$  y  $\mathbf{R}$  coincidieran ( $\phi = 0$ ).

Todos los objetos que no sean reflectores ideales exhiben reflexiones especulares en un rango finito de posiciones de visualización en torno al vector  $\mathbf{R}$ . Las superficies brillantes tienen un rango de reflexión especular estrecho, mientras que las superficies mates tienen un rango de reflexión más amplio. Un modelo empírico para el cálculo de reflexión especular, desarrollado por Phong Bui Tuong y denominado **modelo de reflexión especular de Phong** o simplemente **modelo de Phong**, define las intensidad de la reflexión especular como proporcionales a  $\cos_n s \phi$ . Al ángulo  $\phi$  pueden asignársele valores en el rango de  $0^\circ$  a  $90^\circ$ , de modo que  $\cos \phi$  varía de 0 a 1.0. El valor asignado al **exponente de reflexión especular**  $n_s$  estará determinado por el tipo de superficie que queramos mostrar. Una superficie muy brillante se modelará con un valor de  $n_s$  muy



**FIGURA 10.16.** El ángulo de reflexión especular es igual al ángulo de incidencia  $\theta$ .



**FIGURA 10.17.** Modelado de las reflexiones especulares (área sombreada) mediante el parámetro  $n_s$ .

grande (por ejemplo, 100 o más), mientras que los valores más pequeños (hasta como mínimo 1) se utilizan para las superficies más mates. Para un reflector perfecto,  $n_s$  es infinita. Para una superficie rugosa, a  $n_s$  se le asigna un valor próximo a 1. Las Figuras 10.17 y 10.18 muestran el efecto de  $n_s$  sobre el rango angular para el cual podemos esperar ver reflexiones especulares.

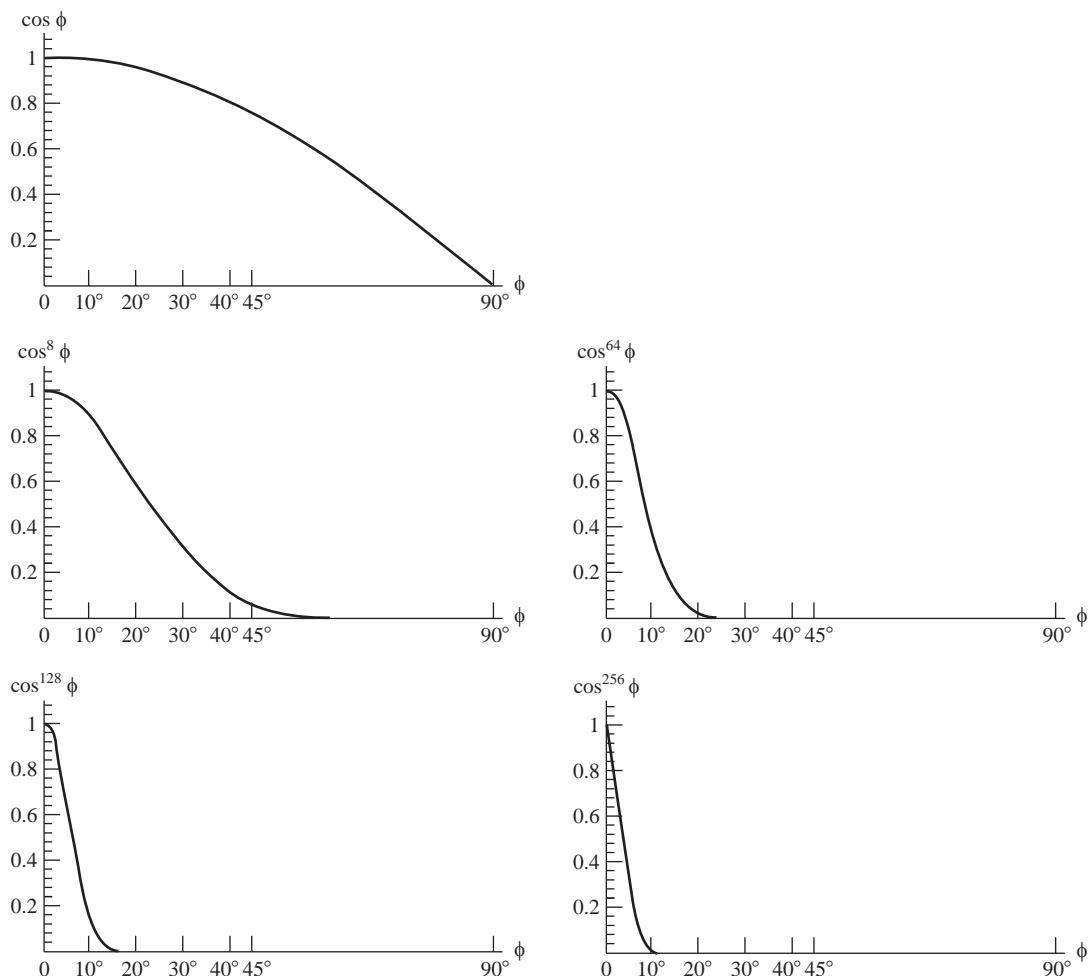
La intensidad de la reflexión especular depende de las propiedades de los materiales de la superficie y del ángulo de incidencia, así como de otros factores tales como la polarización y el color de la luz incidente. Podemos modelar aproximadamente las variaciones de intensidad especular monocromática utilizando un **coeficiente de reflexión especular**,  $W(\theta)$ , para cada superficie. La Figura 10.19 muestra la variación general de  $W(\theta)$  en el rango que va de  $\theta = 0^\circ$  a  $\theta = 90^\circ$  para unos cuantos materiales. En general,  $W(\theta)$  tiende a incrementarse a medida que aumenta el ángulo de incidencia. Para  $\theta = 90^\circ$ , toda la luz incidente se refleja ( $W(\theta) = 1$ ). La variación de la intensidad especular con respecto al ángulo de incidencia se describe mediante las *leyes de Fresnel de la reflexión*. Utilizando la función de reflexión espectral  $W(\theta)$ , podemos escribir el modelo de reflexión especular de Phong de la forma siguiente:

$$I_{l,\text{spec}} = W(\theta) I_l \cos^{n_s} \phi \quad (10.13)$$

donde  $I_l$  es la intensidad de la fuente luminosa y  $\phi$  es el ángulo de visualización relativo a la dirección de reflexión especular  $\mathbf{R}$ .

Como puede verse en la Figura 10.19, los materiales transparentes, como el cristal, exhiben reflexiones especulares apreciables únicamente cuando  $\theta$  se aproxima a  $90^\circ$ . Para  $\theta = 0$ , sólo se refleja aproximadamente el 4 por ciento de la luz que incide sobre una superficie de cristal, y para casi todo el rango de valores de  $\theta$ , la intensidad reflejada es inferior al 10 por ciento de la intensidad incidente. Pero para muchos materiales opacos, la reflexión especular es prácticamente constante para todos los ángulos de incidencia. En este caso, podemos modelar razonablemente los efectos especulares sustituyendo  $W(\theta)$  por un coeficiente constante de reflexión especular  $k_s$ . Entonces, simplemente asignamos a  $k_s$  algún valor en el rango de 0 a 1.0 para cada superficie.

Puesto que  $\mathbf{V}$  y  $\mathbf{R}$  son vectores unitarios en las direcciones de visualización y de reflexión especular, podemos calcular el valor de  $\cos \phi$  mediante el producto escalar  $\mathbf{V} \cdot \mathbf{R}$ . Además, no se generará ningún efecto especular para una superficie si  $\mathbf{V}$  y  $\mathbf{L}$  se encuentran en el mismo lado del vector normal  $\mathbf{N}$ , o si la fuente luminosa está situada detrás de la superficie. Así, asumiendo que el coeficiente de reflexión especular es constante para cada material, podemos determinar la intensidad de la reflexión especular debida a una fuente de iluminación puntual sobre una posición de la superficie mediante la fórmula:



**FIGURA 10.18.** Gráficas de  $\cos_{n_s} \phi$  utilizando cinco valores diferentes para el exponente de reflexión espectral  $n_s$ .

$$I_{l,\text{spec}} = \begin{cases} k_s I_l (\mathbf{V} \cdot \mathbf{R})^{n_s}, & \text{si } \mathbf{V} \cdot \mathbf{R} > 0 \text{ y } \mathbf{N} \cdot \mathbf{L} > 0 \\ 0,0, & \text{si } \mathbf{V} \cdot \mathbf{R} < 0 \text{ o } \mathbf{N} \cdot \mathbf{L} \leq 0 \end{cases} \quad (10.14)$$

La dirección de  $\mathbf{R}$ , el vector de reflexión, puede calcularse a partir de las direcciones de los vectores  $\mathbf{L}$  y  $\mathbf{N}$ . Como puede verse en la Figura 10.20, la proyección  $\mathbf{L}$  sobre la dirección del vector normal tiene una magnitud igual al producto escalar  $\mathbf{N} \cdot \mathbf{L}$ , que también es igual a la magnitud de la proyección del vector unitario  $\mathbf{R}$  sobre la dirección de  $\mathbf{N}$ .

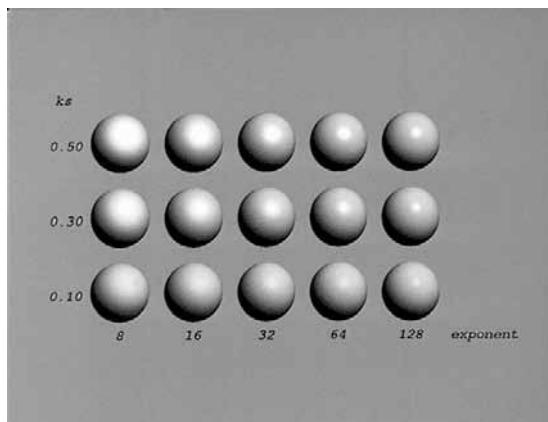
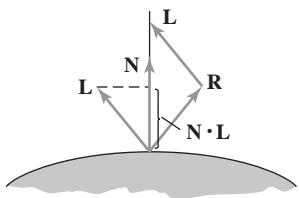
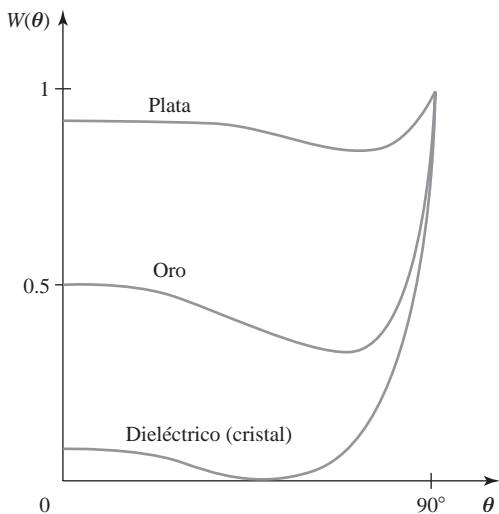
Por tanto, a partir de este diagrama, vemos que:

$$\mathbf{R} + \mathbf{L} = (2\mathbf{N} \cdot \mathbf{L})\mathbf{N}$$

y el vector de reflexión espectral puede calcularse como:

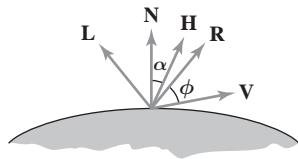
$$\mathbf{R} = (2\mathbf{N} \cdot \mathbf{L})\mathbf{N} - \mathbf{L} \quad (10.15)$$

La Figura 10.21 ilustra el fenómeno de la reflexión espectral para diversos valores de  $k_s$  y  $n_s$ , en una esfera iluminada mediante una única fuente puntual.



Para calcular  $\mathbf{V}$ , utilizamos la posición de la superficie y la posición de visualización, de la misma forma que obteníamos el vector unitario  $\mathbf{L}$  (Ecuación 10.11). Pero si se va a utilizar una dirección de visualización fija para todas las posiciones de una escena, podemos hacer  $\mathbf{V} = (0.0, 0.0, 1.0)$ , que es un vector unitario en la dirección  $z$  positiva. Los cálculos especulares requieren menos tiempo utilizando un valor de  $\mathbf{V}$  constante, aunque las imágenes no son tan realistas.

Puede obtenerse un modelo de Phong algo más simple utilizando el **vector medio  $\mathbf{H}$**  entre  $\mathbf{L}$  y  $\mathbf{V}$  para calcular el rango de reflexiones especulares. Si sustituimos  $\mathbf{V} \cdot \mathbf{R}$  en el modelo de Phong por el producto escalar  $\mathbf{N} \cdot \mathbf{H}$ , esto simplemente sustituye el cálculo empírico  $\cos \phi$  por el cálculo empírico  $\cos \alpha$  (Figura 10.22). El vector medio se obtiene como:

FIGURA 10.22. Vector medio **H** según la bisectriz del ángulo formado por **L** y **V**.

$$\mathbf{H} = \frac{\mathbf{L} + \mathbf{V}}{|\mathbf{L} + \mathbf{V}|} \quad (10.16)$$

Para superficies no planas,  $\mathbf{N} \cdot \mathbf{H}$  requiere menos cálculos que  $\mathbf{V} \cdot \mathbf{R}$ , porque el cálculo de  $\mathbf{R}$  en cada punto de la superficie implica al vector variable  $\mathbf{N}$ . Asimismo, si tanto el observador como la fuente luminosa están lo suficientemente lejos de la superficie, los vectores  $\mathbf{V}$  y  $\mathbf{L}$  son constantes, por lo que también  $\mathbf{H}$  será constante para todos los puntos de la superficie. Si el ángulo  $\mathbf{H}$  y  $\mathbf{N}$  es superior a  $90^\circ$ ,  $\mathbf{N} \cdot \mathbf{H}$  será negativo y asignaremos el valor 0.0 a la contribución correspondiente a la reflexión especular.

El vector **H** representa la dirección que produciría una reflexión especular máxima de la superficie en la dirección de visualización, para una posición dada de una fuente luminosa puntual. Por esta razón, **H** se denomina en ocasiones dirección de orientación de la superficie para máximo resalte. Asimismo, si el vector **V** es coplanar con los vectores **L** y **R** (y por tanto con **N**), el ángulo  $\alpha$  tiene el valor  $\phi/2$ . Cuando **V**, **L** y **N** no son coplanares,  $\alpha > \phi/2$ , dependiendo de la relación espacial de los tres vectores.

### Reflexiones difusa y especular combinadas

Para una única fuente luminosa puntual, podemos modelar las reflexiones difusa y especular combinadas para una posición de una superficie iluminada mediante la fórmula:

$$\begin{aligned} I &= I_{\text{diff}} + I_{\text{spec}} \\ &= k_a I_a + k_d I_l (\mathbf{N} \cdot \mathbf{L}) + k_s I_l (\mathbf{N} \cdot \mathbf{H})^{n_s} \end{aligned} \quad (10.17)$$

La superficie sólo estará iluminada por la luz ambiente cuando la fuente luminosa esté detrás de la superficie, y no habrá efectos especulares si **V** y **L** se encuentran en el mismo lado del vector normal **N**. La Figura 10.23 ilustra los efectos de iluminación superficial producidos por los diversos términos de la Ecuación 10.17.

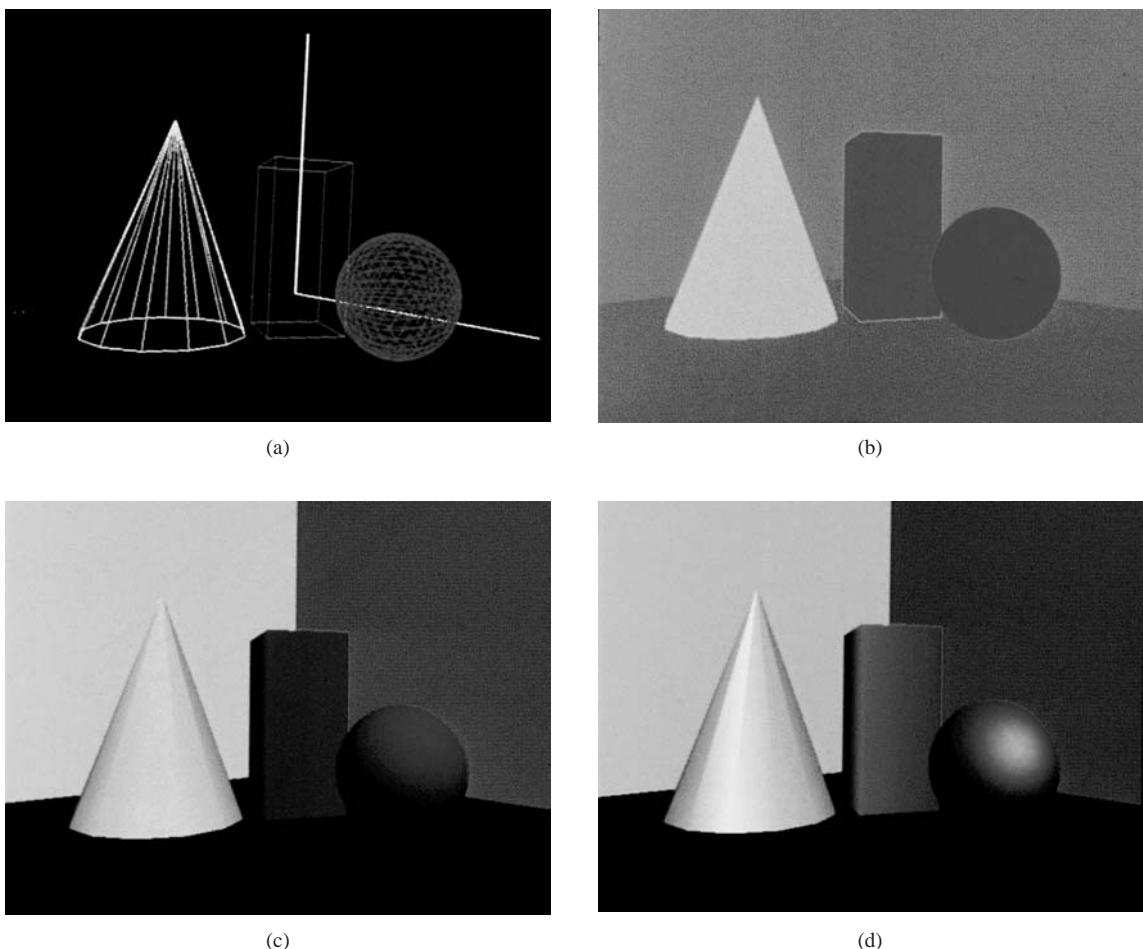
### Reflexiones especular y difusa para múltiples fuentes luminosas

Podemos colocar cualquier número de fuentes luminosas que deseemos en una escena. Para múltiples fuentes puntuales, calculamos las reflexiones difusa y especular como la suma de las contribuciones debidas a las diversas fuentes:

$$\begin{aligned} I &= I_{\text{ambdiff}} + \sum_{l=1}^n [I_{l,\text{diff}} + I_{l,\text{spec}}] \\ &= k_a I_a + \sum_{l=1}^n I_l [k_d (\mathbf{N} \cdot \mathbf{L}) + k_s (\mathbf{N} \cdot \mathbf{H})^{n_s}] \end{aligned} \quad (10.18)$$

### Emisión de luz superficial

Algunas superficies en una escena pueden emitir luz, además de reflejar la luz procedente de otras fuentes. Por ejemplo, una escena de una habitación puede contener lámparas, mientras que una escena nocturna de



**FIGURA 10.23.** Una escena alámbrica (a) se muestra en (b) utilizando únicamente luz ambiente, con un color distinto para cada objeto. Las reflexiones difusas resultantes de la iluminación con luz ambiente y una única fuente puntual se ilustran en (c). Para esta imagen,  $k_s = 0$  para todas las superficies. En (d) se muestran reflexiones tanto difusas como especulares para la iluminación con una única fuente puntual y con luz ambiente.

exteriores podría incluir farolas, anuncios luminosos y focos de coches. Podemos modelar empíricamente las emisiones de luz superficial incluyendo simplemente un término de emisión  $I_{\text{surfemission}}$  en el modelo de iluminación, de la misma forma que simulábamos la luz de fondo utilizando un nivel de luz ambiente. Esta emisión superficial se suma entonces a las reflexiones superficiales resultantes de las fuentes luminosas y de la luz de fondo.

Para iluminar otros objetos a partir de una superficie emisora de luz, podemos posicionar una fuente de luz direccional detrás de la superficie con el fin de producir un cono luminoso que atraviese la superficie. O bien podemos simular la emisión mediante un conjunto de fuentes luminosas puntuales distribuidas por toda la superficie. En general, sin embargo, las superficies de emisión no suelen utilizarse en el modelo básico de iluminación con el fin de iluminar otras superficies, debido al tiempo de cálculo adicional requerido. En lugar de eso, las emisiones superficiales se utilizan como forma simple de aproximar la apariencia de la superficie de una fuente luminosa compleja. Esto produce un efecto de resplandor para dicha superficie. En la Sección 10.12 hablaremos del modelo de radiosidad, que es un método más realista de modelar las emisiones de luz superficiales.

## Modelo básico de iluminación con focos y con atenuación de la intensidad

Podemos formular un modelo general de iluminación monocromática para las reflexiones superficiales que incluya múltiples fuentes luminosas puntuales, factores de atenuación, efectos de luz direccional (focos), fuentes situadas en el infinito y emisiones superficiales mediante la fórmula:

$$I = I_{\text{surfemission}} + I_{\text{ambdiff}} + \sum_{l=1}^n f_{l,\text{radatten}} f_{l,\text{angatten}} (I_{l,\text{diff}} + I_{l,\text{spec}}) \quad (10.19)$$

La función radial de atenuación  $f_{l,\text{radatten}}$  se evalúa mediante la Ecuación 10.2 y la función angular de atenuación mediante la Ecuación 10.5. Para cada fuente luminosa, calculamos la reflexión difusa en un punto de la superficie mediante la fórmula:

$$I_{l,\text{diff}} = \begin{cases} 0,0, & \text{si } \mathbf{N} \cdot \mathbf{L}_l \leq 0.0 \text{ (fuente luminosa detrás del objeto)} \\ k_d I_l (\mathbf{N} \cdot \mathbf{L}_l), & \text{en caso contrario} \end{cases} \quad (10.20)$$

Y el término de reflexión especular debido a la iluminación mediante una fuente puntual se calcula con expresiones similares:

$$I_{l,\text{spec}} = \begin{cases} 0,0, & \text{si } \mathbf{N} \cdot \mathbf{L}_l \leq 0.0 \\ k_s I_l \max \{ 0,0, (\mathbf{N} \cdot \mathbf{H}_l)^{n_s} \}, & \text{en caso contrario} \end{cases} \quad (10.21)$$

Para garantizar que la intensidad de cada píxel no exceda el valor máximo admisible, podemos aplicar algún tipo de procedimiento de normalización. Un enfoque simple consiste en definir una magnitud máxima para cada término de la ecuación de intensidad. Si alguno de los términos calculados excede del máximo, simplemente le asignamos el valor máximo. Otra forma de evitar los desbordamientos del valor de la intensidad consiste en normalizar los términos individuales, dividiendo cada uno de ellos por la magnitud del término más grande. Un procedimiento más complicado consiste en calcular todas las intensidades de píxel de la escena y luego cambiar la escala de este conjunto de intensidades al rango de intensidades que va de 0.0 a 1.0.

Asimismo, los valores de los coeficientes de la función radial de atenuación y los parámetros ópticos de las superficies de una escena pueden ajustarse para evitar que las intensidades calculadas excedan del valor máximo admisible. Este es un método muy efectivo para limitar los valores de intensidad cuando toda la escena está iluminada por una única fuente luminosa. En general, sin embargo, a las intensidades calculadas nunca se les permite exceder del valor 1.0 y los valores de intensidad negativos se ajustan al valor 0.0.

## Consideraciones relativas al color RGB

Para un modelo de color RGB, cada especificación de intensidad en el modelo de iluminación es un vector de tres elementos que indica las componentes roja, verde y azul de dicha intensidad. Así, para cada fuente luminosa,  $I_l = (I_{lR}, I_{lG}, I_{lB})$ . De modo similar, los coeficientes de reflexión también se especifican mediante componentes RGB:  $k_a = (k_{aR}, k_{aG}, k_{aB})$ ,  $k_d = (k_{dR}, k_{dG}, k_{dB})$  y  $k_s = (k_{sR}, k_{sG}, k_{sB})$ . Cada componente de color de la superficie se calcula entonces mediante una fórmula separada. Por ejemplo, la componente azul de las reflexiones difusa y especular para una fuente puntual se calcula a partir de las Ecuaciones 10.20 y 10.21 modificadas, de la forma siguiente:

$$I_{lB,\text{diff}} = k_{dB} I_{lB} (\mathbf{N} \cdot \mathbf{L}_l) \quad (10.22)$$

y

$$I_{l,\text{spec}} = k_{sB} I_{lB} \max \{ 0,0, (\mathbf{N} \cdot \mathbf{H}_l)^{n_s} \} \quad (10.23)$$

Lo más común es que las superficies se iluminen con fuentes de color blanco, pero para efectos especiales o para iluminación de interiores, podemos utilizar otros colores para las fuentes luminosas. A continuación, definimos los coeficientes de reflexión para modelar cada color de superficie concreto. Por ejemplo, si queremos que un objeto tenga una superficie azul, seleccionaremos un valor distinto de cero en el rango de 0.0 a 1.0 para la componente de reflectividad azul,  $k_{dB}$ , mientras que a las componentes de reflectividad roja y verde les asignaremos el valor cero ( $k_{dR} = k_{dG} = 0.0$ ). Todas las componentes rojas y verdes distintas de cero en la luz incidente serán absorbidas y sólo se reflejará la componente azul.

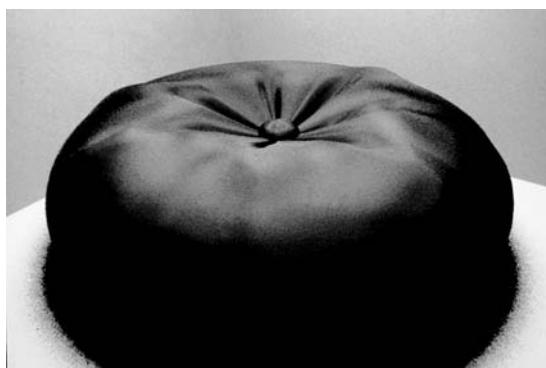
En su modelo de reflexión especular original, Phong asignaba el parámetro  $k_s$  a un valor constante, independiente del color de la superficie. Esto produce reflexiones especulares que tienen el mismo color que la luz incidente (usualmente blanco), lo que da a la superficie una apariencia plástica. Para que el material no tenga aspecto plástico, el color de la reflexión especular debe definirse en función de las propiedades de la superficie y puede ser diferente tanto del color de la luz incidente como del color de las reflexiones difusas. Podemos aproximar los efectos especulares en tales superficies haciendo que el coeficiente de reflexión especular dependa del color, como en la Ecuación 10.23. La Figura 10.24 ilustra las reflexiones de color en una superficie mate, mientras que las Figuras 10.25 y 10.26 muestran las reflexiones de color en superficies metálicas. En la Figura 10.27 se muestran reflexiones de luz en las superficies de los objetos debidas a múltiples fuentes de luz coloreada.

Otro método para establecer el color de la superficie consiste en especificar las componentes de los vectores de color difuso y especular para cada superficie, pero conservando los coeficientes de reflexión como constantes de un único valor. Para un modelo de color RGB, por ejemplo, las componentes de estos dos vectores de color de superficie podrían designarse ( $S_{dR}, S_{dG}, S_{dB}$ ) y ( $S_{sR}, S_{sG}, S_{sB}$ ). La componente azul de la reflexión difusa (Ecuación 10.22) se calcularía entonces como:

$$I_{IB,diff} = k_d S_{dB} I_{IB} (\mathbf{N} \cdot \mathbf{L}_l) \quad (10.24)$$

Esta técnica proporciona una flexibilidad algo mayor, ya que pueden configurarse independientemente los parámetros de color de la superficie y los valores de reflectividad.

En algunos paquetes gráficos, se incluyen parámetros de iluminación adicionales que permiten asignar múltiples colores a una fuente luminosa, contribuyendo cada color a uno de los efectos de iluminación superficial. Por ejemplo, puede utilizarse uno de los colores como contribución a la iluminación general de fondo de la escena. De forma similar, otro color de la fuente luminosa puede usarse como intensidad luminosa para



**FIGURA 10.24.** Reflexiones luminosas en la superficie de un cojín negro de nylon, modelado mediante patrones de tela tejida y representado utilizando métodos de Monte-Carlo para trazado de rayos. (Cortesía de Stephen H. Westin, Program of Computer Graphics, Cornell University.)



**FIGURA 10.25.** Reflexiones luminosas en una tetera cuyos parámetros de reflexión se han especificado para simular superficies de aluminio bruñido y que ha sido representada utilizando métodos de Monte-Carlo para trazado de rayos. (Cortesía de Stephen H. Westin, Program of Computer Graphics, Cornell University.)



**FIGURA 10.26.** Reflexiones luminosas en trompetas cuyos parámetros de reflexión se han definido para simular superficies brillantes de cobre. (Cortesía de SOFTIMAGE, Inc.)



**FIGURA 10.27.** Reflexiones luminosas debidas a múltiples fuentes de luz de varios colores. (Cortesía de Sun Micro-systems.)

los cálculos de reflexión difusa, mientras que un tercer color de la fuente podría emplearse en los cálculos de la reflexión especular.

### Otras representaciones del color

Podemos describir los colores utilizando otros modelos distintos de la representación RGB. Por ejemplo, un color puede representarse utilizando las componentes cyan, magenta y amarillo, o bien describirlo en términos de un tono concreto y los niveles percibidos de brillo y de saturación del color. Podemos incorporar cualquiera de estas representaciones, incluyendo especificaciones de color con más de tres componentes, en nuestro modelo de iluminación. Como ejemplo, la Ecuación 10.24 puede expresarse en términos de cualquier color espectral de longitud de onda  $\lambda$  como:

$$I_{\lambda, \text{diff}} = k_d S_{d\lambda} I_{\lambda} (\mathbf{N} \cdot \mathbf{L}_l) \quad (10.25)$$

En el Capítulo 12 se explican con mayor detalle las diversas representaciones del color que resultan útiles en aplicaciones de infografía.

### Luminancia

Otra característica del color es la **luminancia**, que en ocasiones se denomina también energía luminosa. La luminancia proporciona información acerca del nivel de claridad u oscuridad de un color, y es una medida psicológica de nuestra percepción del brillo que, varía con la cantidad de iluminación que observemos.

Físicamente, el color se describe en términos del rango de frecuencias de la energía radiante visible (luz) y la luminancia se calcula como una suma ponderada de las componentes de intensidad dentro de un entorno de iluminación concreto. Puesto que cualquier tipo de iluminación contiene un rango continuo de frecuencias, el valor de luminancia se calcula como:

$$\text{luminancia} = \int_{\text{visible } f} p(f) I(f) df \quad (10.26)$$

El parámetro  $I(f)$  de este cálculo representa la intensidad de la componente luminosa de frecuencia  $f$  que está radiando en una dirección concreta. El parámetro  $p(f)$  es una función de proporcionalidad experimen-

talmente determinada que varía tanto con la frecuencia como con el nivel de iluminación. La integral se realiza para todas las intensidades a lo largo del rango de frecuencias contenido en la luz.

Para imágenes en escala de grises y monocromáticas, nos basta con los valores de luminancia para describir la iluminación de un objeto. Y de hecho, algunos paquetes gráficos permiten expresar los parámetros de iluminación en términos de la luminancia. Las componentes de color verde de una fuente luminosa son las que más contribuyen a la luminancia, mientras que las componentes de color azul son las que contribuyen menos. Por tanto, la luminancia de una fuente de color RGB se suele calcular mediante la fórmula:

$$\text{luminancia} = 0.299R + 0.587G + 0.114B \quad (10.27)$$

En ocasiones, pueden conseguirse mejores efectos de iluminación incrementando la contribución de la componente verde de cada color RGB. Una recomendación empírica para este cálculo es la fórmula  $0.2125R + 0.7154G + 0.0721B$ . El parámetro de luminancia suele representarse mediante el símbolo  $Y$ , que se corresponde con la componente  $Y$  del modelo de color  $XYZ$  (Sección 12.3).

## 10.4 SUPERFICIES TRANSPARENTES

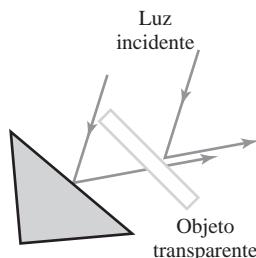
---

Podemos describir un objeto, como el cristal de una ventana, como *transparente* si podemos ver las cosas que están situadas detrás del objeto. De forma similar, si no podemos ver las cosas que están detrás del objeto, diremos que el objeto es *opaco*. Además, algunos objetos transparentes, como el cristal esmerilado y ciertos materiales plásticos, son **translúcidos**, de modo que la luz transmitida se difunde en todas direcciones. Los objetos visualizados a través de materiales translúcidos parecen borrosos y a menudo no se les puede identificar claramente.

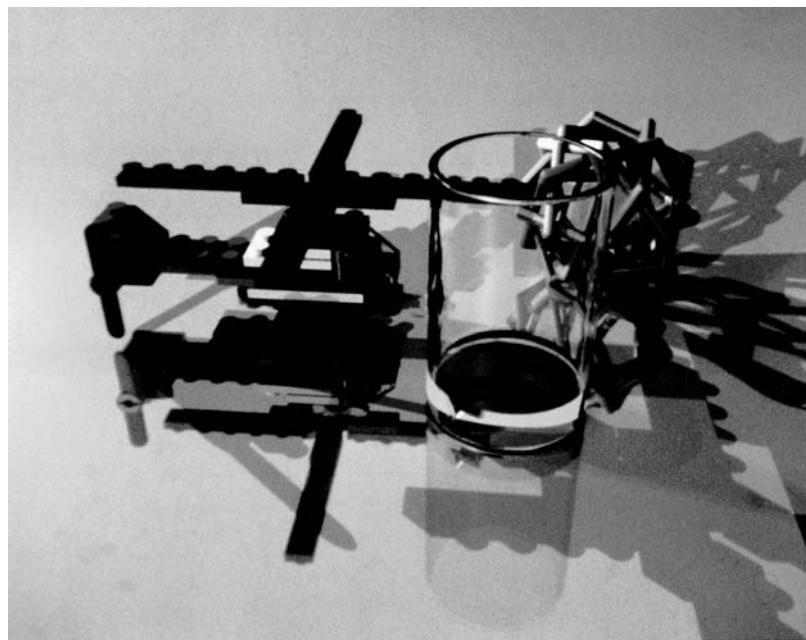
Una superficie transparente, en general, produce luz tanto reflejada como transmitida. La luz transmitida a través de la superficie es el resultado de emisiones y reflexiones de los objetos y de las fuentes situadas detrás del objeto transparente. La Figura 10.28 ilustra las contribuciones de intensidad a la iluminación superficial para un objeto transparente que se encuentre delante de un objeto opaco. La Figura 10.29, por su parte, muestra los efectos de transparencia que pueden conseguirse en una escena generada por computadora.

### Materiales translúcidos

En la superficie de un objeto transparente puede producirse tanto transmisión difusa como especular. Los efectos difusos tienen gran importancia cuando haya que modelar materiales translúcidos. La luz que pasa a través de un material translúcido se dispersa, de modo que los objetos situados en segundo plano se ven como imágenes borrosas. Podemos simular la transmisión difusa distribuyendo las contribuciones de intensidad de los objetos de segundo plano a lo largo de un área finita, o bien utilizar métodos de trazado de rayos para simular los objetos translúcidos. Estas manipulaciones requieren mucho tiempo de procesamiento, por lo que los modelos básicos de iluminación sólo suelen calcular los efectos de transparencia especular.



**FIGURA 10.28.** La emisión de luz de una superficie transparente es, en general, una combinación de luz reflejada y transmitida.



**FIGURA 10.29.** Una vista de una escena obtenida mediante trazado de rayos, donde se muestra un vaso transparente. Se pueden apreciar tanto transmisiones de luz procedentes de los objetos situados detrás del vaso como reflexiones lumínicas producidas en la superficie del vaso. (Cortesía de Eric Haines, Autodesk, Inc.)

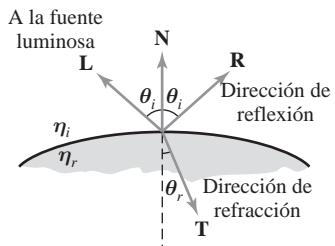
## Refracción de la luz

Pueden obtenerse imágenes realistas de un material transparente modelando el trayecto de **refracción** de un rayo de luz a través del material. Cuando un haz luminoso incide sobre una superficie transparente, parte del mismo se refleja y parte se transmite a través del material, en forma de luz refractada, como se muestra en la Figura 10.30. Puesto que la velocidad de la luz es diferente para los distintos materiales, el trayecto de la luz refractada será distinto del de la luz incidente. La dirección de la luz refractada, especificada por el **ángulo de refracción** con respecto al vector normal a la superficie, está en función del **índice de refracción** del material y de la dirección de la luz incidente. El índice de refracción se define como el cociente entre la velocidad de la luz en el vacío y la velocidad de la luz en el material. El ángulo de refracción  $\theta_r$  se calcula aplicando la **ley de Snell**:

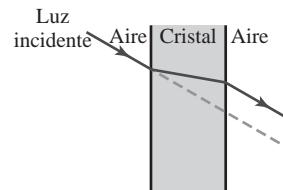
$$\sin \theta_r = \frac{\eta_i}{\eta_r} \sin \theta_i \quad (10.28)$$

donde  $\theta_i$  es el ángulo de incidencia,  $\eta_i$  es el índice de refracción del material a través del que viajaba la luz y  $\eta_r$  es el índice de refracción del material a través del que la luz se refracta.

De hecho, el índice de refracción también depende de otros factores, como la temperatura del material y la longitud de onda de la luz incidente. Así, las diversas componentes de color de la luz incidente blanca, por ejemplo, se refractan con ángulos distintos que varían con la temperatura. Además, dentro de los materiales anisótropos como el cuarzo cristalino, la velocidad de la luz depende de la dirección, y algunos materiales transparentes exhiben una *doble reflexión*, que hace que se generen dos rayos luminosos refractados. Para la mayoría de las aplicaciones, sin embargo, podemos utilizar un único índice de refracción promedio para cada material, tal como se numera en la Tabla 10.1. Utilizando el índice de refracción del aire (aproximadamente 1.0) que rodea a un panel de cristal ( $\text{índice de refracción} \approx 1.61$ ) en la Ecuación 10.28, con un ángulo de incidencia de  $30^\circ$ , obtenemos un ángulo de refracción de unos  $18^\circ$  para la luz que pasa a través del cristal.



**FIGURA 10.30.** Dirección de reflexión **R** y dirección de refracción (transmisión) **T** para un rayo de luz que incide sobre una superficie con un índice de refracción  $n_r$ .



**FIGURA 10.31.** Refracción de la luz a través de un panel de cristal. El rayo refractado emergente describe una trayectoria que es paralela a la del rayo luminoso incidente (línea punteada).

**TABLA 10.1.** ÍNDICE PROMEDIO DE REFRACCIÓN PARA ALGUNOS MATERIALES COMUNES.

Material	Índice de refracción
Vacio (aire u otro gas)	1.00
Cristal común	1.52
Cristal pesado	1.61
Cristal de sílex común	1.61
Cristal de sílex pesado	1.92
Sal cristalina	1.55
Cuarzo	1.54
Agua	1.33
Hielo	1.31

La Figura 10.31 ilustra las modificaciones del trayecto debidas a la refracción para un rayo de luz que atraviesa una fina lámina de cristal. El efecto global de la refracción consiste en desplazar la luz incidente hasta una trayectoria paralela cuando el rayo emerge del material. Puesto que los cálculos relacionados con las funciones trigonométricas de la Ecuación 10.28 requieren mucho esfuerzo de procesamiento, estos efectos de refracción pueden aproximarse simplemente desplazando la trayectoria de la luz incidente según una cantidad apropiada, que dependerá de cada material.

A partir de la ley de Snell y del diagrama de la Figura 10.30, podemos obtener el vector unitario de transmisión **T** en la dirección de refracción  $\theta_r$  mediante la fórmula:

$$\mathbf{T} = \left( \frac{\eta_i}{\eta_r} \cos \theta_i - \cos \theta_r \right) \mathbf{N} - \frac{\eta_i}{\eta_r} \mathbf{L} \quad (10.29)$$

donde **N** es el vector unitario normal a la superficie y **L** es el vector unitario en la dirección que va desde el punto de la superficie hasta la fuente luminosa. El vector de transmisión **T** puede utilizarse para localizar las intersecciones del trayecto de refracción con los objetos situados detrás de la superficie transparente. Incluir los objetos de refracción en una escena puede producir imágenes muy realistas, pero la determinación de los trayectos de refracción y las intersecciones con los objetos requiere una cantidad de proceso considerable. La mayoría de los métodos del espacio de imagen basados en líneas de barrido modelan la transmisión de la luz mediante aproximaciones que reducen el tiempo de procesamiento. Los efectos de refracción precisos sólo se suelen mostrar utilizando algoritmos de trazado de rayos (Sección 10.11).

## Modelo básico de transparencia

Un procedimiento más simple para modelar los objetos transparentes consiste en ignorar los desplazamientos de los trayectos debidos a la refracción. En la práctica, este enfoque equivale a suponer que no hay ningún cambio en el índice de refracción de un material a otro, de modo que el ángulo de refracción es siempre igual al ángulo de incidencia. Este método acelera los cálculos de las intensidades y puede producir efectos de transparencia razonables para superficies poligonales finas.

Podemos combinar la intensidad transmitida  $I_{\text{trans}}$  a través de una superficie transparente desde un objeto situado en segundo plano con la intensidad reflejada  $I_{\text{refl}}$  por la propia superficie (Figura 10.32) utilizando un **coeficiente de transparencia**  $k_t$ . Al parámetro  $k_t$  se le asigna un valor entre 0.0 y 1.0 para especificar qué porcentaje de la luz procedente de los objetos situados en segundo plano hay que transmitir. La intensidad total en la superficie se calcula entonces como:

$$I = (1 - k_t)I_{\text{refl}} + k_t I_{\text{trans}} \quad (10.30)$$

El término  $(1 - k_t)$  es el **factor de opacidad**. Por ejemplo, si el factor de transparencia tiene un valor de 0.3, entonces el 30 por ciento de la luz de los objetos de segundo plano se combinará con un 70 por ciento de la luz reflejada por la superficie.

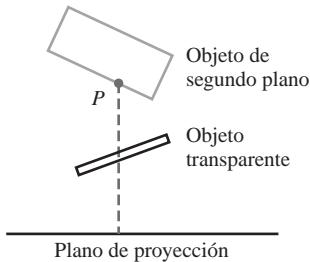
Este procedimiento puede usarse para combinar los efectos de iluminación de cualquier número de objetos transparentes y opacos, siempre y cuando procesemos las superficies según su orden de profundidad (desde atrás hacia delante). Por ejemplo, mirando a través del cristal de la Figura 10.29, podemos ver los objetos opacos que están situados detrás de dos superficies transparentes. De forma similar, cuando miramos a través del parabrisas de un automóvil, podemos ver los objetos situados dentro del vehículo, así como cualquier objeto que esté situado detrás del parabrisas trasero.

Para objetos muy transparentes, asignaremos a  $k_t$  un valor próximo a 1.0. Los objetos casi opacos transmiten muy poca luz procedente de los objetos situados en segundo plano, así que podemos asignar a  $k_t$  un valor próximo a 0.0 para estos materiales. También se puede hacer que  $k_t$  sea función de la posición concreta de la superficie, de tal forma que las diferentes partes de un objeto transmitan más o menos luz procedente de las superficies situadas en segundo plano.

Podemos modificar los algoritmos de visibilidad basados en la ordenación de la profundidad con el fin de tener en cuenta la transparencia, para lo cual ordenaremos primero las superficies según su profundidad y luego determinaremos si cualquiera de las superficies visibles es transparente. Si lo es, la intensidad reflejada en la superficie se combinará con la intensidad superficial de los objetos situados detrás suyo, con el fin de obtener la intensidad de píxel en cada punto proyectado de la superficie.

Los efectos de transparencia también pueden implementarse utilizando una técnica de búfer de profundidad modificada. Podemos dividir las superficies de una escena en dos grupos, de modo que se procesen primero todas las superficies opacas. Terminado este proceso, el búfer de imagen contendrá las intensidades de las superficies visibles y el búfer de profundidad contendrá sus profundidades. A continuación, se compara la profundidad de los objetos transparentes con los valores previamente almacenados en el búfer de profundidad. Si alguna de las superficies transparentes es visible, su intensidad reflejada se calculará y combinará con la intensidad de la superficie opaca previamente almacenada en el búfer de imagen. Este método puede modificarse para producir imágenes más precisas, utilizando espacio de almacenamiento adicional para la profundidad y para otros parámetros de las superficies transparentes. Esto permite comparar los valores de profundidad de las superficies transparentes entre sí, además de compararlos con la profundidad de las superficies opacas. Entonces, las superficies transparentes visibles se representarán combinando sus intensidades superficiales con aquellas de las superficies visibles y opacas que estén situadas detrás.

Otra posible técnica es el método basado en búfer A. Para cada posición de píxel del búfer A, los parches de superficie de todas las superficies solapadas se guardan y almacenan según su orden de profundidad. Entonces, las intensidades de los parches de superficie transparentes y opacos que se solapan en profundidad se combinan en el orden de visibilidad adecuado con el fin de producir la intensidad final promedio para el píxel.



**FIGURA 10.32.** La intensidad de un objeto situado en segundo plano en el punto **P** puede combinarse con la intensidad reflejada por la superficie de un objeto transparente a lo largo de una línea de proyección perpendicular (punteada).

## 10.5 EFECTOS ATMOSFÉRICOS

Otro factor que a veces se incluye en los modelos de iluminación es el efecto de la atmósfera sobre el color de un objeto. Una atmósfera neblinosa hace que los colores se difuminen y que los objetos parezcan más tenues. Por tanto, podríamos especificar una función que modificara los colores de las superficies de acuerdo con la cantidad de polvo, humo o niebla que queramos simular en la atmósfera. El efecto de atmósfera neblinosa se suele simular mediante una función exponencial de atenuación tal como:

$$f_{\text{atmo}}(d) = e^{-\rho d} \quad (10.31)$$

o

$$f_{\text{atmo}}(d) = e^{-(\rho d)^2} \quad (10.32)$$

El valor asignado a  $d$  es la distancia hasta el objeto desde la posición de visualización. El parámetro  $\rho$  en estas funciones exponenciales se utiliza para definir un valor de densidad positivo para la atmósfera. Los valores mayores de  $\rho$  producen una atmósfera más densa y hacen que se atenúen más los colores de las superficies. Después de calculado el color de la superficie de un objeto, multiplicaremos dicho color por una de las funciones atmosféricas con el fin de reducir su intensidad según una cantidad que dependerá del valor de densidad que hayamos asignado a la atmósfera.

En lugar de la función exponencial, podríamos simplificar los cálculos de la atenuación atmosférica utilizando la función lineal 9.9 de variación de la intensidad según la profundidad. Esto hace que se reduzca la intensidad de los colores de la superficie de los objetos distantes, aunque si hacemos esto no tendremos posibilidad de variar la densidad de la atmósfera.

Algunas veces puede ser necesario también simular un color atmosférico. Por ejemplo, el aire en una habitación llena de humo podría modelarse con un cierto tono gris o, quizás, con un azul pálido. Podría emplearse el siguiente cálculo para combinar el color de la atmósfera con el color de un objeto:

$$I = f_{\text{atmo}}(d)I_{\text{obj}} + [1 - f_{\text{atmo}}(d)]I_{\text{atmo}} \quad (10.33)$$

donde  $f_{\text{atmo}}$  es una función exponencial o lineal de atenuación atmosférica.

## 10.6 SOMBRAS

Pueden utilizarse métodos de detección de la visibilidad para localizar regiones que no estén iluminadas por las fuentes de luz. Con la posición de visualización situada en la ubicación de una fuente luminosa, podemos determinar qué secciones de las superficies de una escena no son visibles. Éstas serán las áreas de sombra. Una vez determinadas las áreas de sombra para todas las fuentes luminosas, las sombras pueden tratarse como patrones superficiales y almacenarse en matrices de memoria de patrones. La Figura 10.33 ilustra una serie de regiones de sombra sobre la cara de un carácter animado. En esta imagen, las regiones de sombra son secciones de la superficie que no son visibles desde la posición de la fuente luminosa que está situada encima de



**FIGURA 10.33.** Patrones de sombra mapeados sobre la cara de Aki Ross, un carácter animado de la película *Final Fantasy: The Spirits Within*. (Cortesía de Square Pictures, Inc. © 2001 FFFP. Todos los derechos reservados.)

la figura. Así, la mano y el brazo levantados son iluminados, pero las secciones de la cara situadas detrás del brazo, según la línea de visión que proviene de la fuente luminosa, estarán en sombras. La escena de la Figura 10.29 muestra los efectos de las sombras producidas por múltiples fuentes luminosas.

Los patrones de sombra generados mediante un método de detección de superficies visibles son válidos para cualquier posición de visualización seleccionada, mientras no se varíen las posiciones de las fuentes luminosas. Las superficies que sean visibles desde la posición de visualización se sombrean de acuerdo con el modelo de iluminación, que puede combinarse con patrones de texturas. Podemos mostrar las áreas de sombras únicamente con la intensidad de la luz ambiente, o podemos combinar la luz ambiente con una serie de texturas de superficie especificadas.

## 10.7 PARÁMETROS DE LA CÁMARA

---

Los procedimientos de visualización e iluminación que hemos considerado hasta ahora producen imágenes nítidas, que son equivalentes a fotografiar una escena con una cámara tradicional. Sin embargo, cuando fotografiamos una escena real, podemos ajustar la cámara de modo que sólo algunos de los objetos estén enfocados. Los demás objetos estarán más o menos desenfocados, dependiendo de la distribución en profundidad de los objetos de la escena. Podemos simular la apariencia de los objetos desenfocados en un programa infográfico proyectando cada posición de esos objetos sobre un área que cubra múltiples posiciones de píxel, mezclando los colores del objeto con los de otros objetos con el fin de producir un patrón de proyección borroso. Este procedimiento es similar a los métodos utilizados en *antialiasing*, y podemos incorporar estos efectos de la cámara tanto en los algoritmos de línea de proyección como en los de trazado de rayos. Las escenas generadas por computadora parecen más realistas cuando se incluyen los efectos de enfoque, pero estos cálculos de enfoque requieren mucho tiempo de procesamiento. En la Sección 10.11 se analizan los métodos de especificación de los parámetros de la cámara y del objetivo para simular los efectos de enfoque.

## 10.8 VISUALIZACIÓN DE LA INTENSIDAD DE LA LUZ

---

Una intensidad superficial calculada mediante un modelo de iluminación puede tener cualquier valor en el rango que va de 0.0 a 1.0, pero un sistema de gráficos por computadora sólo puede mostrar un conjunto limitado de intensidades, por tanto, los valores de intensidad calculados deben convertirse a uno de los valores

permitidos en el sistema. Además, el número de niveles de intensidad permitidos en el sistema puede distribuirse de modo que se correspondan con la forma en que nuestros ojos perciben las diferencias de intensidad. Cuando mostramos escenas en un sistema monocromo, podemos convertir las intensidades calculadas en patrones de semitono, como se explica en la Sección 10.9.

## Distribución de los niveles de intensidad del sistema

Para cualquier sistema, el número de niveles de intensidad permitidos puede distribuirse en el rango de 0.0 a 1.0 de modo que esta distribución se corresponda con nuestra percepción de lo que son intervalos de intensidad iguales entre niveles. Los humanos percibimos las intensidades relativas de la luz de la misma forma en que percibimos las intensidades relativas de sonido. Según una escala logarítmica. Esto significa que si el cociente de dos valores de intensidad es igual al cociente de otras dos intensidades, percibiremos que la diferencia entre cada par de intensidades es la misma. Como ejemplo, el hombre percibe la diferencia entre las intensidades 0.20 y 0.22 igual que la diferencia entre 0.80 y 0.88. Por tanto, para mostrar  $n+1$  niveles de intensidad sucesivos con una diferencia percibida de brillo igual, los niveles de intensidad en el monitor deben espaciarse de modo que el cociente de las intensidades sucesivas sea constante:

$$\frac{I_1}{I_0} = \frac{I_2}{I_1} = \dots = \frac{I_n}{I_{n-1}} = r \quad (10.34)$$

donde  $I$  representa la intensidad de una de las componentes de color de un haz luminoso. El nivel más bajo que puede mostrarse se representa como  $I_0$  y el nivel más alto como  $I_n$ . Cualquier nivel de intensidad intermedio podrá entonces expresarse en términos de  $I_0$  como:

$$I_k = r^k I_0 \quad (10.35)$$

Podemos calcular el valor de  $r$  a partir de los valores de  $I_0$  y  $n$  de un sistema concreto efectuando la sustitución  $k = n$  en la expresión anterior. Puesto  $I_n = 1.0$ , tendremos:

$$r = \left( \frac{1.0}{I_0} \right)^{1/n} \quad (10.36)$$

Por tanto, la fórmula de  $I_k$  de la Ecuación 10.35 puede reescribirse como:

$$I_k = I_0^{(n-k)/n} \quad (10.37)$$

Como ejemplo, si  $I_0 = \frac{1}{8}$  para un sistema con  $n = 3$ , tendremos  $r = 2$  y los cuatro valores de intensidad serán  $\frac{1}{8}, \frac{1}{4}, \frac{1}{2}$  y 1.0.

El valor de intensidad más bajo  $I_0$  depende de las características del monitor y se encuentra normalmente en el rango comprendido entre 0.005 y aproximadamente 0.025. Esta intensidad residual en un monitor de vídeo se debe a la luz reflejada por los fósforos de la pantalla. Por tanto, una región «negra» de la pantalla siempre tendrá un cierto valor de intensidad por encima de 0.0. Para una imagen en escala de grises con 8 bits por píxel ( $n = 255$ ) e  $I_0 = 0.01$ , el cociente de intensidades sucesivas es aproximadamente  $r = 1.0182$ . Los valores aproximados de las 256 intensidades de este sistema serán 0.0100, 0.0102, 0.0104, 0.0106, 0.0107, 0.0109, ..., 0.9821 y 1.0000.

Con las componentes de color RGB se utilizan métodos similares. Por ejemplo, podemos expresar la intensidad de la componente azul de un color en el nivel  $k$  en términos del menor valor de intensidad azul obtenible, mediante la fórmula:

$$I_{Bk} = r_B^k I_{B0} \quad (10.38)$$

donde,

$$r_B = \left( \frac{1.0}{I_{B0}} \right)^{1/n} \quad (10.39)$$

y  $n$  es el número de niveles de intensidad.

## Corrección gamma y tablas de sustitución de vídeo

Cuando mostramos imágenes en color o monocromáticas sobre un monitor de vídeo, las variaciones de brillo percibidas son no lineales, mientras que los modelos de iluminación producen una variación lineal de los valores de intensidad. El color RGB (0.25, 0.25, 0.25) obtenido a partir del modelo de iluminación representa la mitad de intensidad del color (0.5, 0.5, 0.5). Usualmente, estas intensidades calculadas se almacenan en un archivo de imagen en forma de valores enteros que van de 0 a 255, con un byte para cada una de las tres componentes RGB. Este archivo de imagen que describe las intensidades también es lineal, por lo que un píxel con el valor (64, 64, 64) representará la mitad de intensidad de un píxel que tenga el valor (128, 128, 128). Las tensiones del cañón de electrones, que controla el número de electrones que inciden sobre la pantalla de fósforo, produce niveles de brillo que están determinados por la **curva de respuesta del monitor** que se muestra en la Figura 10.34. Por tanto, el valor de intensidad mostrado (64, 64, 64) no parecerá la mitad de brillante que el valor (128, 128, 128).

Para compensar las no linealidades del monitor, los sistemas gráficos utilizan una **tabla de consulta de vídeo** que ajusta los valores lineales de intensidad de entrada. La curva de respuesta del monitor está descrita mediante la función exponencial:

$$I = aV^\gamma \quad (10.40)$$

La variable  $I$  representa la intensidad mostrada y el parámetro  $V$  es la correspondiente tensión en el tubo de rayos catódicos. Los valores de los parámetros  $a$  y  $\gamma$  dependen de las características del monitor utilizado dentro del sistema gráfico. Así, si queremos mostrar un valor de intensidad concreto  $I$ , el valor de tensión que permitirá generar esta intensidad es:

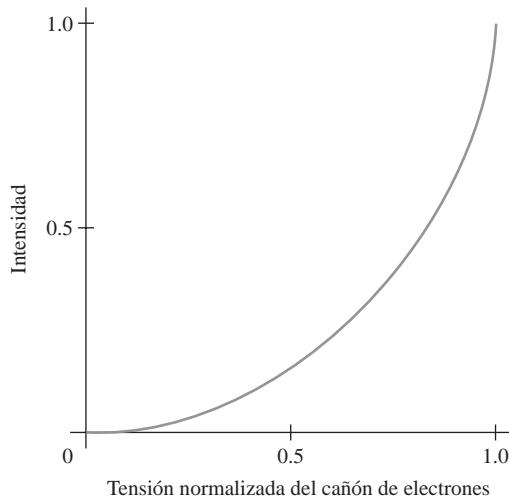
$$V = \left( \frac{I}{a} \right)^{1/\gamma} \quad (10.41)$$

Este cálculo se denomina **corrección gamma** de la intensidad, y los valores de gamma están normalmente en el rango comprendido entre 1.7 y 2.3. El estándar de señal de NTSC (National Television System Committee) es  $\gamma=2.2$ . La Figura 10.35 muestra un curva de corrección gamma utilizando el valor de gamma de NTSC, estando tanto la intensidad como la tensión normalizadas en el intervalo de 0 a 1.0. La Ecuación 10.41 se utiliza para definir la tabla de consulta de vídeo que convierte los valores enteros de intensidad de un archivo de imagen a valores que controlen las tensiones del cañón de electrones.

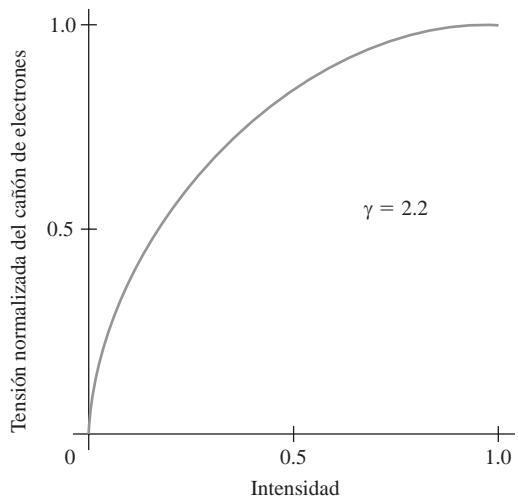
Podemos combinar la corrección gamma con el mapeado logarítmico de intensidad para generar una tabla de sustitución. Si  $I$  es un valor de intensidad de entrada producido por un modelo de iluminación, primero localizamos la intensidad más próxima  $I_k$  en una tabla de valores creada con la Ecuación 10.34 o la Ecuación 10.37. Alternativamente, podemos determinar el número del nivel correspondiente a este valor de intensidad aplicando la fórmula:

$$k = \text{round} \left[ \log_r \left( \frac{I}{I_0} \right) \right] \quad (10.42)$$

y luego podemos calcular el valor de intensidad para este nivel utilizando la Ecuación 10.37. Una vez dispongamos del valor de intensidad  $I_k$ , podemos calcular la tensión del cañón de electrones mediante la ecuación:



**FIGURA 10.34.** Una curva típica de respuesta de un monitor, mostrando la variación en la intensidad (o brillo) visualizada en función de la tensión normalizada del cañón de electrones.



**FIGURA 10.35.** Una curva de consulta de vídeo para corrección de las intensidades que mapea un valor de intensidad normalizado a una tensión normalizada del cañón de electrones, utilizando una colección gamma con  $\gamma = 2.2$ .

$$V_k = \left( \frac{I_k}{a} \right)^{1/\gamma} \quad (10.43)$$

Los valores  $V_k$  pueden entonces almacenarse en las tablas de sustitución, almacenando los valores de  $k$  en las posiciones de píxel del búfer de imagen. Si un sistema concreto no dispone de tabla de sustitución, pueden almacenarse directamente en el búfer de imagen los valores calculados de  $V_k$ . La transformación combinada a una escala logarítmica de intensidad, seguida del cálculo de  $V_k$  utilizando la Ecuación 10.43 se denomina también en ocasiones corrección gamma.

Si los amplificadores de vídeo de un monitor están diseñados para convertir los valores de intensidad lineales en tensiones del cañón de electrones, podemos combinar los dos procesos de conversión de la intensidad. En este caso, la corrección gamma estará integrada en el hardware y los valores logarítmicos  $I_k$  deben precalcularse y almacenarse en el búfer de imagen (o en la tabla de colores).

## Visualización de imágenes de plano continuo

Los sistemas infográficos de alta calidad proporcionan generalmente 256 niveles de intensidad para cada componente de color, pero en muchas aplicaciones pueden obtenerse imágenes aceptables con un menor número de niveles. Un sistema de cuatro niveles proporciona unas capacidades de sombreado mínimas para las imágenes de tono continuo, mientras que pueden generarse imágenes fotorrealistas en sistemas capaces de proporcionar entre 32 y 256 niveles de intensidad por píxel.

La Figura 10.36 muestra una fotografía en tono continuo impresa con diversos niveles de intensidad. Cuando se utiliza un pequeño número de niveles de intensidad para reproducir una imagen de tono continuo, los bordes entre las diferentes regiones de intensidad (denominados *contornos*) son claramente visibles. En la reproducción de dos niveles, las características faciales de la mujer que aparece en la fotografía apenas son identificables. Utilizando cuatro niveles de intensidad, comenzamos a identificar los patrones de sombreado originales, pero los efectos de contorneado son excesivos. Con 8 niveles de intensidad, los efectos de contorneado siguen siendo obvios, pero comenzamos a disponer de una mejor indicación del sombreado original.



**FIGURA 10.36.** Una fotografía de tono continuo (a) impresa con 2 niveles de intensidad (b), cuatro niveles de intensidad (c) y 8 niveles de intensidad (d).

Para 16 o más niveles de intensidad, los efectos de contorneado disminuyen y las reproducciones son bastante similares al original. Las reproducciones de imágenes de plano continuo utilizando más de 32 niveles de intensidad sólo muestran diferencias muy sutiles con respecto al original.

## 10.9 PATRONES DE SEMITONO Y TÉCNICAS DE ALEATORIZACIÓN

Con un sistema que tenga muy pocos niveles de intensidad disponibles, podemos crear un incremento aparente en el número de niveles visualizados incorporando múltiples posiciones de píxel en la visualización de cada valor de intensidad de una escena. Cuando contemplamos una pequeña región compuesta de varias posiciones de píxel, nuestros ojos tienden a integrar o promediar los detalles menores, obteniendo una intensidad global. Los monitores e impresoras monocromos, en concreto, pueden aprovechar este efecto visual para generar imágenes que parecen mostradas con múltiples valores de intensidad.

Las fotografías de tono continuo se reproducen en periódicos, revistas y libros mediante un proceso de impresión denominado **impresión por semitonos** y las imágenes reproducidas se denominan **semitonos**. Para



**FIGURA 10.37.** Una sección ampliada de una fotografía reproducida mediante un método de impresión por semitonos, donde se muestra cómo se representan los tonos mediante «puntos» de tamaño variable.

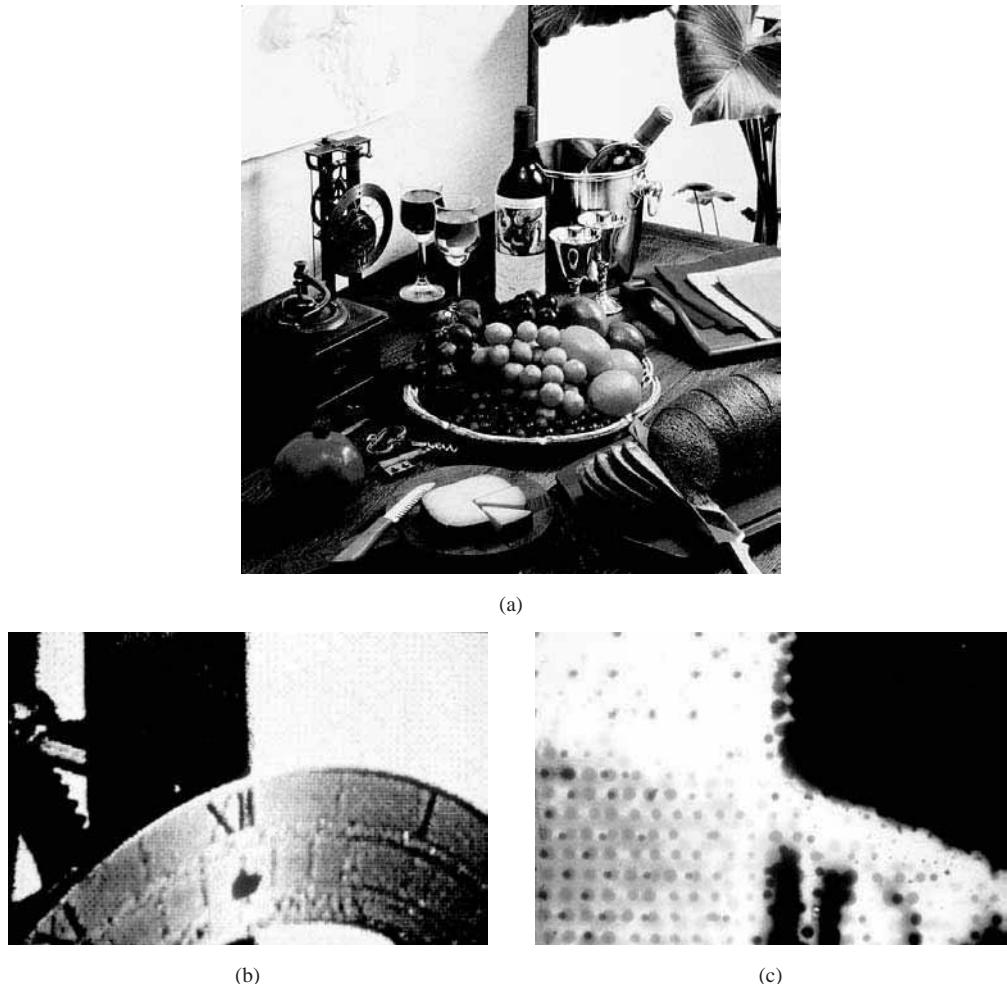
una fotografía en blanco y negro, cada área de intensidad constante se reproduce mediante un conjunto de pequeños círculos negros sobre fondo blanco. El diámetro de cada círculo es proporcional al nivel de oscuridad requerido para dicha región de intensidad. Las regiones más oscuras se imprimen con círculos de mayor tamaño, mientras que las regiones más claras se imprimen con círculos más pequeños (más espacio en blanco). La Figura 10.37 muestra una sección ampliada de una reproducción de semitono en escala de grises. Los semitonos en color se imprimen utilizando pequeños puntos circulares de diversos tamaños y colores, como se muestra en la Figura 10.38. Los semitonos en los libros y revistas se imprimen en papel de alta calidad utilizando aproximadamente entre 60 y 80 círculos de diámetro variable por centímetro. Los periódicos utilizan un papel de menor calidad y menor resolución (entre 25 y 30 puntos por centímetro).

### Aproximaciones de semitonos

En infografía, las reproducciones de semitonos se simulan utilizando regiones de píxeles rectangulares que se denominan **patrones de aproximación de semitonos** o simplemente **patrones de píxeles**. El número de niveles de intensidad que podemos mostrar con este método dependerá de cuántos píxeles incluyamos en la cuadrícula rectangular y de cuántos niveles pueda visualizar el sistema. Con  $n$  por  $n$  píxeles por cada cuadrícula en un sistema monocromo, podemos representar  $n^2 + 1$  niveles de intensidad. La Figura 10.39 muestra una forma de definir los patrones de píxel para representar cinco niveles de intensidad que podrían utilizarse en un sistema monocromo. En el patrón 0, todos los píxeles están desactivados; en el patrón 1, hay un píxel activado; y en el patrón 4, los cuatro píxeles están activados. Un valor de intensidad  $I$  en una escena se mapea a un patrón concreto de acuerdo con el rango enumerado bajo cada una de las cuadrículas mostradas en las figuras. El patrón 0 se utiliza para  $0.0 \leq I < 0.2$ , el patrón 1 para  $0.2 \leq I < 0.4$  y el patrón 4 para  $0.8 \leq I \leq 1.0$ .

Con cuadrículas de 3 por 3 píxeles en un sistema monocromo, podemos mostrar diez niveles de intensidad. En la Figura 10.40 se muestra una forma de definir los diez patrones de píxel para estos niveles. Las posiciones de píxel se seleccionan en cada nivel de modo que los patrones aproximen los tamaños crecientes de los círculos utilizados en las reproducciones de semitono. Es decir, las posiciones de los píxeles «activados» están cerca del centro de la cuadrícula para los niveles de intensidad inferiores y se expanden hacia afuera a medida que se incrementa el nivel de intensidad.

Para cualquier tamaño de cuadrícula de píxeles, podemos representar los patrones de píxeles para las diversas intensidades posibles mediante una máscara (matriz) de posiciones de píxel. Como ejemplo, podríamos usar la siguiente máscara para generar los nueve patrones de cuadrícula 3 por 3 para niveles de intensidad por encima de 0 que se muestran en la Figura 10.40.



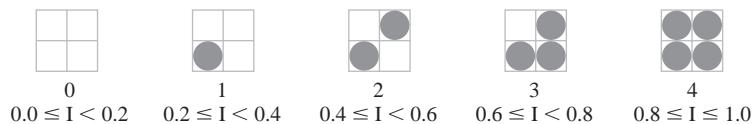
**FIGURA 10.38.** Patrones de puntos para semitonos en color. La parte superior de la esfera del reloj en el semitono en color (a) se muestra agrandada según un factor 10 en (b) y según un factor 50 en (c). (*Cortesía de IRIS Graphics, Inc., Bedford, Massachusetts.*)

$$\begin{bmatrix} 8 & 3 & 7 \\ 5 & 1 & 2 \\ 4 & 9 & 6 \end{bmatrix} \quad (10.44)$$

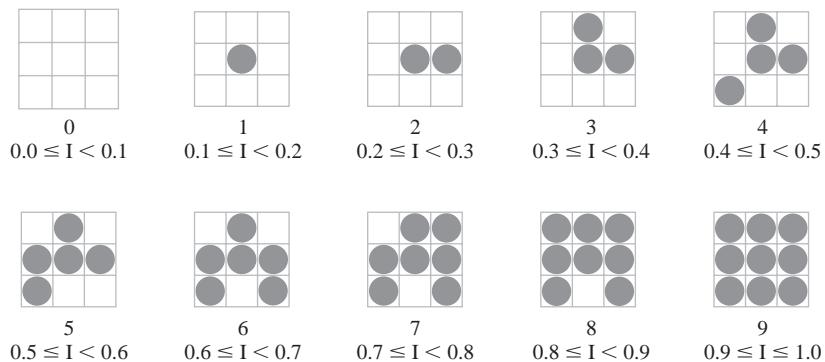
Para mostrar una intensidad concreta cuyo número de nivel sea  $k$ , activaremos todos los píxeles cuyo número de posición sea igual o inferior a  $k$ .

Aunque la utilización de patrones de píxeles  $n$  por  $n$  incrementa el número de intensidades que pueden representarse, la resolución del área de visualización se reduce según el factor  $1/n$  en las direcciones  $x$  e  $y$ . Utilizando patrones de cuadrícula de 2 por 2 en un área de pantalla de 512 por 512, por ejemplo, se reduce la resolución a 256 por 256 posiciones de intensidad. Y con patrones 3 por 3, reducimos la resolución del área de tamaño 512 por 512 a 128 por 128.

Otro problema con las cuadrículas de píxeles es que, a medida que se incrementa el tamaño de la cuadrícula, los patrones de subcuadrícula comienzan a hacerse perceptibles. El tamaño de cuadrícula que podrá utilizarse sin distorsionar las variaciones de intensidad dependerá del tamaño de cada píxel visualizado. Por



**FIGURA 10.39.** Un conjunto de patrones de cuadrícula de 2 por 2 píxeles que puede utilizarse para mostrar cinco niveles de intensidad en un sistema monocromo, indicándose los píxeles «activados» como círculos rojos. Los valores de intensidad asignados a cada uno de los patrones de cuadrícula se indican debajo de las matrices de píxeles.



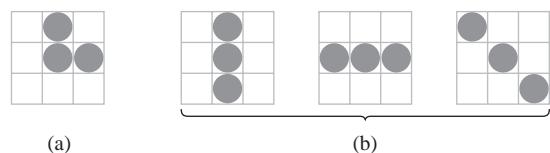
**FIGURA 10.40.** Un conjunto de patrones de cuadrícula de 3 por 3 píxeles que puede usarse para mostrar diez niveles de intensidad en un sistema monocromo, indicándose los píxeles «activados» como círculos rojos. Los valores de intensidad que se asignan a cada uno de los patrones de la cuadrícula se indican debajo de las matrices de píxeles.

tanto, para los sistemas de menor resolución (menos píxeles por centímetro), nos tendremos que dar por satisfechos con un número menor de niveles de intensidad. Por otra parte, las imágenes de alta calidad requieren al menos 64 niveles de intensidad, lo que quiere decir que necesitamos cuadrículas de 8 por 8 píxeles. Y para conseguir una resolución equivalente a la de los semitonos de los libros y las revistas, debemos mostrar 60 puntos por centímetro. Por tanto, será necesario visualizar  $60 \times 8 = 480$  puntos por centímetro. Algunos dispositivos, como por ejemplo las filmadoras de alta calidad, son capaces de obtener esta resolución.

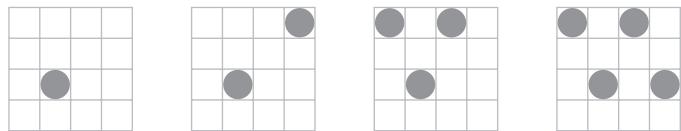
Los patrones de cuadrícula de píxeles para las aproximaciones de semitono deben también construirse de forma que se minimicen el contorneado y otros efectos visuales no presentes en la escena original. Podemos minimizar el contorneado haciendo que cada patrón de cuadrícula sucesivo represente una evolución con respecto al patrón anterior, es decir, formamos el patrón de nivel  $k$  añadiendo una posición «activada» al patrón de cuadrícula utilizado para el nivel  $k - 1$ . Así, si una posición de píxel está activada para un determinado nivel de intensidad, también lo estará para todos los niveles superiores (Figuras 10.39 y 10.40). Podemos minimizar la introducción de otros efectos visuales evitando la existencia de patrones simétricos. Con una cuadrícula de 3 por 3 píxeles, por ejemplo, el tercer nivel de intensidad por encima de cero se representará mejor mediante el patrón de la Figura 10.41(a) que mediante cualquiera de las disposiciones simétricas de la Figura 10.41(b). Los patrones simétricos de esta figura producirían trazos verticales, horizontales o diagonales en las áreas de gran tamaño que estuvieran sombreadas con el nivel de intensidad 3. En las tareas de impresión en dispositivos tales como filmadoras y algunas impresoras, los píxeles aislados no se reproducen de manera adecuada. Por tanto conviene evitar los patrones de cuadrícula con un único píxel «activado» o con píxeles «activados» aislados, como en la Figura 10.42.

Los métodos de aproximación de semitonos también pueden aplicarse para incrementar el número de niveles de intensidad del sistema que sean capaces de mostrar más de dos niveles de intensidad por píxel. Por ejemplo, en un sistema de escala de grises que pueda visualizar cuatro valores de intensidad por píxel, podemos utilizar cuadrículas de 2 por 2 píxeles para representar 13 diferentes niveles de intensidad. La Figura

**FIGURA 10.41.** Para una cuadrícula de 3 por 3 píxeles, el patrón de (a) es mejor que cualquiera de los patrones simétricos de (b) para representar el tercer nivel de intensidad por encima de 0.

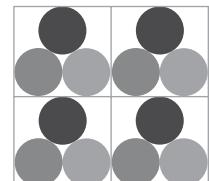


**FIGURA 10.42.** Patrones de cuadrícula de semitonos con píxeles aislados que no pueden reproducirse de manera adecuada en algunos dispositivos de impresión.



<table border="1"><tr><td>0</td><td>0</td></tr><tr><td>0</td><td>0</td></tr></table>	0	0	0	0	<table border="1"><tr><td>0</td><td>1</td></tr><tr><td>0</td><td>0</td></tr></table>	0	1	0	0	<table border="1"><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	0	1	1	0	<table border="1"><tr><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	1	1	1	0	<table border="1"><tr><td>1</td><td>1</td></tr><tr><td>1</td><td>1</td></tr></table>	1	1	1	1	<table border="1"><tr><td>1</td><td>2</td></tr><tr><td>1</td><td>1</td></tr></table>	1	2	1	1					
0	0																																	
0	0																																	
0	1																																	
0	0																																	
0	1																																	
1	0																																	
1	1																																	
1	0																																	
1	1																																	
1	1																																	
1	2																																	
1	1																																	
0	1	2	3	4	5																													
<table border="1"><tr><td>1</td><td>2</td></tr><tr><td>2</td><td>1</td></tr></table>	1	2	2	1	<table border="1"><tr><td>2</td><td>2</td></tr><tr><td>2</td><td>1</td></tr></table>	2	2	2	1	<table border="1"><tr><td>2</td><td>2</td></tr><tr><td>2</td><td>2</td></tr></table>	2	2	2	2	<table border="1"><tr><td>2</td><td>3</td></tr><tr><td>2</td><td>2</td></tr></table>	2	3	2	2	<table border="1"><tr><td>2</td><td>3</td></tr><tr><td>3</td><td>2</td></tr></table>	2	3	3	2	<table border="1"><tr><td>3</td><td>3</td></tr><tr><td>3</td><td>2</td></tr></table>	3	3	3	2	<table border="1"><tr><td>3</td><td>3</td></tr><tr><td>3</td><td>3</td></tr></table>	3	3	3	3
1	2																																	
2	1																																	
2	2																																	
2	1																																	
2	2																																	
2	2																																	
2	3																																	
2	2																																	
2	3																																	
3	2																																	
3	3																																	
3	2																																	
3	3																																	
3	3																																	
6	7	8	9	10	11	12																												

**FIGURA 10.43.** Representaciones de las intensidades 0 a 12 obtenidas mediante patrones de semitonos utilizando cuadrículas de 2 por 2 píxeles en un sistema de cuatro niveles, con los niveles de intensidad de píxel etiquetados de 0 a 3.



**FIGURA 10.44.** Un patrón de cuadrícula de 2 por 2 píxeles para la visualización de colores RGB.

10.43 ilustra una forma de definir los 13 patrones de cuadrícula de píxeles, pudiendo asignarse a cada píxel los niveles de intensidad 0, 1, 2 y 3.

De forma similar, podemos utilizar patrones de cuadrícula de píxeles para incrementar el número de niveles de intensidad que pueden representarse en un sistema en color. Por ejemplo, un sistema RGB con tres bits por píxel utilizará un bit por píxel para cada cañón de color. Así, cada píxel se visualiza mediante tres puntos de fósforo, de modo que puede asignarse al píxel uno cualquiera de ocho colores diferentes (incluidos el blanco y el negro). Pero con patrones de cuadrícula de 2 por 2 píxeles, son 12 los puntos de fósforo que podemos utilizar para representar un color, como se muestra en la Figura 10.44. El cañón de electrones rojo permite activar cualquier combinación de los cuatro puntos rojos del patrón de cuadrícula y esto proporciona cinco posibles configuraciones para el color rojo del patrón. Lo mismo puede decirse de los cañones verde y azul, lo que nos da un total de 125 combinaciones de color diferentes que pueden representarse con nuestros patrones de cuadrícula 2 por 2.

## Técnicas de aleatorización

El término **aleatorización** (*dithering*) se utiliza en varios contextos. Fundamentalmente, hace referencia a técnicas empleadas para aproximar semitonos sin reducir la resolución, a diferencia de lo que sucede con los

patrones de cuadrículas de píxeles. Pero el término aleatorización se emplea también en ocasiones como sinónimo para cualquier esquema de aproximación de semitonos, e incluso para referirse a las aproximaciones de semitonos en color.

Los valores aleatorios sumados a las intensidades de los píxeles con el fin de descomponer los contornos se suelen denominar **ruido de aleatorización**. Se han utilizado diversos algoritmos para generar las distribuciones aleatorias. El efecto de todos ellos consiste en añadir ruido a la imagen completa, lo que tiende a suavizar las fronteras entre los distintos niveles de intensidad.

Un método denominado **aleatorización ordenada** genera variaciones de intensidad mediante una aplicación biyectiva de los puntos de una escena a las posiciones de píxel utilizando una **matriz de aleatorización**  $\mathbf{D}_n$  para seleccionar cada nivel de intensidad. La matriz  $\mathbf{D}_n$  contiene  $n$  por  $n$  elementos a los que se asignan valores enteros positivos diferentes en el rango que va de 0 a  $n^2 - 1$ . Por ejemplo, podemos generar cuatro niveles de intensidad mediante:

$$\mathbf{D}_2 = \begin{bmatrix} 3 & 1 \\ 0 & 2 \end{bmatrix} \quad (10.45)$$

y podemos generar nueve niveles de intensidad mediante:

$$\mathbf{D}_3 = \begin{bmatrix} 7 & 2 & 6 \\ 4 & 0 & 1 \\ 3 & 8 & 5 \end{bmatrix} \quad (10.46)$$

Los elementos de las matrices  $\mathbf{D}_2$  y  $\mathbf{D}_3$  están en el mismo orden que la máscara de píxeles utilizada para definir las cuadrículas de 2 por 2 y 3 por 3 píxeles, respectivamente. En un sistema monocromo, determinamos los valores de intensidad que hay que visualizar comparando las intensidades de entrada con los elementos de la matriz. Primero se cambia la escala de los niveles de intensidad de entrada al rango  $0 \leq I \leq n_2$ . Si hay que aplicar la intensidad  $I$  a la posición de pantalla  $(x, y)$ , calculamos la posición de referencia (fila y columna) en la matriz de aleatorización de la forma siguiente:

$$j = (x \bmod n) + 1, \quad k = (y \bmod n) + 1 \quad (10.47)$$

Si  $I > \mathbf{D}_n(j, k)$ , activaremos el píxel situado en la posición  $(x, y)$ . En caso contrario, el píxel permanecerá desactivado. En las aplicaciones de color RGB, este procedimiento se implementa para la intensidad de cada una de las componentes individuales de color (roja, verde y azul).

Los elementos de la matriz de aleatorización se asignan según las directrices que ya hemos comentado para las cuadrículas de píxeles, es decir, con el objetivo de minimizar los efectos visuales artificiales, como el contorneado. La aleatorización ordenada produce áreas de intensidad constante idénticas a las que se generan mediante los patrones de cuadrículas de píxeles, cuando los valores de los elementos de la matriz se corresponden con los de la máscara de la cuadrícula de aproximación de semitonos. Las variaciones con respecto a las imágenes obtenidas mediante cuadrículas de píxeles se producen en la frontera de dos áreas de intensidad diferente.

Normalmente, el número de niveles de intensidad utilizados será un múltiplo de 2. Las matrices de aleatorización de orden superior,  $n \geq 4$ , se obtienen entonces a partir de las matrices de orden inferior utilizando la relación recurrente:

$$\mathbf{D}_n = \begin{bmatrix} 4\mathbf{D}_{n/2} + \mathbf{D}_2(1,1)\mathbf{U}_{n/2} & 4\mathbf{D}_{n/2} + \mathbf{D}_2(1,2)\mathbf{U}_{n/2} \\ 4\mathbf{D}_{n/2} + \mathbf{D}_2(2,1)\mathbf{U}_{n/2} & 4\mathbf{D}_{n/2} + \mathbf{D}_2(2,2)\mathbf{U}_{n/2} \end{bmatrix} \quad (10.48)$$

El parámetro  $\mathbf{U}_{n/2}$  representa la matriz «unidad» (todos los elementos iguales a 1). Como ejemplo, si se especifica  $\mathbf{D}_2$  como en la Ecuación 10.45, la relación recursiva 10.48 nos da:

$$\mathbf{D}_4 = \begin{bmatrix} 15 & 7 & 13 & 5 \\ 3 & 11 & 1 & 9 \\ 12 & 4 & 10 & 6 \\ 0 & 8 & 2 & 10 \end{bmatrix} \quad (10.49)$$

Otro método para mapear una imagen con  $m$  por  $n$  puntos a un área de visualización de  $m$  por  $n$  píxeles es la **difusión de error**. Con este método, el error entre un valor de intensidad de entrada y el nivel de intensidad seleccionado para una determinada posición de píxel se dispersa, o difunde, a las posiciones de pixel situadas a la derecha y por debajo de la posición de píxel actual. Comenzando con una matriz  $\mathbf{M}$  de valores de intensidad obtenida escaneando una fotografía, podemos tratar de construir una matriz  $\mathbf{I}$  de valores de intensidad de píxel para un área de la pantalla. En este caso, podemos hacerlo leyendo las filas de  $\mathbf{M}$  de izquierda a derecha, comenzando por la fila superior, y determinando el nivel de intensidad de píxel más próximo disponible para cada elemento de  $\mathbf{M}$ . Entonces, el error entre el valor almacenado en la matriz  $\mathbf{M}$  y el nivel de intensidad mostrado en cada posición de píxel se distribuye a los elementos vecinos utilizando el siguiente algoritmo simplificado:

```

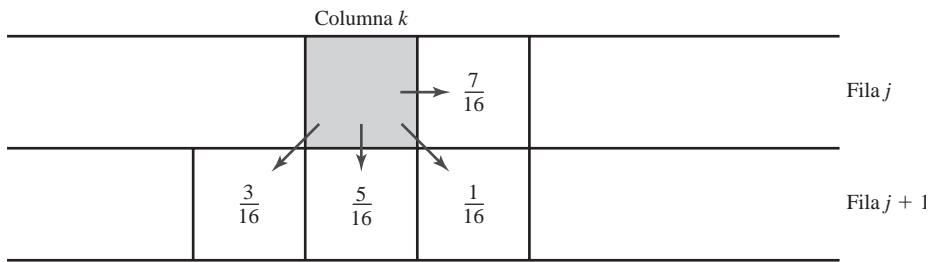
for (j = 0; j < m; j++) {
    for (k = 0; k < n; k++) {
        /* Determinar el valor de intensidad del sistema disponible
         * que esté más próximo al valor de M [j][k] y
         * asignar este valor a I [j][k].
        */
        error = M [j][k] - I [j][k];
        I [j][k+1] = M [j][k+1] + alpha * error;
        I [j+1][k-1] = M [j+1][k-1] + beta * error;
        I [j+1][k] = M [j+1][k] + gamma * error;
        I [j+1][k+1] = M [j+1][k+1] + delta * error;
    }
}

```

Una vez asignados los niveles de intensidad a los elementos de la matriz  $\mathbf{I}$ , mapeamos la matriz sobre algún área de un dispositivo de salida, como por ejemplo una impresora o un monitor de vídeo. Por supuesto, no podemos dispersar el error más allá de la última columna de la matriz ( $k = n$ ) o por debajo de la última fila de la matriz ( $j = m$ ) y para un sistema monocromo los valores de intensidad del sistemas son simplemente 0 y 1. Los parámetros para distribuir el error pueden elegirse de modo que se satisfaga la relación:

$$\alpha + \beta + \gamma + \delta \leq 1 \quad (10.50)$$

Una posible elección para los parámetros de difusión de errores que produce muy buenos resultados es  $(\alpha, \beta, \gamma, \delta) = (\frac{7}{16}, \frac{3}{16}, \frac{5}{16}, \frac{1}{16})$ . La Figura 10.45 ilustra la distribución de errores utilizando estos valores de parámetros. La difusión de errores produce en ocasiones «fantasmas» en las imágenes al repetir ciertas partes de la imagen (ecos), particularmente con características faciales tales como la línea del pelo o el contorno de la nariz. Estos fantasmas pueden reducirse a menudo en dichos casos seleccionando valores de los parámetros de difusión de errores cuya suma dé un valor inferior a 1 y cambiando la escala de los valores de la matriz después de dispersar los errores. Una forma de cambiar la escala consiste en multiplicar todos los elementos de la matriz por 0.8 y luego sumar 0.1. Otro método para mejorar la calidad de la imagen consiste en alternar la lectura de las filas de la matriz, leyendo una de derecha a izquierda y la siguiente de izquierda a derecha.



**FIGURA 10.45.** Fracción del error de intensidad que puede distribuirse a las posiciones de píxel adyacentes utilizando un esquema de difusión de errores.

34	48	40	32	29	15	23	31
42	58	56	53	21	5	7	10
50	62	61	45	13	1	2	18
38	46	54	37	25	17	9	26
28	14	22	30	35	49	41	33
20	4	6	11	43	59	57	52
12	0	3	19	51	63	60	44
24	16	8	27	39	47	55	36

**FIGURA 10.46.** Un posible esquema de distribución para dividir la matriz de intensidad en 64 clases de difusión de puntos, numeradas de 0 a 63.

Una variación del método de difusión de errores es la **difusión de puntos**. Con este método, la matriz  $m$  por  $n$  de valores de intensidad se divide en 64 clases, numeradas de 0 a 63, como se muestra en la Figura 10.46. El error entre un error de matriz y la intensidad visualizada se distribuye entonces únicamente a aquellos elementos vecinos de la matriz que tengan un número de clase mayor. La distribución de los 64 números de clase está realizada con el objetivo de minimizar el número de elementos que estén completamente rodeados por elementos con un número de clase inferior, ya que esto tendería a dirigir todos los errores de los elementos circundantes hacia esa posición.

## 10.10 MÉTODOS DE REPRESENTACIÓN DE POLÍGONOS

Los cálculos de intensidad de un modelo de iluminación pueden aplicarse a los procedimientos de representación superficial de varias formas. Podemos utilizar un modelo de iluminación para determinar la intensidad superficial en cada posición de píxel proyectada, o bien podemos aplicar el modelo de iluminación a unos pocos puntos seleccionados y aproximar la intensidad en el resto de las posiciones de la superficie. Los paquetes gráficos realizan normalmente la representación superficial utilizando algoritmos de línea de barrido que reducen el tiempo de procesamiento tratando sólo con superficies poligonales y calculando las intensidades superficiales exclusivamente en los vértices. Después, se interpolan las intensidades de los vértices para las otras posiciones de la superficie poligonal. Se han desarrollado otros métodos de representación poligonal basados en líneas de barrido más precisos y asimismo los algoritmos de trazado de rayos permiten calcular la intensidad en cada punto proyectado de la superficie para el caso de superficies tanto planas como curvas. Vamos a considerar primero los esquemas de representación de superficies basados en líneas de barrido que se suelen aplicar al caso de los polígonos. Después, en la Sección 10.11, examinaremos los métodos que pueden utilizarse en los procedimientos de trazado de rayos.

## Representación superficial con intensidad constante

El método más simple para representar una superficie poligonal consiste en asignar el mismo color a todas las posiciones proyectadas de superficie. En este caso, utilizamos el modelo de iluminación para determinar la intensidad de las tres componentes de color RGB en una única posición de la superficie, como por ejemplo en un vértice o en el centroide del polígono. Esta técnica, denominada **representación superficial de intensidad constante** o **representación superficial plana**, proporciona un método rápido y simple para mostrar las caras poligonales de un objeto y que puede ser útil para generar rápidamente una imagen que nos indique la apariencia general de una superficie curva, como en la Figura 10.50(b). La representación plana también resulta útil durante las tareas de diseño o en otras aplicaciones en las que queramos identificar rápidamente las caras poligonales individuales utilizadas para modelar una superficie curva.

En general, la representación plana de una superficie poligonal proporciona una imagen precisa de la superficie si se cumplen todas las siguientes suposiciones:

- El polígono es una cara de un poliedro y no una malla de aproximación de una superficie curva.
- Todas las fuentes luminosas que iluminan el polígono están lo suficientemente lejos de la superficie, de modo que  $\mathbf{N} \cdot \mathbf{L}$  y la función de atenuación son constantes para toda el área del polígono.
- La posición de visualización está lo suficientemente lejos del polígono, de modo que  $\mathbf{V} \cdot \mathbf{R}$  es constante para toda el área del polígono.

Incluso si alguna de estas condiciones no es cierta, podemos seguir aproximando razonablemente bien los efectos de iluminación de la superficie utilizando mecanismos de representación superficial de intensidad constante si las caras poligonales del objeto son pequeñas.

## Representación de superficies por el método de Gouraud

Este esquema, desarrollado por Henri Gouraud y denominado **representación de superficies por el método de Gouraud** o **representación de superficies por interpolación de la intensidad**, interpola linealmente los valores de intensidad de los vértices en las caras poligonales de un objeto iluminado. Desarrollado para representar una superficie curva que esté aproximada mediante una malla poligonal, el método de Gouraud efectúa una transición suave entre los valores de intensidad de una cara poligonal y los valores de las caras poligonales adyacentes, a lo largo de las aristas comunes. Esta interpolación de intensidades en el área del polígono elimina las discontinuidades de intensidad que pueden aparecer en la representación plana de superficies.

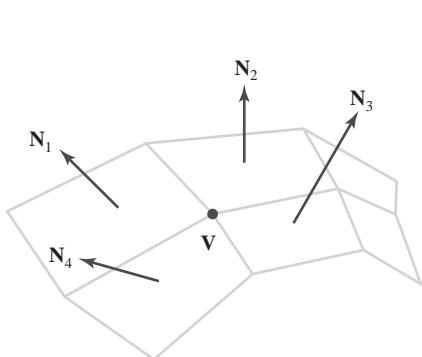
Cada sección poligonal de una superficie curva teselada se procesa mediante el método de representación de superficies de Gouraud utilizando los siguientes procedimientos:

- (1) Se determina el vector unitario normal promedio en cada vértice del polígono.
- (2) Se aplica un modelo de iluminación en cada vértice del polígono para obtener la intensidad luminosa en dicha posición.
- (3) Se interpolan linealmente las intensidades de los vértices para el área proyecta del polígono.

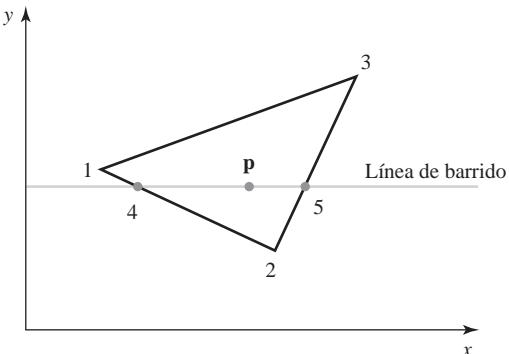
En cada vértice del polígono, obtenemos un vector normal calculando el promedio entre los vectores normales de todos los polígonos de la malla de la superficie que comparten dicho vértice, como se ilustra en la Figura 10.47. Así, para cualquier posición de vértice  $\mathbf{V}$ , obtenemos el vector unitario normal del vértice mediante la fórmula:

$$\mathbf{N}_V = \frac{\sum_{k=1}^n \mathbf{N}_k}{\left| \sum_{k=1}^n \mathbf{N}_k \right|} \quad (10.51)$$

Una vez obtenido el vector normal en el vértice, invocamos el modelo de iluminación para calcular la intensidad superficial en dicho punto.



**FIGURA 10.47.** El vector normal en el vértice  $V$  se calcula como promedio de las normales de superficie para cada uno de los polígonos que comparten dicho vértice.



**FIGURA 10.48.** Para representar la superficie de Gouraud, la intensidad en el punto 4 se interpola linealmente a partir de las intensidades en los vértices 1 y 2. La intensidad en el punto 5 se interpola linealmente a partir de las intensidades en los vértices 2 y 3. En un punto interior  $p$  se asigna un valor de intensidad que se interpole linealmente a partir de las intensidades en las posiciones 4 y 5.

Después de haber calculado todas las intensidades de vértice para una cara poligonal, podemos interpolar los valores de los vértices para obtener las intensidades en las posiciones situadas a lo largo de las líneas de barrido que intersecten con el área proyectada del polígono, como se ilustra en la Figura 10.48. Para cada línea de barrido, la intensidad de la intersección de la línea de barrido con una arista del polígono se interpola linealmente a partir de las intensidades de los extremos de dicha arista. Para el ejemplo de la Figura 10.48, la arista del polígono cuyos extremos están en las posiciones 1 y 2 es intersectada por la línea de barrido en el punto 4. Un método rápido para obtener la intensidad en el punto 4 consiste en interpolar entre los valores correspondientes a los vértices 1 y 2 utilizando únicamente el desplazamiento vertical de la línea de barrido:

$$I_4 = \frac{y_4 - y_2}{y_1 - y_2} I_1 + \frac{y_1 - y_4}{y_1 - y_2} I_2 \quad (10.52)$$

En esta expresión, el símbolo  $I$  representa la intensidad de una de las componentes de color RGB. De forma similar, la intensidad en la intersección derecha de esta línea de barrido (punto 5) se interpola a partir de los valores de intensidad en los vértices 2 y 3. Partiendo de estas dos intensidades de los extremos, podemos interpolar linealmente para obtener las intensidades de píxel en las distintas posiciones que componen la línea de barrido. La intensidad de una de las componentes de color RGB en el punto  $p$  de la Figura 10.48, por ejemplo, se calcula a partir de las intensidades de los puntos 4 y 5 como:

$$I_p = \frac{x_5 - x_p}{x_5 - x_4} I_4 + \frac{x_p - x_4}{x_5 - x_4} I_5 \quad (10.53)$$

En la implementación de los mecanismos de representación de Gouraud, podemos realizar los cálculos de intensidad representado por las Ecuaciones 10.52 y 10.53 de manera eficiente utilizando métodos incrementales. Comenzando por una línea de barrido que intersecte uno de los vértices del polígono, podemos obtener incrementalmente los valores de intensidad para otras líneas de barrido que intersecten una arista conectada a dicho vértice. Suponiendo que las caras poligonales sean convexas, cada línea de barrido que cruce el polígono tendrá dos intersecciones con las aristas, como por ejemplo los puntos 4 y 5 de la Figura 10.48. Una vez obtenidas las intensidades en las dos intersecciones de la línea de barrido con las aristas, aplicamos los procedimientos incrementales para obtener las intensidades de píxel en la línea de barrido.

Como ejemplo del cálculo incremental de las intensidades, vamos a considerar las líneas de barrido  $y$  e  $y - 1$  de la Figura 10.49, que intersectan la arista izquierda de un polígono. Si la línea de barrido  $y$  es la línea de barrido situada inmediatamente por debajo del vértice de  $y_1$  con intensidad  $I_1$ , es decir  $y = y_1 - 1$ , podemos calcular la intensidad  $I$  en la línea de barrido  $y$  a partir de la Ecuación 10.52, de la manera siguiente:

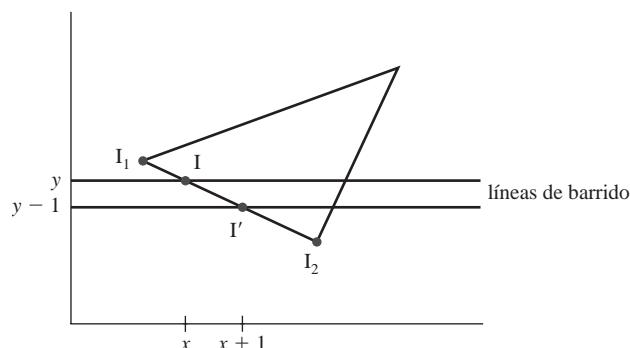
$$I = I_1 + \frac{I_2 - I_1}{y_1 - y_2} \quad (10.54)$$

Si continuamos bajando por la arista del polígono, la intensidad en la arista para la siguiente línea de barrido,  $y - 1$ , será:

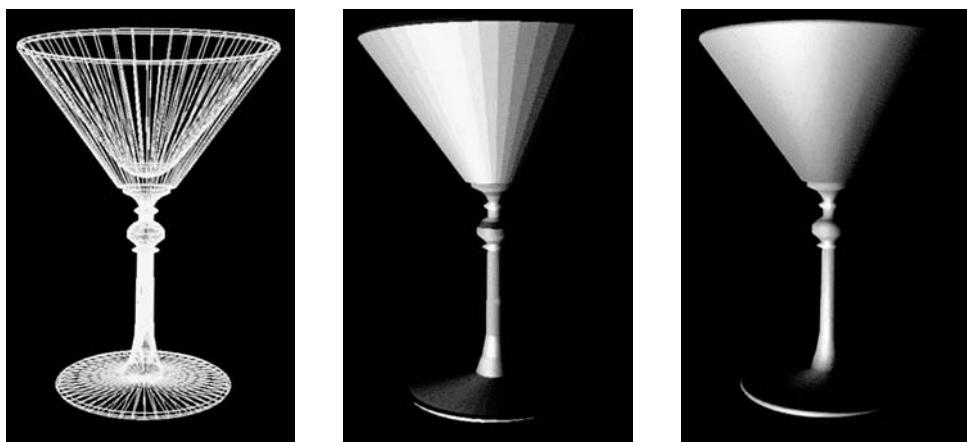
$$I' = I + \frac{I_2 - I_1}{y_1 - y_2} \quad (10.55)$$

Así, cada valor sucesivo de intensidad al descender por la arista se calcula simplemente sumando el término constante  $(I_2 - I_1)/(y_1 - y_2)$  al valor de intensidad anterior. Se utilizan cálculos incrementales similares para obtener las intensidades en las sucesivas posiciones de píxel horizontales dentro de cada línea de barrido.

Los mecanismos de representación superficial de Gouraud pueden combinarse con un algoritmo de detección de superficies ocultas con el fin de llenar los polígonos visibles para cada línea de barrido. En la Figura 10.50(c) se muestra un ejemplo de objeto tridimensional representado por el método de Gouraud.



**FIGURA 10.49.** Interpolación incremental de los valores de intensidad a lo largo de una arista de un polígono para líneas de barrido sucesivas.



**FIGURA 10.50.** Una aproximación de un objeto mediante malla poligonal (a) se muestra utilizando representación superficial plana en (b) y representación superficial de Gouraud en (c).

Este método de interpolación de la intensidad elimina las discontinuidades asociadas con los mecanismos de representación plana, pero presenta algunas otras deficiencias. Los resaltes en la superficie se muestran en ocasiones con formas anómalas y la interpolación lineal de la intensidad puede provocar la aparición de trazos brillantes u oscuros, denominados **bandas de Mach**, sobre la superficie. Estos efectos pueden reducirse dividiendo la superficie en un número mayor de caras poligonales o utilizando cálculos de intensidad más precisos.

### Representación superficial de Phong

Un método más preciso de interpolación para la representación de un malla poligonal fue el que desarrolló posteriormente Phong Bui Tuong. Esta técnica, denominada **representación superficial de Phong** o **representación por interpolación de los vectores normales**, realiza una interpolación de los vectores normales en lugar de interpolar los valores de intensidad. El resultado es un cálculo más preciso de los valores de intensidad, una imagen más realista de los resaltes de la superficie y una enorme reducción en los efectos de bandas de Mach. Sin embargo, el método de Phong requiere realizar más cálculos que el método de Gouraud.

Cada sección poligonal de una superficie curva teselada se procesa mediante el método de representación superficial de Phong utilizando los siguientes procedimientos:

- (1) Se determina el vector unitario normal promedio en cada vértice del polígono.
- (2) Se interpolan linealmente las normales a los vértices para el área proyectada del polígono.
- (3) Se aplica un modelo de iluminación en las distintas posiciones de las líneas de barrido para calcular las intensidades de los píxeles utilizando los vectores normales interpolados.

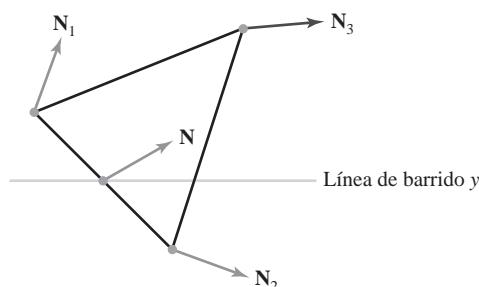
Los procedimientos de interpolación para vectores normales en el método de Phong son los mismos que para los valores de intensidad en el método de Gouraud. El vector normal  $N$  de la Figura 10.51 se interpola verticalmente a partir de los vectores normales en los vértices 1 y 2 mediante la fórmula:

$$\mathbf{N} = \frac{y - y_2}{y_1 - y_2} \mathbf{N}_1 + \frac{y_1 - y}{y_1 - y_2} \mathbf{N}_2 \quad (10.56)$$

Después, aplicamos los mismos métodos incrementales para obtener los vectores normales en las sucesivas líneas de barrido y en las sucesivas posiciones de píxel dentro de cada línea de barrido. La diferencia entre las dos técnicas de representación superficial es que ahora deberemos aplicar el modelo de iluminación para cada posición de píxel proyectada dentro de las líneas de barrido, con el fin de obtener los valores de intensidad superficial.

### Representación superficial rápida de Phong

Podemos reducir el tiempo de procesamiento en el método de representación de Phong realizando ciertas aproximaciones en los cálculos del modelo de iluminación. Los algoritmos de **representación superficial rápida de Phong** llevan a cabo los cálculos de intensidad utilizando una expansión en serie de Taylor truncada y limitando las caras poligonales a parches de superficie triangulares.



**FIGURA 10.51.** Interpolación de las normales a la superficie a lo largo de una arista de un polígono.

Puesto que el método de Phong interpola los vectores normales a partir de las normales en los vértices, podemos escribir la expresión para el cálculo de la normal  $\mathbf{N}$  a la superficie en la posición  $(x, y)$  de un parche triangular como:

$$\mathbf{N} = \mathbf{Ax} + \mathbf{By} + \mathbf{C} \quad (10.57)$$

donde los vectores  $\mathbf{A}$ ,  $\mathbf{B}$  y  $\mathbf{C}$  se determinan a partir de las tres ecuaciones de los vértices:

$$\mathbf{N}_k = \mathbf{Ax}_k + \mathbf{By}_k + \mathbf{C}, \quad k = 1, 2, 3 \quad (10.58)$$

donde  $(x_k, y_k)$  denota una posición proyectada de un vértice del triángulo sobre el plano de píxeles.

Si omitimos los parámetros de reflectividad y atenuación, podemos escribir los cálculos para la reflexión difusa de la luz de una fuente luminosa en un punto de la superficie  $(x, y)$  como:

$$\begin{aligned} I_{\text{diff}}(x, y) &= \frac{\mathbf{L} \cdot \mathbf{N}}{|\mathbf{L}| |\mathbf{N}|} \\ &= \frac{\mathbf{L} \cdot (\mathbf{Ax} + \mathbf{By} + \mathbf{C})}{|\mathbf{L}| |\mathbf{Ax} + \mathbf{By} + \mathbf{C}|} \\ &= \frac{(\mathbf{L} \cdot \mathbf{A})x + (\mathbf{L} \cdot \mathbf{B})y + \mathbf{L} \cdot \mathbf{C}}{|\mathbf{L}| |\mathbf{Ax} + \mathbf{By} + \mathbf{C}|} \end{aligned} \quad (10.59)$$

Esta expresión puede escribirse en la forma:

$$I_{\text{diff}}(x, y) = \frac{ax + by + c}{[dx^2 + exy + fy^2 + gx + hy + i]^{1/2}} \quad (10.60)$$

donde se utilizan parámetros como  $a$ ,  $b$ ,  $c$  y  $d$  para representar los diversos productos escalares. Por ejemplo,

$$a = \frac{\mathbf{L} \cdot \mathbf{A}}{|\mathbf{L}|} \quad (10.61)$$

Finalmente, podemos expresar el denominador de la Ecuación 10.60 como una expansión en serie de Taylor y retener los términos de hasta segundo grado en  $x$  e  $y$ . Esto nos da:

$$I_{\text{diff}}(x, y) = T_5 x^2 + T_4 xy + T_3 y^2 + T_2 x + T_1 y + T_0 \quad (10.62)$$

donde cada  $T_k$  es función de los diversos parámetros de la Ecuación 10.60, como  $a$ ,  $b$  y  $c$ .

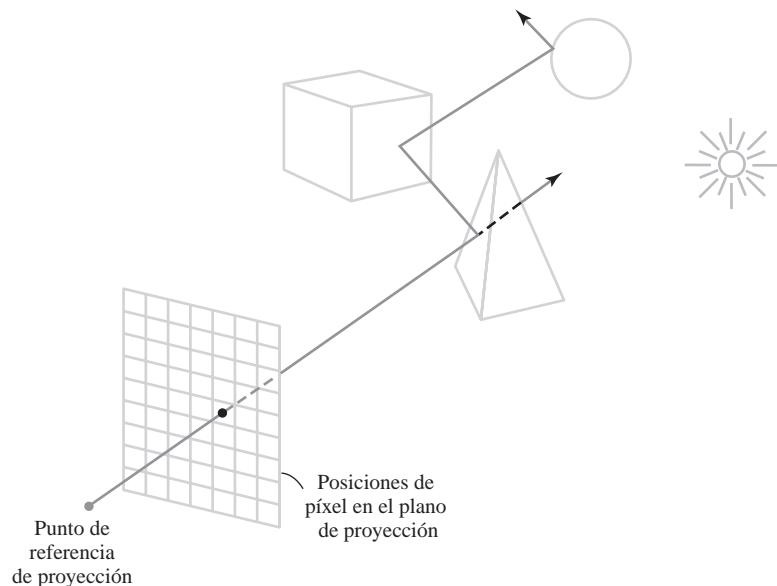
Utilizando diferencias finitas, podemos evaluar la Ecuación 10.62 con sólo dos sumas para cada posición de píxel  $(x, y)$ , una vez que hayamos evaluado los parámetros iniciales de diferencias finitas. Aunque las simplificaciones en el método rápido de Phong reducen los cálculos requeridos para la representación superficial de Phong, sigue siendo necesario un tiempo aproximadamente igual a dos veces el correspondiente a la representación superficial de Gouraud. Y el método básico de Phong, utilizando diferencias finitas, es de 6 a 7 veces más lento que el algoritmo de representación de Gouraud.

Los mecanismos de representación rápida de Phong para reflexión difusa pueden ampliarse para incluir reflexiones especulares utilizando aproximaciones similares para evaluar los términos especulares tales como  $(\mathbf{N} \cdot \mathbf{H})^{n_s}$ . Además, podemos generalizar el algoritmo para incluir una posición de visualización finita y otros polígonos distintos de los triángulos.

## 10.11 MÉTODOS DE TRAZADO DE RAYOS

---

En la Sección 8.20, hemos presentado la noción de *proyección de rayos*, que es una técnica que se utiliza en geometría constructiva de sólidos para localizar las intersecciones con las superficies a lo largo de la trayec-



**FIGURA 10.52.** Trayectos múltiples de reflexión y transmisión para un rayo trazado desde el punto de referencia de proyección a través de una posición de píxel y a través de una escena que contiene diversos objetos.



**FIGURA 10.53.** Una escena generada mediante trazado de rayos, donde se muestran los efectos globales de reflexión y transparencia. (Cortesía de Evans & Sutherland.)

toria de un rayo trazado desde una posición de píxel. También hemos hablado de los métodos de proyección de rayos en la Sección 9.10 como medio para identificar las superficies visibles en una escena. El **trazado de rayos** es la generalización del procedimiento básico de proyección de rayos. En lugar de limitarnos a localizar la superficie visible desde cada posición de píxel, lo que hacemos es continuar rebotando el rayo correspondiente al píxel a través de la escena, como se ilustra en la Figura 10.52, con el fin de recopilar las diversas contribuciones de intensidad. Esto proporciona una técnica simple y potente de representación para obtener efectos globales de reflexión y transmisión. Además, el algoritmo básico de trazado de rayos detecta las superficies visibles, identifica las áreas en sombra, permite representar efectos de transparencia, genera vistas de proyección en perspectiva y admite efectos de iluminación con múltiples fuentes luminosas. Se han desarrollado numerosas extensiones del algoritmo básico para la generación de imágenes fotorrealistas. Las imágenes de escenas generadas mediante trazado de rayos pueden ser enormemente realistas, particularmente cuando la escena contiene objetos brillantes, pero los algoritmos de trazado de rayos requieren un tiempo de cálculo considerable. En la Figura 10.53 se muestra un ejemplo de los efectos globales de reflexión y transmisión que pueden conseguirse mediante las técnicas de trazado de rayos.

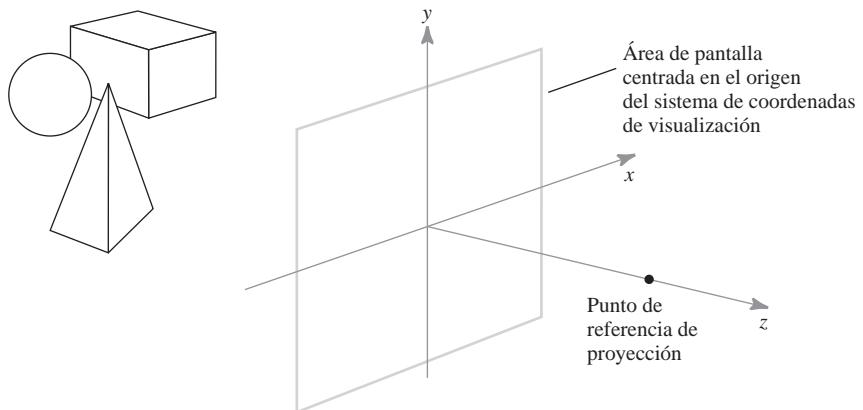
## Algoritmo básico de trazado de rayos

El sistema de coordenadas para un algoritmo de trazado de rayos suele definirse como se muestra en la Figura 10.54, con el punto de referencia de proyección en el eje  $z$  y las posiciones de píxel en el plano  $xy$ . Después describimos la geometría de una escena en este sistema de coordenadas y generamos los rayos correspondientes al píxel. Para una lista de proyección en perspectiva de la escena, cada rayo se origina en el punto de referencia de proyección (centro de proyección), pasa a través del centro de un píxel y se adentra en la escena para formar las diversas ramas del rayo, según los trayectos de reflexión y transmisión. Las contribuciones a la intensidad del píxel se acumulan entonces para las distintas superficies intersectadas. Esta técnica de representación está basada en los principios de la óptica geométrica. Los rayos luminosos que salen de las superficies de una escena emanan en todas direcciones y algunos de ellos pasan a través de las posiciones de píxel situadas en el plano de proyección. Puesto que hay un número infinito de emanaciones de rayos, determinamos las contribuciones de intensidad para un píxel concreto trazando hacia atrás el trayecto luminoso desde la posición del píxel hacia la escena. En el algoritmo básico de trazado de rayos, se genera un rayo luminoso para cada píxel.

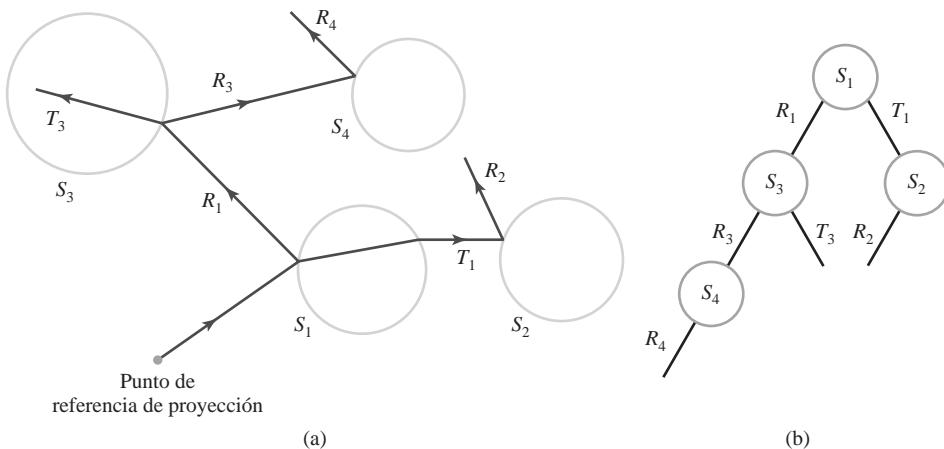
A medida que se genera el rayo de cada píxel, se procesa la lista de superficies de la escena para determinar si el rayo interseca con alguna de ellas. En caso afirmativo, calculamos la distancia desde el píxel al punto de intersección con la superficie. Una vez comprobadas todas las superficies, la distancia de intersección calculada más pequeña identificará la superficie visible para dicho píxel. Entonces reflejamos el rayo en la superficie visible según un proyecto de reflexión especular (ángulo de reflexión igual al ángulo de incidencia). Para una superficie transparente, también enviamos un rayo a través de la superficie en la dirección de refracción. Los rayos reflejado y refractado se denominan **rayos secundarios**.

Repetimos entonces los procedimientos de procesamiento de rayos para los rayos secundarios. Comprobamos si existen intersecciones con las superficies y, en caso afirmativo, se usa la superficie intersecada más próxima a lo largo de la trayectoria de un rayo secundario para producir recursivamente la siguiente generación de trayectos de reflexión y refracción. A medida que los rayos de un píxel se bifurcan a través de una escena, cada superficie recursivamente intersecada se añade a un **árbol de trazado de rayos** binario, como se muestra en la Figura 10.55. Utilizamos las ramas izquierdas del árbol para representar los trayectos de reflexión y las ramas derechas para representar los trayectos de transmisión. La profundidad máxima de los árboles de trazado de rayos puede configurarse como opción del usuario o puede determinarse según la cantidad de almacenamiento disponible. Terminamos cada trayecto del árbol binario correspondiente a un píxel si se cumple cualquiera de las siguientes condiciones:

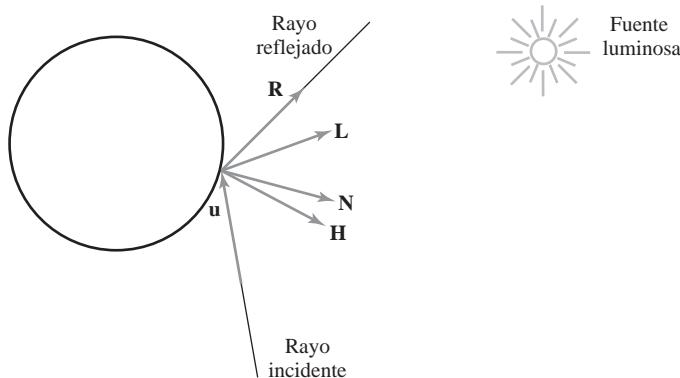
- El rayo no interseca ninguna superficie.
- El rayo intersecta a una fuente de luminosa que no es una superficie reflectante.
- El árbol ha alcanzado su profundidad máxima permitida.



**FIGURA 10.54.** Sistema de coordenadas para trazado de rayos.



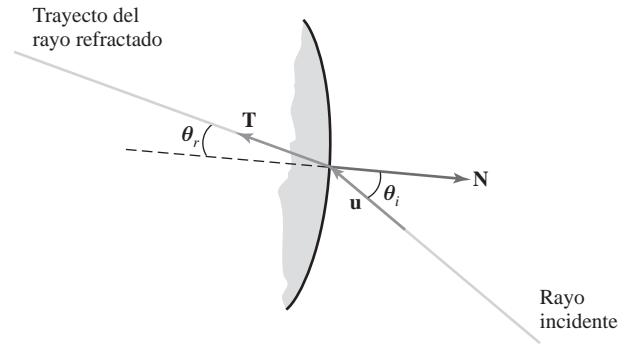
**FIGURA 10.55.** Los trayectos de reflexión y refracción para un rayo de píxel que viaja a través de una escena se muestran en (a) y el correspondiente árbol binario de trazado de rayos se indica en (b).



**FIGURA 10.56.** Vectores unitarios en la superficie de un objeto intersectado por un rayo que incide según la dirección  $\mathbf{u}$ .

En cada intersección con una superficie, invocamos el modelo básico de iluminación para determinar la contribución de dicha superficie a la intensidad. Este valor de intensidad se almacena en la posición correspondiente al nodo de la superficie dentro del árbol del píxel. A un rayo que intersecte una fuente luminosa no reflectante se le puede asignar la intensidad de la fuente, aunque las fuentes luminosas en el algoritmo básico de trazado de rayos son usualmente fuentes puntuales situadas en posiciones que caen más allá de los límites de coordenadas de la escena. La Figura 10.56 muestra una superficie intersectada por un rayo y los vectores unitarios utilizados para los cálculos de la intensidad luminosa reflejada. El vector unitario  $\mathbf{u}$  define la dirección de la trayectoria del rayo,  $\mathbf{N}$  es el vector unitario normal a la superficie,  $\mathbf{R}$  es el vector unitario de reflexión,  $\mathbf{L}$  es el vector unitario que indica la dirección de una fuente luminosa puntual y  $\mathbf{H}$  es el vector unitario medio entre  $\mathbf{L}$  y  $\mathbf{V}$ . Para los cálculos de trazado de rayos, la dirección de visualización es  $\mathbf{V} = -\mathbf{u}$ . El trayecto definido según la dirección de  $\mathbf{L}$  se denomina **rayo de sombra**. Si algún objeto intersecta el rayo de sombra entre la superficie y la fuente luminosa puntual, dicha posición de la superficie estará en sombra con respecto a la fuente. La luz ambiente en la superficie se calcula como  $k_a I_a$ , la reflexión difusa debido a la fuente es proporcional a  $k_d (\mathbf{N} \cdot \mathbf{L})$  y la componente de reflexión especular es proporcional a  $k_s (\mathbf{H} \cdot \mathbf{N})^{n_s}$ . Como se explica en la Sección 10.3, la dirección reflexión de especular para el trayecto del rayo secundario  $\mathbf{R}$  depende de la normal a la superficie y de la dirección del rayo incidente:

$$\mathbf{R} = \mathbf{u} - (2\mathbf{u} \cdot \mathbf{N})\mathbf{N} \quad (10.63)$$



**FIGURA 10.57.** Trayecto de transmisión del rayo refractado  $\mathbf{T}$  a través de un material transparente.

Para una superficie transparente, necesitamos también obtener las contribuciones de intensidad de la luz transmitida (refractada) a través del material. Podemos localizar la fuente de esta contribución trazando un rayo secundario según la dirección de transmisión  $\mathbf{T}$ , como se muestra en la Figura 10.57. El vector unitario de transmisión  $\mathbf{T}$  puede obtenerse a partir de los vectores  $\mathbf{u}$  y  $\mathbf{N}$  de la forma siguiente:

$$\mathbf{T} = \frac{\eta_i}{\eta_r} \mathbf{u} - \left( \cos \theta_r - \frac{\eta_i}{\eta_r} \cos \theta_i \right) \mathbf{N} \quad (10.64)$$

Los parámetros  $\eta_i$  y  $\eta_r$  son los índices de refracción en el material incidente y en el material refractante, respectivamente. El ángulo de refracción  $\theta_r$  puede calcularse a partir de la ley de Snell:

$$\cos \theta_r = \sqrt{1 - \left( \frac{\eta_i}{\eta_r} \right)^2 (1 - \cos^2 \theta_i)} \quad (10.65)$$

Después de haber completado el árbol binario para un píxel, se acumulan las contribuciones de intensidad, comenzando por la parte inferior (nodos terminales) del árbol. La intensidad superficial correspondiente a cada nodo del árbol se atenúa según la distancia con respecto a la superficie padre (el siguiente nodo subiendo por el árbol) y se suma a la intensidad existente en dicha superficie padre. La intensidad asignada al píxel es la suma de las intensidades atenuadas en el nodo raíz del árbol de rayos. Si el rayo principal de un píxel no intersecta ningún objeto de la escena, el árbol de trazado de rayos estará vacío y se asignará al píxel la intensidad de fondo.

### Cálculos de intersección entre rayos y superficie

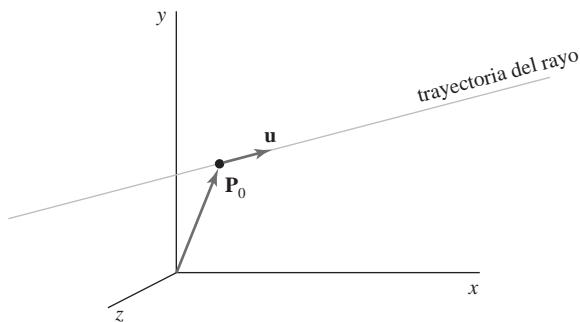
Un rayo puede describirse con una posición inicial  $\mathbf{P}_0$  y un vector de dirección unitario  $\mathbf{u}$ , como se ilustra en la Figura 10.58. Las coordenadas para cualquier punto  $\mathbf{P}$  situado a lo largo del rayo, a una distancia  $s$  de  $\mathbf{P}_0$ , se calculan entonces a partir de la **ecuación del rayo**:

$$\mathbf{P} = \mathbf{P}_0 + s\mathbf{u} \quad (10.66)$$

Inicialmente, el vector  $\mathbf{P}_0$  puede definirse en la posición  $\mathbf{P}_{pix}$  del píxel del plano de proyección, o puede seleccionarse como posición inicial el punto de referencia de proyección. El vector unitario  $\mathbf{u}$  se obtiene inicialmente a partir de la posición del píxel a través de la cual pasa un rayo y del punto de referencia de proyección:

$$\mathbf{u} = \frac{\mathbf{P}_{pix} - \mathbf{P}_{prp}}{|\mathbf{P}_{pix} - \mathbf{P}_{prp}|} \quad (10.67)$$

Aunque no es necesario que  $\mathbf{u}$  sea un vector unitario, si lo es, se simplificarán algunos de los cálculos.



**FIGURA 10.58.** Descripción de un rayo con un vector de posición inicial  $\mathbf{P}_0$  y un vector de dirección unitario  $\mathbf{u}$ .

Para localizar el punto de intersección del rayo con una superficie, utilizamos la ecuación de la superficie para hallar la posición  $\mathbf{P}$ , como se representa en la Ecuación 10.66. Esto nos da un valor para el parámetro  $s$ , que es la distancia desde  $\mathbf{P}_0$  hasta el punto de intersección con la superficie a lo largo de la trayectoria del rayo.

En cada superficie intersectada, los vectores  $\mathbf{P}_0$  y  $\mathbf{u}$  se actualizan para los rayos secundarios en el punto de intersección entre el rayo y la superficie. Para los rayos secundarios, la dirección de reflexión para  $\mathbf{u}$  es  $\mathbf{R}$  y la dirección de transmisión es  $\mathbf{T}$ . Cuando se detecta una intersección entre un rayo secundario y una superficie, se resuelve un sistema formado por la ecuación del rayo y la ecuación de la superficie para obtener las coordenadas de intersección, después de lo cual se actualiza el árbol binario y se genera el siguiente conjunto de rayos de reflexión y refracción.

Se han desarrollado algoritmos eficientes de cálculos de intersecciones entre rayos y superficies para las formas que más comúnmente aparecen en las escenas, incluyendo diversas superficies de tipo *spline*. El procedimiento general consiste en combinar la ecuación del rayo con las ecuaciones que describen la superficie y resolver el sistema para obtener el valor del parámetro  $s$ . En muchos casos, se utilizan métodos numéricos de localización de raíces y cálculos incrementales para localizar los puntos de intersección con una superficie. Para objetos complejos, a menudo resulta conveniente transformar la ecuación del rayo al sistema de coordenadas local en el que el objeto está definido. Asimismo, los cálculos de intersección para un objeto complejo pueden simplificarse en muchos casos transformando el objeto a una forma más adecuada. Como ejemplo, podemos efectuar los cálculos de trazado de rayos para una elipsoide transformando las ecuaciones del rayo y de la superficie a un problema de intersección con esferas. La Figura 10.59 muestra una escena con trazado de rayos que contiene múltiples objetos y patrones de textura.

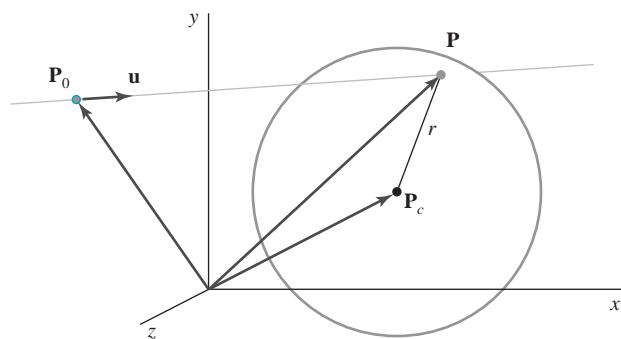
### Intersecciones entre rayos y esferas

Los objetos más simples para efectuar un trazado de rayos son las esferas. Si tenemos una esfera de radio  $r$  y centro  $\mathbf{P}_c$  (Figura 10.60), cualquier punto  $\mathbf{P}$  de la superficie satisfará la ecuación de la esfera:

$$|\mathbf{P} - \mathbf{P}_c|^2 - r^2 = 0 \quad (10.68)$$



**FIGURA 10.59.** Una escena de un trazado de rayos que muestra los efectos de reflexión global de los patrones de textura de las superficies. (Cortesía de Sun Microsystems.)



**FIGURA 10.60.** Un rayo intersecta una esfera con radio  $r$  y centro  $\mathbf{P}_c$ .

Sustituyendo la Ecuación del rayo 10.66 para  $\mathbf{P}$  en la ecuación anterior, tenemos:

$$|\mathbf{P}_0 - s\mathbf{u} - \mathbf{P}_c|^2 - r^2 = 0 \quad (10.69)$$

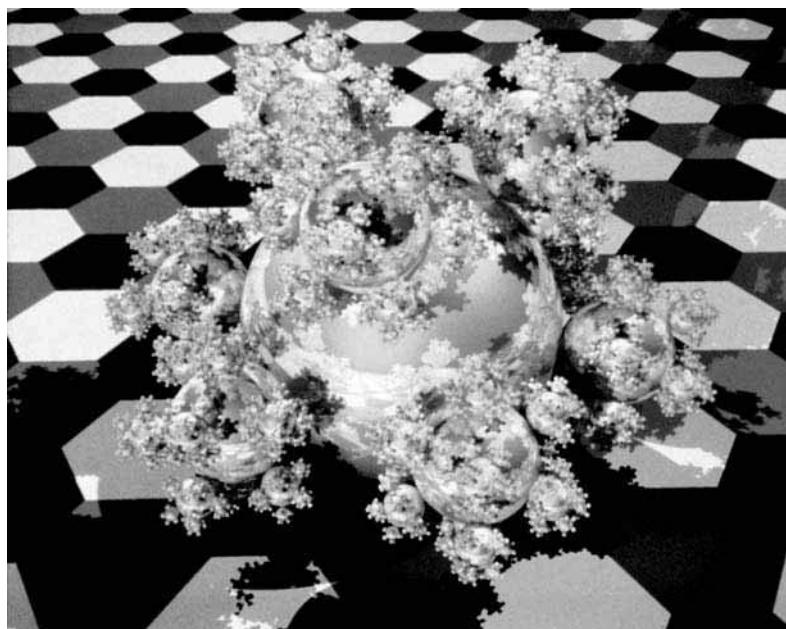
Si representamos  $\mathbf{P}_c - \mathbf{P}_0$  como  $\Delta\mathbf{P}$  y expandimos el producto escalar, obtenemos la ecuación cuadrática:

$$s^2 - 2(\mathbf{u} \cdot \Delta\mathbf{P})s + (|\Delta\mathbf{P}|^2 - r^2) = 0 \quad (10.70)$$

cuya solución es:

$$s = \mathbf{u} \cdot \Delta\mathbf{P} \pm \sqrt{(\mathbf{u} \cdot \Delta\mathbf{P})^2 - |\Delta\mathbf{P}|^2 + r^2} \quad (10.71)$$

Si el discriminante es negativo, o bien el rayo no intersecta la esfera o la esfera se encuentra detrás de  $\mathbf{P}_0$ . En cualquiera de los dos casos, podemos eliminar la esfera de los cálculos posteriores, ya que asumimos que la escena se encuentra delante del plano de proyección. Cuando el discriminante no es negativo, se obtienen las coordenadas de intersección con la superficie a partir de la ecuación del rayo 10.66 utilizando el más pequeño de los dos valores de la Ecuación 10.71. La Figura 10.61 muestra una escena de trazado de rayos que contiene un copo de nieve formado por esferas brillantes y donde se ilustran los efectos de reflexión global en las superficies que se pueden obtener mediante trazado de rayos.



**FIGURA 10.61.** Un copo de nieve representado mediante trazado de rayos y donde se utilizan 7381 esferas y 3 fuentes luminosas. (Cortesía de Eric Haines, Autodesk, Inc.)

Resulta posible realizar algunas optimizaciones en los cálculos de intersección entre el rayo y la esfera con el fin de reducir el tiempo de procesamiento. Además, la Ecuación 10.71 está sujeta a errores de redondeo cuando se procesa una esfera de pequeño tamaño que está muy alejada de la posición inicial del rayo. Es decir, si:

$$r^2 \ll |\Delta\mathbf{P}|^2$$

podríamos perder el término  $r^2$  debido a los errores de precisión de  $|\Delta\mathbf{P}|_2$ . Podemos evitar esto en la mayoría de los casos reordenando los cálculos correspondientes a la distancia  $s$  de la forma siguiente:

$$s = \mathbf{u} \cdot \Delta\mathbf{P} \pm \sqrt{r^2 - |\Delta\mathbf{P} - (\mathbf{u} \cdot \Delta\mathbf{P})\mathbf{u}|^2} \quad (10.72)$$

### Intersecciones entre rayos y poliedros

Los cálculos de intersección para poliedros son más complicados que para las esferas. Por tanto, a menudo resulta más eficiente procesar un poliedro realizando un test inicial de intersección sobre un volumen que lo encierre. Por ejemplo, la Figura 10.62 muestra un poliedro dentro de una esfera. Si un rayo no intersecta la esfera circunscrita, eliminamos el poliedro de las comprobaciones posteriores. En caso contrario, identificamos a continuación las caras frontales del poliedro como esos polígonos que satisfagan la desigualdad:

$$\mathbf{u} \cdot \mathbf{N} < 0 \quad (10.73)$$

donde  $\mathbf{N}$  es la normal a la superficie del polígono. Para cada cara del poliedro que satisfaga la condición 10.73, resolvemos la ecuación del plano:

$$\mathbf{N} \cdot \mathbf{P} = -D \quad (10.74)$$

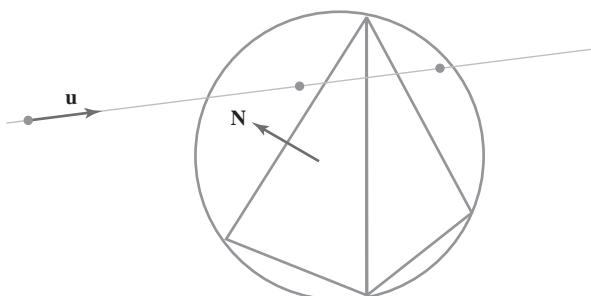
para calcular la posición  $\mathbf{P}$  que también satisfaga la ecuación del rayo 10.66. Aquí,  $\mathbf{N} = (A, B, C)$  y  $D$  es el cuarto parámetro del plano. La posición  $\mathbf{P}$  estará tanto en el plano como en el trayecto del rayo si se cumple que:

$$\mathbf{N} \cdot (\mathbf{P}_0 + s\mathbf{u}) = -D \quad (10.75)$$

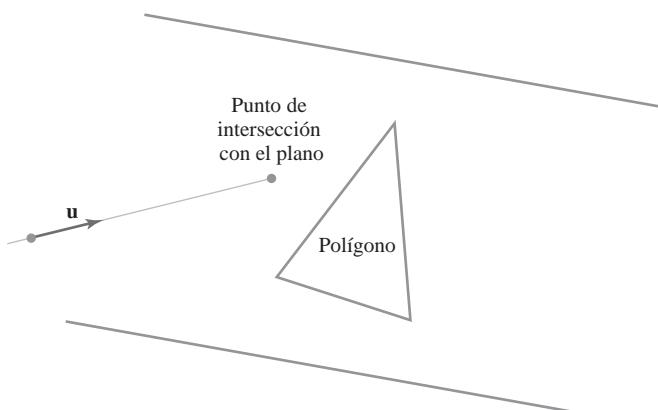
y la distancia desde la posición inicial del rayo hasta el plano es:

$$s = -\frac{D + \mathbf{N} \cdot \mathbf{P}_0}{\mathbf{N} \cdot \mathbf{u}} \quad (10.76)$$

Esto nos da una posición sobre el plano infinito que contiene la cara poligonal, pero esta posición puede no estar dentro de las fronteras del polígono (Figura 10.63). Por tanto, necesitamos realizar una comprobación de si el punto está dentro del polígono (Sección 3.15) para determinar si el rayo intersecta esta cara del poliedro. Realizaremos este test para cada cara que satisfaga la desigualdad 10.73. La distancia  $s$  más pequeña hasta un polígono intersectado identifica la posición de intersección sobre la superficie del poliedro. Si ninguno de



**FIGURA 10.62.** Un poliedro encerrado en una esfera circunscrita.



**FIGURA 10.63.** Intersección de un rayo en el plano de un polígono.

los puntos de intersección de la Ecuación 10.76 son puntos interiores a los polígonos, el rayo no intersecta el poliedro.

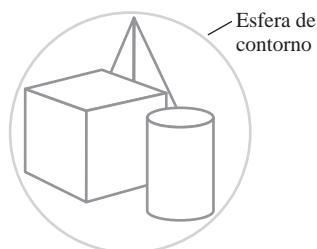
### Reducción de los cálculos de intersección con los objetos

Los cálculos de intersección entre las superficies y los rayos pueden representar hasta un 95 por ciento del tiempo de procesamiento en una representación mediante trazado de rayos de una escena. Para una escena con muchos objetos, la mayor parte del tiempo de procesamiento para cada rayo se invierte en comprobar objetos que no son visibles según el trazado del rayo. Por tanto, se han desarrollado diversos métodos para reducir el tiempo de procesamiento invertido en estos cálculos de intersecciones.

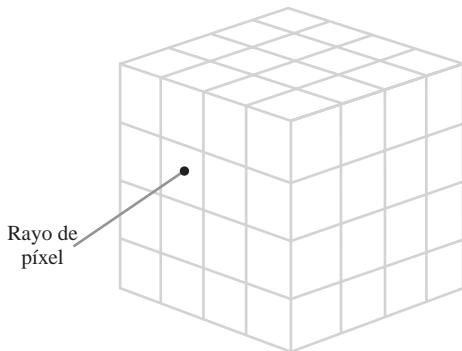
Un método para reducir los cálculos de intersecciones consiste en encerrar grupos de objetos adyacentes dentro de un volumen de contorno, como por ejemplo una esfera o un paralelepípedo (Figura 10.64). Entonces, podemos ver si existen intersecciones de los rayos con el volumen de contorno. Si el rayo no intersecta la superficie del contorno, eliminamos las superficies encerradas de las comprobaciones sucesivas de intersección. Esta técnica puede ampliarse para incluir una jerarquía de volúmenes de contorno. En otras palabras, podemos encerrar varios volúmenes de contorno dentro de un volumen mayor y llevar a cabo las comprobaciones de intersección jerárquicamente: primero comprobamos el volumen más externo, después comprobamos en caso necesario los volúmenes de contorno interiores, etc.

### Métodos de subdivisión espacial

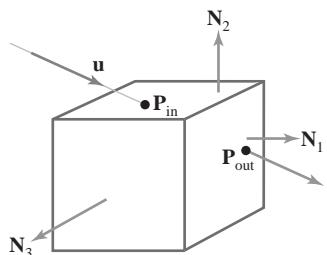
Otra forma de reducir los cálculos de intersecciones consiste en utilizar procedimientos de **subdivisión espacial**. Podemos encerrar una escena completa dentro de un cubo y luego subdividir sucesivamente el cubo hasta que cada subregión (celda) no contenga más de un número máximo de superficies prefijado. Por ejemplo, podemos obligar a que cada celda no contenga más de una superficie. Si hay disponibles capacidades de procesamiento vectoriales y en paralelo, el número máximo de superficies por celda puede determinarse según el tamaño de los registros de vectores y el número de procesadores. La subdivisión espacial del tubo puede alma-



**FIGURA 10.64.** Un grupo de objetos dentro de una esfera.



**FIGURA 10.65.** Intersección de un rayo con un cubo que encierra a todos los objetos de una escena.



**FIGURA 10.66.** Recorrido del rayo a través de una sub-región (celda) de un cubo que encierra a una escena.

cenarse en un árbol octal o en un árbol de partición binaria. Además, podemos realizar una *subdivisión uniforme* dividiendo cada cubo en ocho octantes de igual tamaño en cada caso, o podemos realizar una *subdivisión adaptativa* subdividiendo únicamente aquellas regiones del tubo que contengan objetos.

A continuación, trazamos los rayos a través de las celdas individuales del cubo, realizando comprobaciones de intersección sólo dentro de aquellas celdas que contengan superficies. La primera superficie intersecada será la superficie visible para ese rayo. Existe un compromiso, sin embargo, entre el tamaño de celda y el número de superficies por celda. A medida que reducimos el número máximo de superficies permitidas por celda, se reduce la cantidad de procesamiento necesario para las pruebas de intersección con las superficies, pero esto reduce también el tamaño de las celdas, por lo que son necesarios más cálculos para determinar la trayectoria del rayo a través de las celdas.

La Figura 10.65 ilustra la intersección del rayo de un píxel con la cara frontal de un cubo que rodea a una escena. La posición de intersección en la cara frontal del cubo identifica la celda inicial que tiene que atravesar este rayo. Despues, procesamos el rayo a través de las celdas del cubo determinando las coordenadas de las posiciones de entrada y de salida (Figura 10.66). En cada celda no vacía, comprobamos las intersecciones con las superficies y este procesamiento continúa hasta que el rayo intersecta una superficie de un objeto o sale del cubo delimitador.

Dado un vector unitario de dirección del rayo  $\mathbf{u}$  y una posición de entrada del rayo  $\mathbf{P}_{in}$  para una celda, identificamos las caras potenciales de salida de una celda como aquellas que satisfacen la desigualdad:

$$\mathbf{u} \cdot \mathbf{N}_k > 0 \quad (10.77)$$

donde  $\mathbf{N}_k$  representa el vector unitario normal a la superficie para la cara  $k$  de la celda. Si los vectores unitarios normales a las caras de la celda de la Figura 10.66 están alineados con los ejes de coordenadas cartesianas, entonces,

$$\mathbf{N}_k = \begin{cases} (\pm 1, 0, 0) \\ (0, \pm 1, 0) \\ (0, 0, \pm 1) \end{cases} \quad (10.78)$$

y podemos determinar los tres planos candidatos de salida simplemente comprobando el signo de cada componente de  $\mathbf{u}$ . La posición de salida en cada plano candidato se obtiene a partir de la ecuación del rayo:

$$\mathbf{P}_{out,k} = \mathbf{P}_{in} + s_k \mathbf{u} \quad (10.79)$$

donde  $s_k$  es la distancia según el rayo desde  $\mathbf{P}_{in}$  a  $\mathbf{P}_{out,k}$ . Sustituyendo la ecuación del rayo en la ecuación del plano para cada cara de la celda, tenemos:

$$\mathbf{N}_k \cdot \mathbf{P}_{out,k} = -D_k \quad (10.80)$$

y la distancia del rayo a cada cara candidata de salida se calcula como:

$$s_k = \frac{-D_k - \mathbf{N}_k \cdot \mathbf{P}_{\text{in}}}{\mathbf{N}_k \cdot \mathbf{u}} \quad (10.81)$$

El valor más pequeño calculado para  $s_k$  identifica la cara de salida para la celda. Si las caras de las celdas están alineadas en paralelo con los planos de coordenadas cartesianas, los vectores normales  $\mathbf{N}_k$  son los vectores unitarios 10.78 según los ejes y podemos simplificar los cálculos de la Ecuación 10.81. Por ejemplo, si un plano candidato de salida tiene el vector normal  $(1, 0, 0)$ , entonces para ese plano tendremos:

$$s_k = \frac{x_k - x_0}{u_x} \quad (10.82)$$

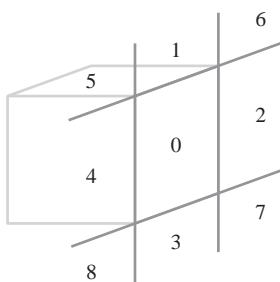
donde  $\mathbf{u} = (u_x, u_y, u_z)$ ,  $x_k = -D_k$  es la posición de coordenadas del plano candidato de salida y  $x_0$  es la posición de coordenadas de la cara de entrada de la celda.

Pueden realizarse diversas modificaciones a los procedimientos de recorrido de las celdas para acelerar el procesamiento. Una posibilidad consiste en elegir un plano de salida candidato  $k$  que sea perpendicular a la dirección de la componente de  $\mathbf{u}$  de mayor tamaño. Este plano de salida candidato se divide entonces en sectores, como se muestra en el ejemplo de la Figura 10.67. El sector del plano candidato que contenga  $\mathbf{P}_{\text{out},k}$  determinará el verdadero plano de salida. Por ejemplo, si el punto de intersección  $\mathbf{P}_{\text{out},k}$  está en el sector 0 para el plano de ejemplo de la Figura 10.67, el plano que habíamos elegido es el verdadero plano de salida y habríamos terminado. Si el punto de intersección está en el sector 1, el verdadero plano de salida será el plano superior y necesitaremos simplemente calcular el punto de salida en la frontera superior de la celda. De la misma forma, el sector 3 identifica el plano inferior como verdadero plano de salida y los sectores 4 y 2 identifican como verdadero plano de salida los planos izquierdo o derecho de la celda, respectivamente. Cuando el punto de salida en el plano candidato cae en los sectores 5, 6, 7 u 8, debemos llevar a cabo dos cálculos de intersección adicionales para identificar el verdadero plano de salida. La implementación de estos métodos en máquinas vectoriales de procesamiento en paralelo permite obtener incrementos aún mayores en la velocidad.

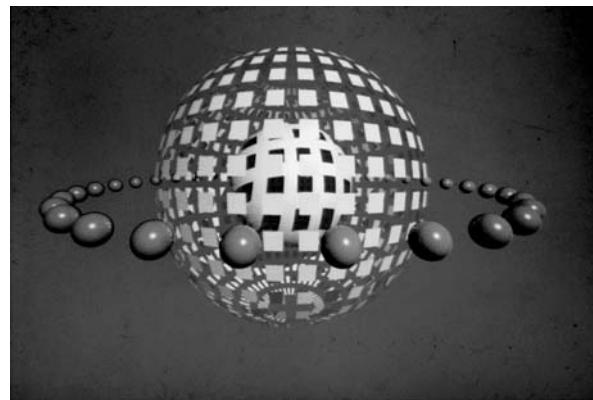
La escena de la Figura 10.68 se obtuvo mediante trazado de rayos empleando métodos de subdivisión espacial. Sin la subdivisión espacial, los cálculos de trazado de rayos tardaban 10 veces más. La eliminación de los polígonos también permitía en este ejemplo acelerar el procesamiento. Para una escena que contenga 20.48 esfera y ningún polígono, el mismo algoritmo se ejecutaba 46 veces más rápido que el algoritmo básico de trazado de rayos.

La Figura 10.69 ilustra otro esquema obtenido mediante trazado de rayos utilizando subdivisión espacial y métodos de procesamiento en paralelo. Esta imagen de la escultura de Rodin *El Pensador* se obtuvo mediante trazado de rayos en 24 segundos utilizando más de 1.5 millones de rayos.

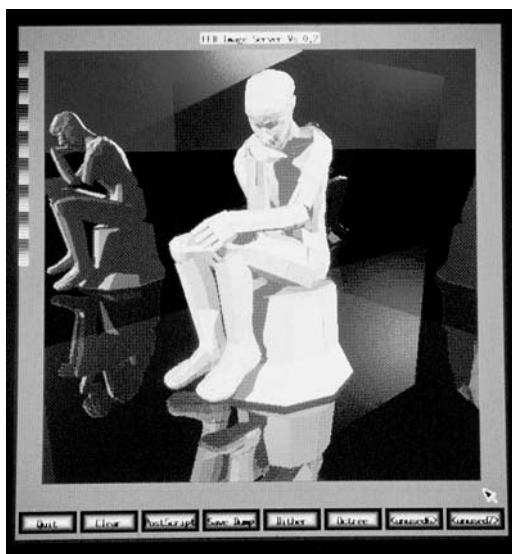
Para obtener la escena de la Figura 10.70 se empleó una técnica de *búfer luminoso*, que es un tipo de particionamiento espacial. Aquí, hay un cubo centrado en cada fuente luminosa puntual y cada cara del cubo se partitiona utilizando una cuadrícula de elementos cuadrados. Entonces, el paquete de trazado de rayos mantiene una lista ordenada de los objetos que son visibles para los haces luminosos a través de cada cuadrado,



**FIGURA 10.67.** Un plano candidato de salida de ejemplo, junto con sus sectores numerados.



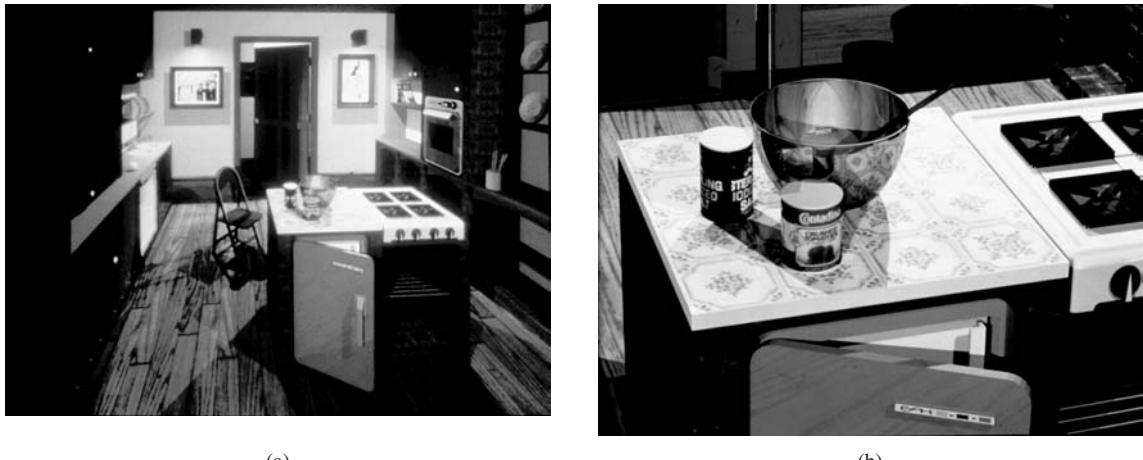
**FIGURA 10.68.** Una escena obtenida mediante trazado de rayos en paralelo que contiene 37 esferas y 720 superficies poligonales. El algoritmo de trazado de rayos utiliza 9 rayos por píxel y una profundidad de árbol de 5. Los métodos de subdivisión espacial permitían procesar la escena 10 veces más rápido que el algoritmo básico de trazado de rayos en un Alliant FX/8. (Cortesía de Lee-Hian Quek, Oracle Corporation, Redwood Shores, California.)



**FIGURA 10.69.** Esta escena obtenida mediante trazado de rayos tardaba 24 segundos en representarse en una computadora paralela Kendall Square Research KSR1 con 32 procesadores. La escultura de Rodin *El Pensador* fue modelada con 3.036 primitivas. Se usaron dos fuentes luminosas y un rayo principal por píxel para obtener los efectos globales de iluminación a partir de los 1.675.776 rayos procesados. (Cortesía de M. J. Keates y R. J. Hubbald, Departamento de Computer Science, Universidad de Manchester, Reino Unido.)

con el fin de acelerar el procesamiento de los rayos de sombra. Como medio para determinar los efectos de iluminación superficial, se calcula un cuadrado para cada rayo de sombra y luego se procesa el rayo de sombra de acuerdo con la lista de objetos correspondientes a dicho cuadrado.

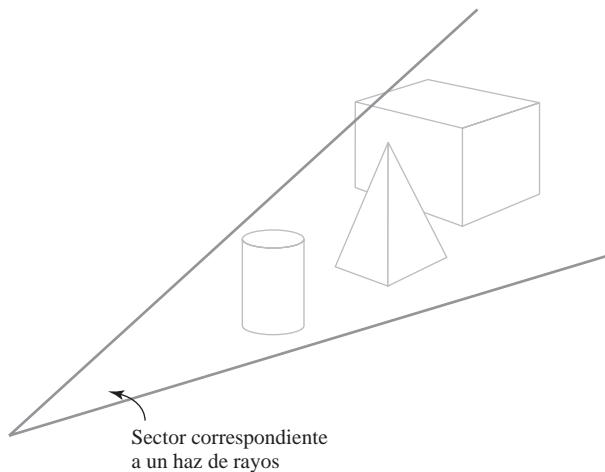
Las comprobaciones de intersección en los programas de trazado de rayos también pueden reducirse mediante procedimientos de subdivisión direccional, considerando sectores que contengan un haz de rayos. Dentro de cada sector, podemos ordenar las superficies según su profundidad, como en la Figura 10.71. Cada rayo necesita entonces comprobar únicamente los objetos correspondientes al sector que contiene a dicho rayo.



(a)

(b)

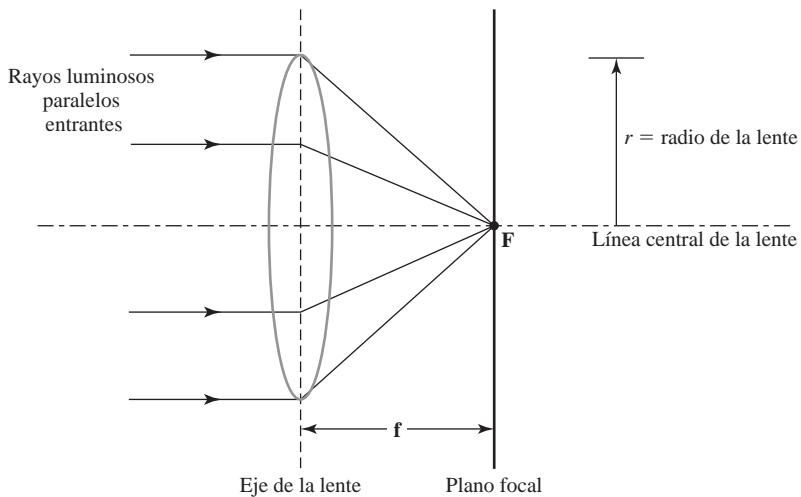
**FIGURA 10.70.** Escena de una habitación iluminada con 5 fuentes luminosas y representada (a) utilizando la técnica de trazado de rayos basada en búfer luminoso para procesar los rayos de sombra. Un primer plano (b) de parte de la habitación mostrada en (a) ilustra los efectos globales de iluminación. La habitación está modelada por 1298 polígonos, 4 esferas, 76 cilindros y 35 cuádricas. El tiempo de representación fue de 246 minutos en un VAX 11/780, comparado con 602 minutos sin la utilización de búferes luminosos. (*Cortesía de Eric Haines y Donald P. Greenberg, Program of Computer Graphics, Cornell University.*)



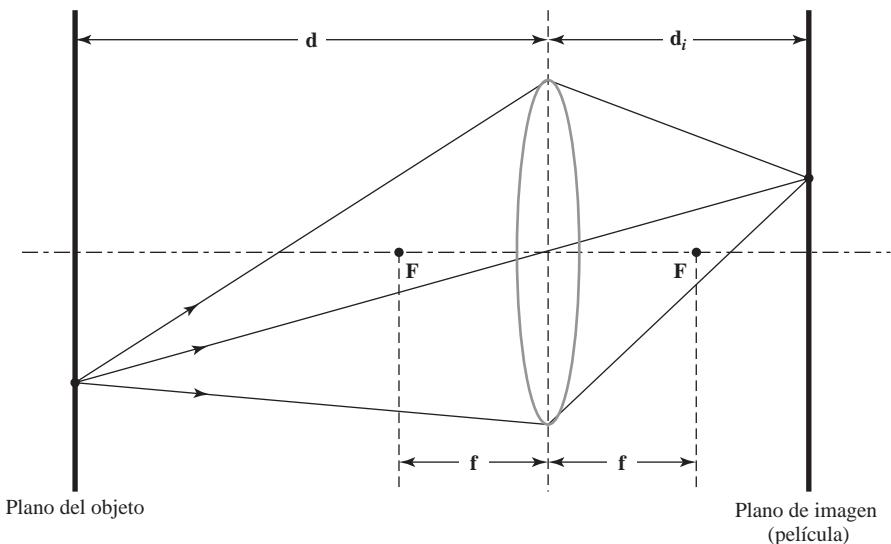
**FIGURA 10.71.** Subdivision direccional del espacio. Los rayos de los píxeles en el sector indicado realizan comprobaciones de intersección en orden de profundidad únicamente para las superficies contenidas dentro de ese sector.

### Simulación de los efectos de enfoque de la cámara

Para modelar los efectos de la cámara en una escena, especificamos la longitud de enfoque y otros parámetros para un objetivo convexo (o apertura de la cámara) que haya que colocar delante del plano de proyección. Los parámetros del objetivo se configuran entonces de modo que algunos objetos de la escena puedan estar enfocados mientras que otros aparecerán desenfocadas. La distancia focal del objetivo es la distancia desde el centro del objetivo hasta el punto focal  $F$ , que es el punto de convergencia para un conjunto de rayos paralelos que pasan a través de la lente, como se ilustra en la Figura 10.72. Un valor típico para la distancia focal de una cámara de 35 mm es  $f = 50$  mm. Las aperturas de las cámaras suelen describirse mediante un parámetro  $n$ , denominado número-f o f-stop, que es el cociente entre la distancia focal y el diámetro de apertura:



**Figura 10.72.** Vista lateral de un objetivo. Los rayos paralelos son enfocados por la lente sobre una posición situada en el plano focal, que está a la distancia  $f$  del centro de la lente.



**FIGURA 10.73.** Parámetros de un objetivo. Un objeto a una distancia  $d$  del objetivo estará enfocado sobre el plano de imagen a una distancia  $d_i$  de la lente.

$$n = \frac{f}{2r} \quad (10.83)$$

Por tanto, podemos utilizar el radio  $r$  o el número-f  $n$ , junto con la distancia focal  $f$ , para especificar los parámetros de la cámara. Para conseguir un modelo de enfoque más preciso, podríamos utilizar el tamaño de la película (anchura y altura) y la distancia focal para simular los efectos de la cámara.

Los algoritmos de trazado de rayos determinan normalmente los efectos de enfoque utilizando la **ecuación de una lente fina** de la óptica geométrica:

$$\frac{1}{d} + \frac{1}{d_j} = \frac{1}{f} \quad (10.84)$$

El parámetro  $d$  es la distancia desde el centro de la lente hasta la posición de un objeto y  $d_i$  es la distancia desde el centro de la lente al plano de imagen, donde ese objeto estará enfocado. El punto del objeto y el de su imagen se encuentran en lados opuestos de la lente, a lo largo de una línea que pasa por el centro de la lente y  $d > f$  (Figura 10.73). Por tanto, para enfocar un objeto concreto situado a una distancia  $d$  de la lente, situamos el plano de píxeles a una distancia  $d_i$  por detrás del objetivo.

Para una posición de la escena situada a una cierta distancia  $d' \neq d$ , el punto proyectado estará desenfocado en el plano de imagen. si  $d' > d$ , el punto estará enfocado en una posición situada delante del plano de imagen, mientras que si  $d' < d$ , el punto estará enfocado en una posición situada detrás del plano de imagen. La proyección de un punto situado en la posición  $d'$  sobre el plano de imagen es aproximadamente un pequeño círculo, denominado **círculo de confusión**, y el diámetro de este círculo puede calcularse como:

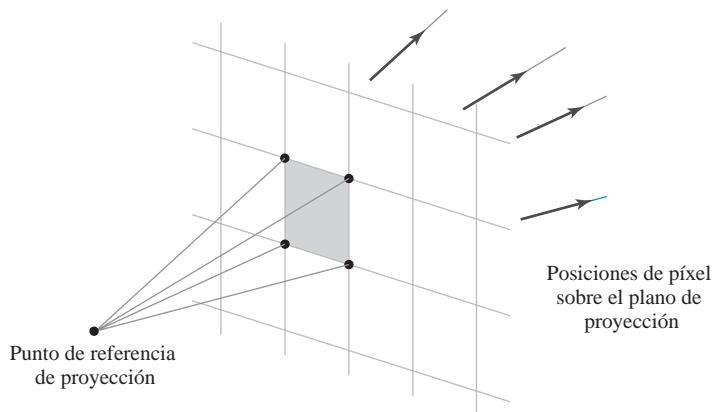
$$2r_c = \frac{|d' - d|f}{nd} \quad (10.85)$$

Podemos elegir los parámetros de la cámara con el fin de minimizar el tamaño del círculo de confusión para un cierto rango de distancias, denominado **profundidad de campo** de la cámara. Además, se trazan múltiples rayos para cada píxel con el fin de muestrear distintas posiciones sobre el área del objetivo; hablaremos de estos *métodos de trazado de rayos distribuidos* en una sección posterior.

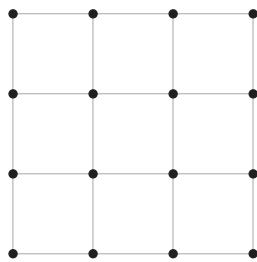
### Trazado de rayos con antialiasing

Dos técnicas básicas de *antialiasing* empleadas en los algoritmos de trazado de rayos son el *supermuestreo* y el *muestreo adaptativo*. El muestreo en el trazado de rayos es una extensión de los métodos de *antialiasing* expuestos en la Sección 4.17. En el supermuestreo y en el muestreo adaptativo, el píxel se trata como un área cuadrada finita en lugar de como un único punto. El supermuestreo utiliza normalmente múltiples rayos equiespaciados (muestras) para cada área de píxel. El muestreo adaptativo utiliza rayos no equiespaciados en algunas regiones del área de píxel. Por ejemplo, pueden usarse más rayos cerca de los bordes de los objetos con el fin de obtener una mejor estimación de las intensidades de los píxeles. (Otro método de muestreo consiste en distribuir aleatoriamente los rayos a lo largo del área del píxel. Hablaremos de esta técnica en la siguiente sección). Cuando se utilizan múltiples rayos por píxel, las intensidades de los rayos de píxel se promedian para obtener la intensidad global del píxel.

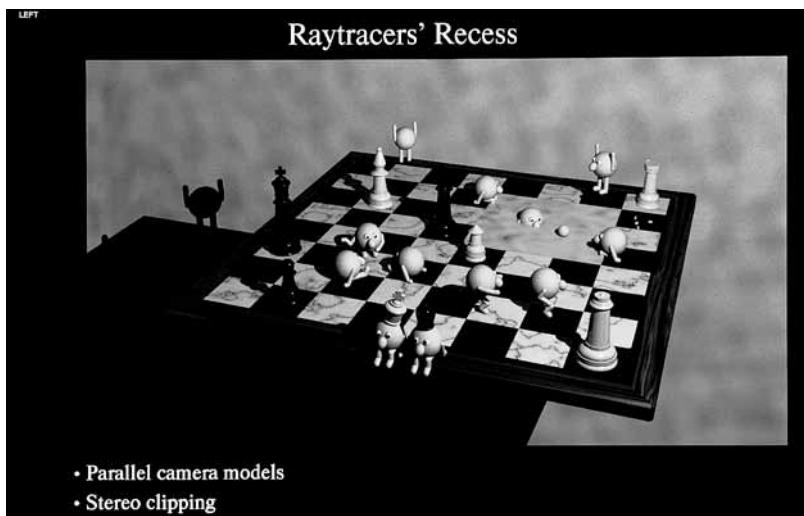
La Figura 10.74 ilustra un procedimiento simple de supermuestreo. Aquí, se genera un rayo a través de cada esquina del píxel. Si las intensidades calculadas para los cuatro rayos no son aproximadamente iguales, o si algún pequeño objeto cae entre los cuatro rayos, dividimos el área del píxel en subpíxeles y repetimos el proceso. Como ejemplo, el píxel de la Figura 10.75 se divide en nueve subpíxeles utilizando 16 rayos, uno en cada esquina de cada subpíxel. Entonces, se utiliza muestreo adaptativo para subdividir aún más dichos sub-



**FIGURA 10.74.** Supermuestreo con cuatro rayos por píxel, uno en cada esquina del píxel.



**FIGURA 10.75.** Subdivisión de un píxel en 9 subpíxeles con un rayo en cada esquina de cada subpíxel.



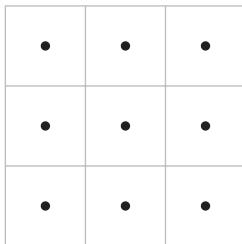
**FIGURA 10.76.** Una escena generada mediante trazado de rayos por subdivisión adaptativa. (Cortesía de Jerry Farm.)

píxeles que rodeen a un pequeño objeto o que no tengan rayos de intensidad aproximadamente igual. Este proceso de subdivisión puede continuarse hasta que todos los rayos de subpíxel tengan aproximadamente las mismas intensidades o hasta que se alcance un límite superior, por ejemplo 256, que indicará el número máximo de rayos por píxel.

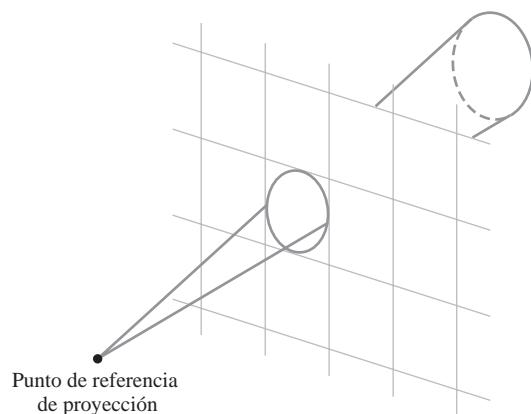
La Figura 10.76 es un ejemplo de escena representada mediante trazado de rayos con subdivisión adaptativa. Se utiliza una fuente luminosa compleja para proporcionar sombras suaves y realistas. Se generaron casi 26 millones de rayos principales, con 33.5 millones de rayos de sombra y 67.3 de rayos de reflexión. Para desarrollar las figuras del juego de ajedrez se emplearon técnicas de figuras articuladas (Sección 13.8). Los patrones de superficie de madera veteada y mármol se generaron mediante métodos de texturas sólidas (Sección 10.16) con una función de ruido.

En lugar de pasar los rayos a través de las esquinas de los píxeles, podemos generar los rayos a través de los centros de los subpíxeles, como en la Figura 10.77. Con este enfoque, podemos ponderar los rayos de acuerdo con alguno de los esquemas de muestreo analizados en la Sección 4.17.

Otro método para el *antialiasing* de las escenas generadas consiste en tratar cada rayo de píxel como si fuera un cono, como se muestra en la Figura 10.78. Sólo se genera un rayo por píxel, pero ahora el rayo tiene una sección transversal finita. Para determinar el porcentaje de recubrimiento del área del píxel por parte de los objetos, calculamos la intersección del cono del píxel en la superficie del objeto. Para una esfera, esto requiere hallar la intersección de dos círculos. Para un poliedro, tenemos que hallar la intersección de un círculo con un polígono.



**FIGURA 10.77.** Posiciones de rayos centradas en las áreas de subpixel.



**FIGURA 10.78.** Un rayo de pixel con forma de cono.

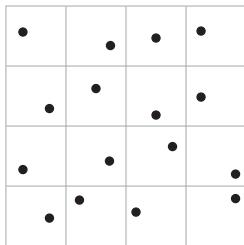
### Trazado de rayos distribuido

Se trata de un método de muestreo estocástico que distribuye aleatoriamente los rayos de acuerdo con los diversos parámetros de un modelo de iluminación. Los parámetros de iluminación incluyen el área del píxel, las direcciones de reflexión y refracción, el área del objetivo de la cámara y el tiempo. Los efectos del *aliasing* se sustituyen así con ruido de bajo nivel, que mejora la calidad de una imagen y permite modelar de manera más precisa los efectos de las superficies brillantes y translúcidas, las aperturas finitas de cámara, las fuentes luminosas finitas y la visualización desenfocada de los objetos en movimiento. El **trazado de rayos distribuido** (también denominado **trazado de rayos de distribución**) proporciona esencialmente una evaluación de Monte Carlo de las múltiples integrales que aparecen en una descripción física precisa de la iluminación de una superficie.

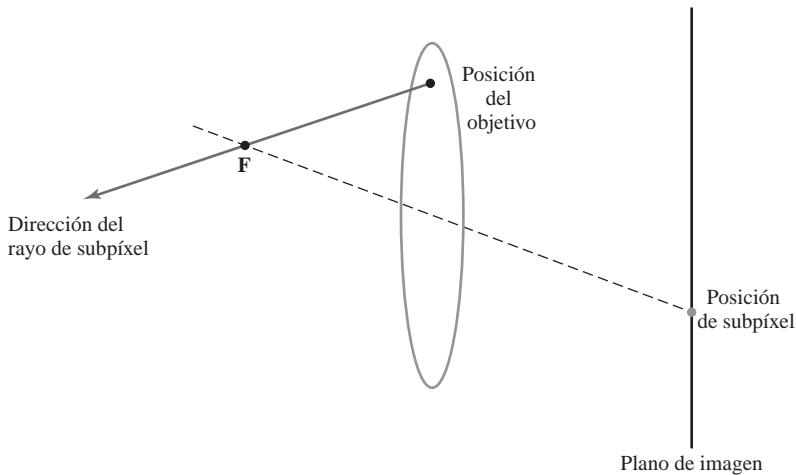
El muestreo de los píxeles se lleva a cabo distribuyendo aleatoriamente una serie de rayos a lo largo del área del píxel. Sin embargo, seleccionar las posiciones de los rayos de forma completamente aleatoria puede hacer que los rayos se agrupen en una pequeña región del área del píxel, dejando sin muestrear una buena parte de éste. Una mejor aproximación para la distribución de los rayos en el área del píxel consiste en utilizar una técnica denominada *fluctuación (jittering)* sobre una cuadrícula regular de subpixeles. Esto se suele llevar a cabo dividiendo inicialmente el área del píxel (un cuadrado unitario) en las 16 subáreas que se muestran en la Figura 10.79 y generando una *posición fluctuante aleatoria* en cada subárea. Las posiciones aleatorias de los rayos se obtienen haciendo fluctuar las coordenadas de los centros de cada subárea en pequeñas cantidades,  $\delta_x$  y  $\delta_y$ , donde  $-0.5 < \delta_x, \delta_y < 0.5$ . Entonces seleccionamos la posición ajustada como  $(x + \delta_x, y + \delta_y)$ , donde  $(x, y)$  es la posición central del píxel.

Aleatoriamente se asignan códigos enteros de 1 a 16 a cada uno de los 16 rayos y se utiliza una tabla de sustitución para obtener valores para los otros parámetros, como el ángulo de reflexión y el tiempo. Cada rayo de subpíxel se traza a través de la escena y se procesa para determinar la contribución de intensidad correspondiente a cada rayo. Las 16 intensidades de rayo se promedian entonces para obtener la intensidad global del píxel. Si las intensidades de los subpixeles varían demasiado, podemos subdividir el píxel todavía más.

Para modelar los efectos del objetivo de la cámara, procesamos los rayos de los píxeles a través de una lente que se sitúa delante del plano de píxeles. Como hemos indicado anteriormente, la cámara se simula utilizando una distancia focal y otros parámetros, de modo que una serie de objetos seleccionados queden enfocados. Entonces, distribuimos los rayos de los subpixeles por el área de apertura. Suponiendo que tengamos 16 rayos por píxel, podemos subdividir el área de apertura en 16 zonas y asignar a continuación a cada subpíxel una posición central en una de las zonas. Puede utilizarse el siguiente procedimiento para determinar la distribución de muestreo para el píxel: se calcula una posición ajustada mediante fluctuación para cada cen-



**FIGURA 10.79.** Muestreo de un píxel utilizando 16 áreas de subpíxel y una posición fluctuante de rayo con respecto a las coordenadas del centro de cada subárea.

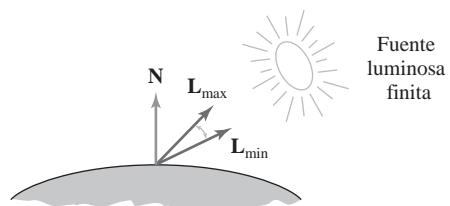
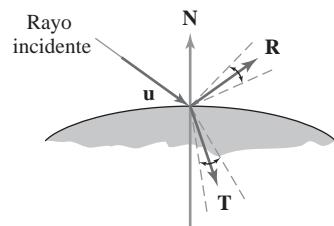


**FIGURA 10.80.** Distribución de los rayos de subpíxel sobre el objetivo de una cámara con distancia focal  $f$ .

tro de zona y se proyecta un rayo hacia la escena desde esta posición de zona ajustada, a través del punto focal de la lente. Ubicamos el punto focal del rayo a una distancia  $f$  de la lente, según la línea que sale del centro del subpíxel y pasa por el centro de la lente, como se muestra en la Figura 10.80. Con el plano de píxeles a una distancia  $d_i$  de la lente (Figura 10.73), las posiciones situadas a lo largo del rayo cerca del plano del objetivo (el plano de enfoque) a una distancia  $d$  delante de la lente, estarán enfocadas. Otras posiciones a lo largo del rayo estarán desenfocadas. Para mejorar la visualización de los objetos desenfocados, puede incrementarse el número de rayos de subpíxel.

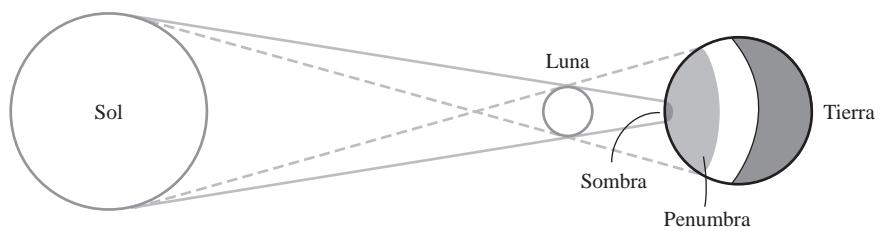
Los trayectos de reflexión y transmisión también se distribuyen por una cierta región espacial. Para simular los brillos de las superficies, los rayos reflejados desde una posición de la superficie se distribuyen alrededor de la dirección de reflexión especular  $\mathbf{R}$  de acuerdo con los códigos de rayo asignados (Figura 10.81). La dispersión máxima en torno a  $\mathbf{R}$  se divide en 16 zonas angulares y cada rayo se refleja según una dirección que se ajusta mediante fluctuación a partir del centro de la zona correspondiente a su código entero. Podemos utilizar el modelo de Phong,  $\cos^{n_s} \phi$ , para determinar la distribución máxima para los ángulos de reflexión. Si el material es transparente, los rayos refractados pueden distribuirse en torno a la dirección de transmisión  $\mathbf{T}$  de forma similar, con el fin de modelar el carácter translúcido de los objetos (Sección 10.4).

Las fuentes de luz no puntuales se pueden tratar distribuyendo una serie de rayos de sombra por todo el área de la fuente de luz, como se ilustra en la Figura 10.82. La fuente luminosa se divide en zonas y a los rayos de sombras se les asignan direcciones fluctuantes según las diversas zonas. Además, las zonas pueden ponderarse de acuerdo con la intensidad de la fuente luminosa dentro de dicha zona y de acuerdo también con el tamaño del área proyectada de la zona sobre la superficie del objeto. Entonces, se envían más rayos de sombra a las zonas que tengan un mayor peso. Si algunos rayos de sombra intersectan objetos opacos situados entre la superficie y la fuente luminosa, se genera una penumbra (región parcialmente iluminada) en dicho punto de la superficie. Pero si todos los rayos de sombra quedan bloqueados, el punto de la superficie estará dentro de una región de sombra (completamente oscuro) con respecto a dicha fuente luminosa. La Figura 10.83 ilustra las regiones de sombra y de penumbra sobre una superficie parcialmente oculta con respecto a una fuente luminosa.

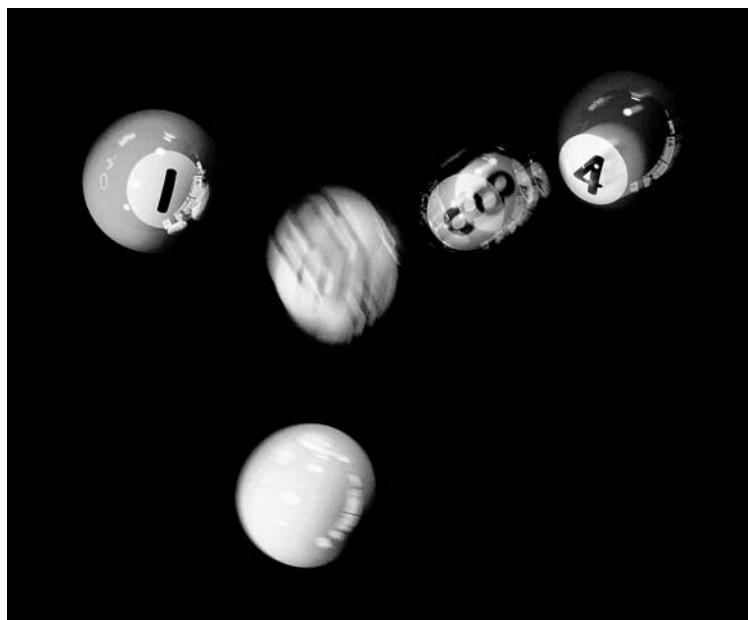


**FIGURA 10.81.** Modelado de objetos brillantes y translúcidos distribuyendo los rayos de subpíxel en torno a la dirección de reflexión  $\mathbf{R}$  y a la dirección de transmisión  $\mathbf{T}$ .

**FIGURA 10.82.** Distribución de los rayos de sombra en torno a una fuente luminosa finita.



**FIGURA 10.83.** Regiones de sombra y de penumbra creadas por un eclipse solar sobre la superficie de la Tierra.

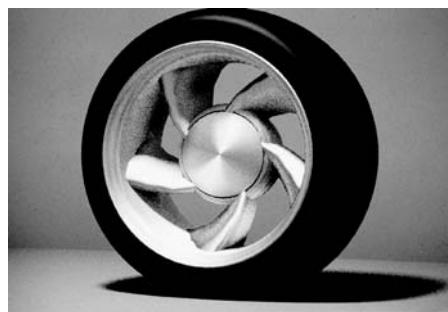


**FIGURA 10.84.** Una escena, titulada *1984*, representada utilizando trazado de rayos distribuido. Los efectos de iluminación incluyen el desenfoque del movimiento, penumbras y reflexiones superficiales para múltiples fuentes luminosas de tamaño finito. (Cortesía de Pixar. © 1984 Pixar. Todos los derechos reservados.)

Podemos crear el efecto de desenfoque de movimiento distribuyendo los rayos a lo largo del tiempo. Se determina un tiempo total y de la imagen y una serie de subdivisiones de este tiempo de imagen de acuerdo con la dinámica de los movimientos requeridos en la escena. Los intervalos temporales se etiquetan con códigos

gos enteros y a cada rayo se le asigna un tiempo fluctuante dentro del intervalo correspondiente al código del rayo. Entonces, los objetos se mueven a sus posiciones correspondientes a dicho punto y se traza el rayo a través de la escena. Para los objetos con mucho desenfoque se utilizan rayos adicionales. Para reducir los cálculos podemos utilizar recuadros o esferas de contorno para las comprobaciones iniciales de intersección de los rayos, es decir, movemos el objeto de contorno de acuerdo con los requisitos de movimiento y comprobamos si se produce una intersección. Si el rayo no intersecta el objeto de contorno, no será necesario procesar las superficies individuales dentro del volumen de contorno. La Figura 10.84 muestra una escena con desenfoque de movimiento. Esta imagen fue representada utilizando trazado de rayos distribuido con 4096 por 3550 píxeles y 16 rayos por píxel. Además de las reflexiones con desenfoque de movimiento, las sombras se muestran con áreas de penumbra que son consecuencia de las fuentes luminosas finitas que iluminan la mesa de billar.

En las Figuras 10.85 y 10.86 se proporcionan ejemplos adicionales de objetos representados con métodos de trazado de rayos distribuido. La Figura 10.87 ilustra los efectos de enfoque, refracción y *antialiasing* que pueden obtenerse con el trazado de rayos distribuido.



**FIGURA 10.85.** Una rueda de aluminio bruñido que muestra los efectos de reflexión y de sombras generados con técnicas de trazado de rayos distribuido. (Cortesía de Stephen H. Westin, Program of Computer Graphics, Cornell University.)



**FIGURA 10.86.** Una escena de una habitación representada mediante métodos de trazado de rayos distribuido. (Cortesía de John Snyder, Jed Lengyel, Devendra Kalra y Al Barr, Computer Graphics Lab, California Institute of Technology. © 1988 Caltech.)



**FIGURA 10.87.** Una escena que muestra los efectos de enfoque, de *antialiasing* y de iluminación que pueden obtenerse con una combinación de métodos de trazado de radios y de radiosidad. Se utilizaron modelos físicos realistas de iluminación para generar los efectos de refracción, incluyendo las cáusticas en la sombra de la copa. (Cortesía de Peter Shirley, Computer Science Department, University of Utah.)

## 10.12 MODELO DE ILUMINACIÓN DE RADIOSIDAD

Aunque el modelo básico de iluminación produce resultados razonables para muchas aplicaciones, hay diversos efectos de iluminación que no quedan descritos de forma precisa mediante las aproximaciones simples de este modelo. Podemos modelar más adecuadamente los efectos de iluminación si tenemos en cuenta las leyes físicas que gobiernan las transferencias de energía radiante dentro de una escena iluminada. Este método para el cálculo de los valores de color de los píxeles se denomina generalmente **modelo de radiosidad**.

### Términos de la energía radiante

En el modelo cuántico de la luz, la energía de la radiación es transportada por los fotones individuales. Para la radiación luminosa monocromática, la energía de cada fotón se calcula como:

$$E_{\text{fotón}, f} = hf \quad (10.86)$$

donde la frecuencia  $f$ , medida en hercios (ciclos por segundo), caracteriza el color de la luz. Una luz azul tiene una alta frecuencia dentro de la banda visible del espectro electromagnético, mientras que una luz roja tiene una baja frecuencia. La frecuencia también nos da la tasa de oscilación para la amplitud de las componentes eléctrica y magnética de la radiación. El parámetro  $h$  es la *constante de Planck*, que tiene el valor  $6.6262 \times 10^{-34}$  julios · sec, independientemente de la frecuencia de la luz.

La energía total para la radiación luminosa monocromática es:

$$E_f = \sum_{\text{todos los fotones}} hf \quad (10.87)$$

La energía radiante para una frecuencia luminosa concreta se denomina también **radiancia espectral**. Sin embargo, cualquier radiación luminosa real, incluso la correspondiente a una fuente «monocromática» con-

tiene un rango de frecuencias. Por tanto, la energía radiante total es la suma para todos los fotones de todas las frecuencias:

$$E = \sum_f \sum_{\text{todos los fotones}} hf \quad (10.88)$$

La cantidad de energía radiante transmitida por unidad de tiempo se denomina **flujo radiante**  $\Phi$ :

$$\Phi = \frac{dE}{dt} \quad (10.89)$$

El flujo radiante también se llama **frecuencia radiante** y se mide en vatios (julios por segundo).

Para obtener los efectos de iluminación para las superficies de una escena, calculamos el flujo radiante por unidad de área que abandona cada superficie. Esta cantidad se denomina **radiosidad**  $B$  o **exitancia radiante**,

$$B = \frac{d\Phi}{dA} \quad (10.90)$$

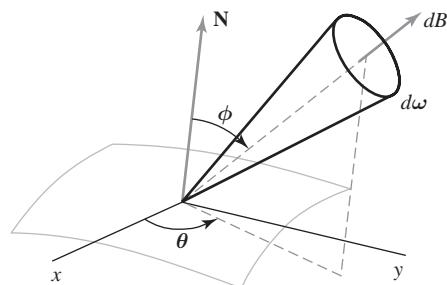
que se mide en unidades de vatios por metro<sup>2</sup>. Y la **intensidad**  $I$  se toma a menudo como medida del flujo radiante en una dirección concreta por unidad de ángulo sólido por unidad de área proyectada, con unidades de vatios/(metro<sup>2</sup> · tereorradianes). Sin embargo, en ocasiones la intensidad se define simplemente como el flujo radiante en una dirección concreta.

Dependiendo de la interpretación del término intensidad, la **radiancia** puede definirse como la intensidad por unidad de área proyectada. Alternativamente, podemos obtener la radiancia a partir del flujo radiante o de la radiosidad por unidad de ángulo sólido.

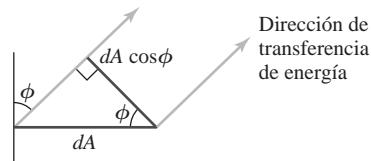
## Modelo básico de radiosidad

Para describir con precisión las reflexiones difusas de una superficie, el modelo de radiosidad calcula las interacciones de energía radiante entre todas las superficies de una escena. Puesto que el conjunto resultante de ecuaciones puede ser extremadamente difícil de resolver, el modelo básico de radiosidad presupone que todas las superficies son reflectores difusos ideales, pequeños y opacos (lambertianos).

Aplicamos el modelo de radiosidad determinando la cantidad diferencial de flujo radiante  $dB$  que sale de cada punto superficial de la escena, y luego sumamos las contribuciones de energía para todas las superficies con el fin de obtener la cantidad de energía transferida entre las mismas. En la Figura 10.88, que ilustra la transferencia de energía radiante desde una superficie,  $dB$  es el flujo radiante visible que emana del punto de la superficie en la dirección dada por los ángulos  $\theta$  y  $\phi$  dentro de un ángulo sólido diferencial  $d\omega$  por unidad de tiempo, por unidad de área superficial.



**FIGURA 10.88.** Energía radiante visible emitida desde un punto de una superficie en la dirección  $(\theta, \phi)$  dentro del ángulo sólido  $d\omega$ .



**FIGURA 10.89.** Para un elemento unitario de superficie, el área proyectada perpendicular a la dirección de transferencia de energía es igual a  $\cos \phi$ .

La intensidad  $I$  para la radiación difusa en la dirección  $(\theta, \phi)$  puede describirse como la energía radiante por unidad de tiempo por unidad de área proyectada por unidad de ángulo sólido, o:

$$I = \frac{dB}{d\omega \cos\phi} \quad (10.91)$$

Suponiendo que la superficie sea un reflector difuso ideal (Sección 10.3), podemos definir la intensidad  $I$  como constante para todas las direcciones de visualización. Por tanto,  $dB/d\omega$  es proporcional al área de superficie proyectada (Figura 10.89). Para obtener la tasa total de radiación de energía desde un punto de una superficie, necesitamos sumar la radiación para todas las direcciones. Es decir, queremos la energía total que emana desde un hemisferio centrado en dicho punto de la superficie, como en la Figura 10.90, lo que nos da:

$$B = \int_{\text{hemi}} dB \quad (10.92)$$

Para un reflector difuso perfecto,  $I$  es constante, por lo que podemos expresar el flujo radiante  $B$  como:

$$B = I \int_{\text{hemi}} \cos\phi \, d\omega \quad (10.93)$$

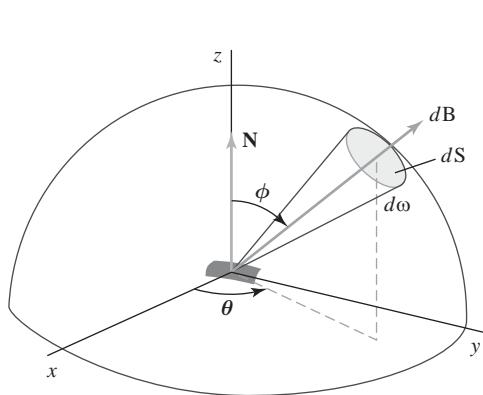
Asimismo, el elemento diferencial de ángulo sólido  $d\omega$  puede expresarse como (Apéndice A):

$$d\omega = \frac{dS}{r^2} = \sin\phi \, d\phi \, d\theta$$

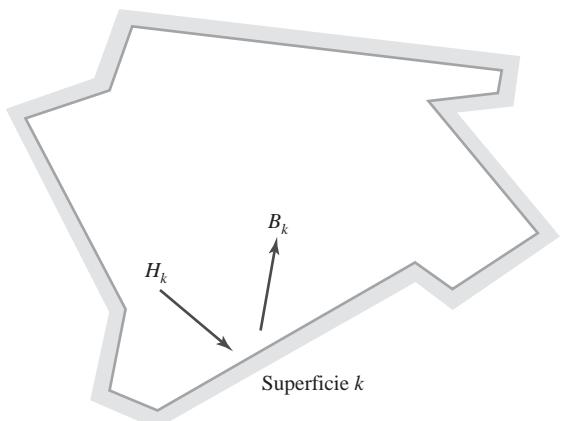
de modo que,

$$\begin{aligned} B &= I \int_0^{2\pi} \int_0^{\pi/2} \cos\phi \sin\phi \, d\phi \, d\theta \\ &= I\pi \end{aligned} \quad (10.94)$$

Podemos formar un modelo para las reflexiones luminosas de las distintas superficies estableciendo un «cierre» de las superficies (Figura 10.91). Cada superficie dentro del cierre es o un reflector, o un emisor



**FIGURA 10.90.** La energía radiante total para un punto de la superficie es la suma de las contribuciones en todas las direcciones de un hemisferio centrado en dicho punto de la superficie.



**FIGURA 10.91.** Un cierre de superficies para el modelo de radiosidad.

(fuente luminosa) o una combinación reflector-emisor. Designamos el parámetro de radiosidad  $B_k$  como la tasa total de energía radiante que abandona la superficie  $k$  por unidad de área. El parámetro de la energía incidente  $H_k$  es la suma de las contribuciones de energía radiante de todas las superficies del cierre que llegan a la superficie  $k$  por unidad de tiempo, por unidad de área. En otras palabras,

$$H_k = \sum_j B_j F_{jk} \quad (10.95)$$

donde el parámetro  $F_{jk}$  se denomina *factor de forma* para las superficies  $j$  y  $k$ . El factor de forma  $F_{jk}$  es la fracción de la energía radiante de la superficie  $j$  que alcanza a la superficie  $k$ .

Para una escena con  $n$  superficies dentro del cierre, la energía radiante de la superficie  $k$  se describe mediante la **ecuación de radiosidad**:

$$\begin{aligned} B_k &= E_k + \rho_k H_k \\ &= E_k + \rho_k \sum_{j=1}^n B_j F_{jk} \end{aligned} \quad (10.96)$$

Si la superficie  $k$  no es una fuente luminosa, entonces  $E_k = 0$ . En caso contrario,  $E_k$  será la tasa de energía emitida desde la superficie  $k$  por unidad de área (vatio/m<sup>2</sup>). El parámetro  $\rho_k$  es el factor de reflectividad para la superficie  $k$  (porcentaje de la luz incidente que es reflejado en todas direcciones). Este factor de reflectividad está relacionado con el coeficiente de reflexión difusa usado en los modelos empíricos de iluminación. Las superficies planas y convexas no pueden «verse» a sí mismas, por lo que no hay ninguna auto-incidencia y el factor de forma  $F_{kk}$  para estas superficies es 0.

Para obtener los efectos de iluminación sobre las diversas superficies contenidas dentro del cierre, necesitamos resolver el sistema de ecuaciones de radiosidad para las  $n$  superficies, dada la matriz de valores para  $E_k$ ,  $\rho_k$  y  $F_{jk}$ . En otras palabras, debemos resolver:

$$(1 - \rho_k F_{kk}) B_k - \rho_k \sum_{j \neq k} B_j F_{jk} = E_k \quad k = 1, 2, 3, \dots, n \quad (10.97)$$

o,

$$\begin{bmatrix} 1 - \rho_1 F_{11} & -\rho_1 F_{12} & \cdots & -\rho_1 F_{1n} \\ -\rho_2 F_{21} & 1 - \rho_2 F_{22} & \cdots & -\rho_2 F_{2n} \\ \vdots & \vdots & & \vdots \\ -\rho_n F_{n1} & -\rho_n F_{n2} & \cdots & 1 - \rho_n F_{nn} \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{bmatrix} = \begin{bmatrix} E_1 \\ E_2 \\ \vdots \\ E_n \end{bmatrix} \quad (10.98)$$

Después, convertimos los valores de intensidad  $I_k$  dividiendo los valores de radiosidad  $B_k$  por  $\pi$ . Para imágenes en color, podemos calcular los componentes RGB individuales de la radiosidad ( $B_{kR}$ ,  $B_{kG}$ ,  $B_{kB}$ ) utilizando las componentes de color para  $\rho_k$  y  $E_k$ .

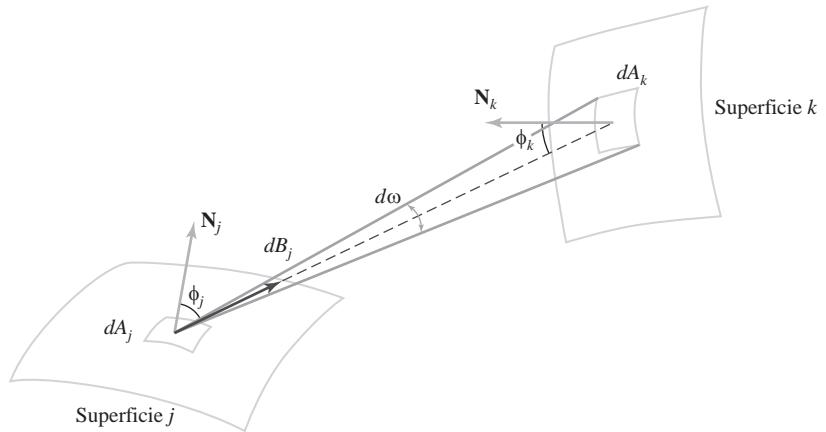
Antes de poder resolver la Ecuación 10.97, debemos determinar los valores de los factores de forma  $F_{jk}$ . Hacemos esto considerando la transferencia de energía desde la superficie  $j$  a la superficie  $k$  (Figura 10.92). La tasa de energía radiante que incide sobre un pequeño elemento superficial  $dA_k$  procedente del elemento de área  $dA_j$  es:

$$dB_j dA_j = (I_j \cos \phi_j d\omega) dA_j \quad (10.99)$$

Pero el ángulo sólido  $d\omega$  puede escribirse en términos de la proyección del elemento de área  $dA_k$  en perpendicular a la dirección  $dB_j$ , de la forma siguiente:

$$d\omega = \frac{dA}{r^2} = \frac{\cos \phi_k dA_k}{r^2} \quad (10.100)$$

Por tanto, podemos expresar la Ecuación 10.99 en la forma:



**FIGURA 10.92.** Transferencia de una cantidad diferencial de energía radiante  $dB_j$  desde un elemento de superficie con área  $dA_j$  hacia el elemento de superficie  $dA_k$ .

$$\frac{dB_j}{dA_j} = \frac{I_j \cos\phi_j \cos\phi_k dA_j dA_k}{r^2} \quad (10.101)$$

El factor de forma entre las dos superficies es el porcentaje de la energía que emana del área  $dA_j$  e incide en:

$$\begin{aligned} F_{dA_j, dA_k} &= \frac{\text{energía incidente en } dA_k}{\text{energía total que sale de } dA_j} \\ &= \frac{I_j \cos\phi_j \cos\phi_k dA_j dA_k}{r^2} \cdot \frac{1}{B_j dA_j} \end{aligned} \quad (10.102)$$

Asimismo,  $B_j = \pi I_j$ , de modo que:

$$F_{dA_j, dA_k} = \frac{\cos\phi_j \cos\phi_k dA_k}{\pi r^2} \quad (10.103)$$

y la fracción de la energía emitida desde el área  $dA_j$  y que incide en la superficie completa  $k$  es:

$$F_{dA_j, A_k} = \int_{\text{surf}_j} \frac{\cos\phi_j \cos\phi_k}{\pi r^2} dA_k \quad (10.104)$$

donde  $A_k$  es el área de la superficie  $k$ . Podemos entonces definir el factor de forma entre las dos superficies como el promedio de la expresión anterior para todo el área, que será:

$$F_{jk} = \frac{1}{A_j} \int_{\text{surf}_j} \int_{\text{surf}_k} \frac{\cos\phi_j \cos\phi_k}{\pi r^2} dA_k dA_j \quad (10.105)$$

Las dos integrales de la Ecuación 10.105 se evalúan utilizando técnicas de integración numérica y estipulando las siguientes condiciones:

$$\sum_{k=1}^n F_{jk} = 1, \quad \text{para todo } k \quad (\text{conservación de la energía})$$

$$A_j F_{jk} = A_k F_{kj} \quad (\text{reflexión uniforme de la luz})$$

$$F_{jj} = 0, \quad \text{para todo } j \quad (\text{suponiendo sólo parches superficiales planos o convexos})$$

Para aplicar el modelo de radiosidad, subdividimos cada superficie de una escena en muchos pequeños polígonos. La apariencia realista de la escena mostrada se mejora al reducir el tamaño de las subdivisiones poligonales, pero entonces hace falta más tiempo para representar la escena. Podemos acelerar el cálculo de los factores de forma utilizando un hemicubo para aproximar el hemisferio. Esto sustituye la superficie esférica por un conjunto de superficies (planas) lineales. Después de evaluados los factores de forma, podemos resolver el conjunto de ecuaciones lineales 10.97 utilizando una técnica numérica tal como la eliminación gaussiana o la descomposición LA (Apéndice A). Alternativamente, podríamos comenzar con valores aproximados para las  $B_j$  y resolver el sistema de ecuaciones lineales iterativamente utilizando el método de Gauss-Seidel. En cada iteración, calculamos una estimación de la radiosidad para el parche superficial  $k$  utilizando los valores de radiosidad previamente obtenidos en la ecuación de radiosidad:

$$B_k = E_k + \rho_k \sum_{j=1}^n B_j F_{jk}$$

Entonces podemos mostrar la escena en cada paso para observar la mejora en la representación de las superficies. Este proceso se repite hasta que los cambios en los valores de radiosidad calculados sean pequeños.

### Método de radiosidad mediante refinamiento progresivo

Aunque el método de radiosidad produce representaciones altamente realistas de las superficies, se necesita un tiempo de procesamiento considerable para calcular los factores de forma y además los requisitos de almacenamiento son muy altos. Utilizando la técnica de *refinamiento progresivo*, podemos reestructurar el algoritmo iterativo de radiosidad para acelerar los cálculos y reducir los requisitos de almacenamiento en cada iteración.

A partir de la ecuación de radiosidad, la transferencia de energía radiante entre dos parches de superficie se calcula como:

$$B_k \text{ debido a } B_j = \rho_k B_j F_{jk} \quad (10.106)$$

Recíprocamente,

$$B_j \text{ debido a } B_k = \rho_j B_k F_{kj}, \text{ para todo } j \quad (10.107)$$

que podemos reescribir como:

$$B_j \text{ debido a } B_k = \rho_j B_k F_{jk} \frac{A_j}{A_k}, \quad \text{para todo } j \quad (10.108)$$

Esta relación es la base para la técnica de refinamiento progresivo de los cálculos de radiosidad. Utilizando un único parche superficial  $k$ , podemos calcular todos los factores de forma  $F_{jk}$  y considerar la transferencia de luz desde dicho parche a todas las demás superficies del entorno. Con este procedimiento, sólo necesitamos calcular y almacenar los valores de los parámetros para un único hemicubo y los factores de forma asociados. En la siguiente iteración, sustituimos estos valores de parámetros por valores correspondientes a otro parche seleccionado. Y podemos mostrar las mejoras sucesivas en la representación de las superficies a medida que pasamos de un parche seleccionado a otro.

Inicialmente, hacemos  $B_k = E_k$  para todos los parches superficiales. Después, seleccionamos el parche con el mayor valor de radiosidad, que será el emisor de luz más brillante y calculamos la siguiente aproximación a la radiosidad para todos los demás parches. Este proceso se repite en cada paso, de modo que las fuentes luminosas se seleccionan primero, empezando por la de mayor energía radiante, y después se seleccionan los demás parches basándose en la cantidad de luz recibida de las fuentes luminosas. Los pasos en una técnica simple de refinamiento progresivo son los que se esbozan en el siguiente algoritmo:

```

for each patch k
    /* Definir hemicubo y calcular factores de forma F [j][k]. */

    for each patch j {
        dRad = rho [j] * B [k] * F [j][k] * A [j] / A [k];
        dB [j] = dB [j] + dRad;
        B [j] = B [j] + dRad;
    }

    dB [k] = 0;

```

En cada paso, se selecciona el parche superficial con el mayor valor de  $\Delta B_k A_k$ , dado que la radiosidad es una medida de la energía radiante por unidad de área. Asimismo, seleccionamos los valores iniciales como  $\Delta B_k = B_k = E_k$  para todos los parches superficiales. Este algoritmo de refinamiento progresivo aproxima la propagación real de la luz a través de una escena en función del tiempo.

La visualización de las superficies representadas en cada paso produce una secuencia de vistas que pasa de una escena oscura a otra completamente iluminada. Después del primer paso, las únicas superficies iluminadas son las fuentes luminosas y los parches no emisores que son visibles para el emisor seleccionado. Para producir vistas iniciales más útiles de la escena, podemos fijar un nivel de luz ambiente de modo que todos los parches tengan algo de iluminación. En cada etapa de la iteración, reducimos entonces la luz ambiente de acuerdo con el nivel de transferencia de energía radiante que haya en la escena.

La Figura 10.93 muestra una escena representada mediante el modelo de radiosidad con refinamiento progresivo. En las Figuras 10.94, 10.95 y 10.96 se ilustran diversas condiciones de iluminación en representaciones de escenas obtenidas con el método de radiosidad. A menudo suelen combinarse los métodos de trazado de rayos con el modelo de radiosidad con el fin de producir sombreados superficiales especulares y difusos altamente realistas, como en la Figura 10.87.



**FIGURA 10.93.** Nave de la catedral de Chartres representada mediante un modelo de radiosidad con refinamiento progresivo. La representación obtenida por John Wallace y John Lin, utilizando el paquete de trazado de rayos y radiosidad Starbase de Hewlett-Packard. Los factores de forma de radiosidad se calcularon mediante método de trazado de rayos. (Courtesy de Eric Haines, Autodesk, Inc. © 1989 Hewlett-Packard Co.)



**FIGURA 10.94.** Imagen de un museo constructivista representada mediante un método de radiosidad de refinamiento progresivo. (Cortesía de Shenchang Eric Chen, Stuart I. Feldman y Julie Dorsey, *Program of Computer Graphics, Cornell University*. © 1988 Cornell University Program of Computer Graphics.)



**FIGURA 10.95.** Simulación de la escalera de la torre en el Engineering Theory Center Building de la Universidad de Cornell, representado mediante un método de radiosidad de refinamiento progresivo. (Cortesía de Keith Howie y Ben Trumbore, *Program of Computer Graphics, Cornell University*. © 1990 Cornell University Program of Computer Graphics.)



(a)



(b)

**FIGURA 10.96.** Simulación de dos esquemas de iluminación para un decorado utilizado en la representación de *La Bohemia* en el Teatro Metropolitano de la Ópera. En (a) se incluye una vista diurna completamente iluminada, mientras que en (b) podemos ver una vista nocturna. (Cortesía de Julie Dorsey y Mark Shepard, *Program of Computer Graphics, Cornell University*. © 1991 Cornell University Program of Computer Graphics.)

## 10.13 MAPEADO DE ENTORNO

Un procedimiento alternativo para modelar reflexiones globales consiste en definir una matriz de valores de intensidad que describa el entorno situado alrededor de un objeto o de un grupo de objetos. En lugar de utilizar trazado de rayos entre los objetos o cálculos de radiosidad para determinar los efectos globales de iluminación difusa y especular, simplemente mapeamos la *matriz de entorno* sobre un objeto en relación con la dirección de visualización. Este procedimiento se denomina **mapeado de entorno** o en ocasiones **mapeado de reflexión** (aunque también podrían modelarse efectos de transparencia con el mapa de entorno). Se trata de una aproximación simple y rápida a las técnicas más precisas de representación basada en trazado de rayos que hemos expuesto en las Secciones 10.11 y 10.12.

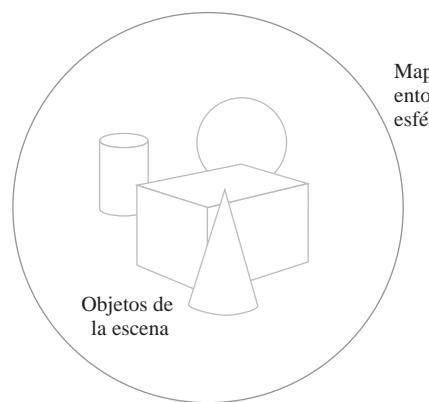
El mapa de entorno se define sobre las superficies de un universo circundante. La información del mapa de entorno incluye los valores de intensidad para las fuentes luminosas, el cielo y otros objetos de fondo. La Figura 10.97 muestra el universo circundante como una esfera, pero a menudo se utiliza un cubo o un cilindro para definir las superficies de entorno que rodean a los objetos de una escena.

Para obtener la imagen de la superficie de un objeto, proyectamos áreas de píxel sobre la superficie del objeto y luego reflejamos cada una de esas áreas proyectadas sobre el mapa de entorno con el fin de obtener los valores de intensidad superficial para el píxel. Si el objeto es transparente, también podemos refractar el área de píxel proyectada hacia el mapa de entorno. El proceso de mapeado de entorno para la reflexión de un área de píxel proyectada se ilustra en la Figura 10.98. La intensidad del píxel se determina promediando los valores de intensidad dentro de la región intersectada del mapa de entorno.

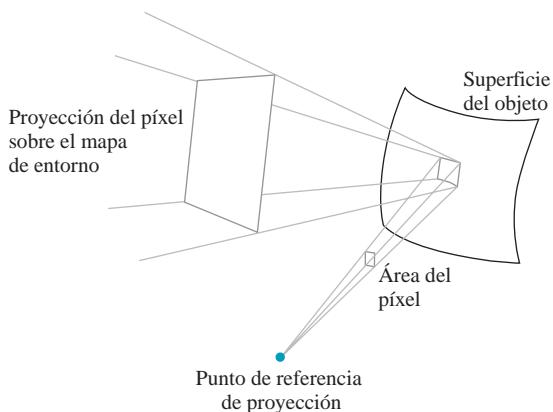
## 10.14 MAPEADO DE FOTONES

Aunque el método de radiosidad puede producir imágenes precisas con efectos de iluminación globales para las escenas simples, este método se hace más difícil de aplicar a medida que se incrementa la complejidad de una escena. Tanto el tiempo de representación como los requisitos de almacenamiento se hacen prohibitivos para las escenas muy complicadas, y muchos efectos de iluminación son difíciles de modelar correctamente. El **mapeado de fotones** proporciona un método general eficiente y preciso para modelar la iluminación global en las escenas complejas.

El concepto básico del mapeado de fotones consiste en separar la información de iluminación de la geometría de una escena. Se trazan trayectorias de rayos a través de la escena desde todas las fuentes luminosas y la



**FIGURA 10.97.** Universo esférico con el mapa de entorno sobre la superficie de la esfera.



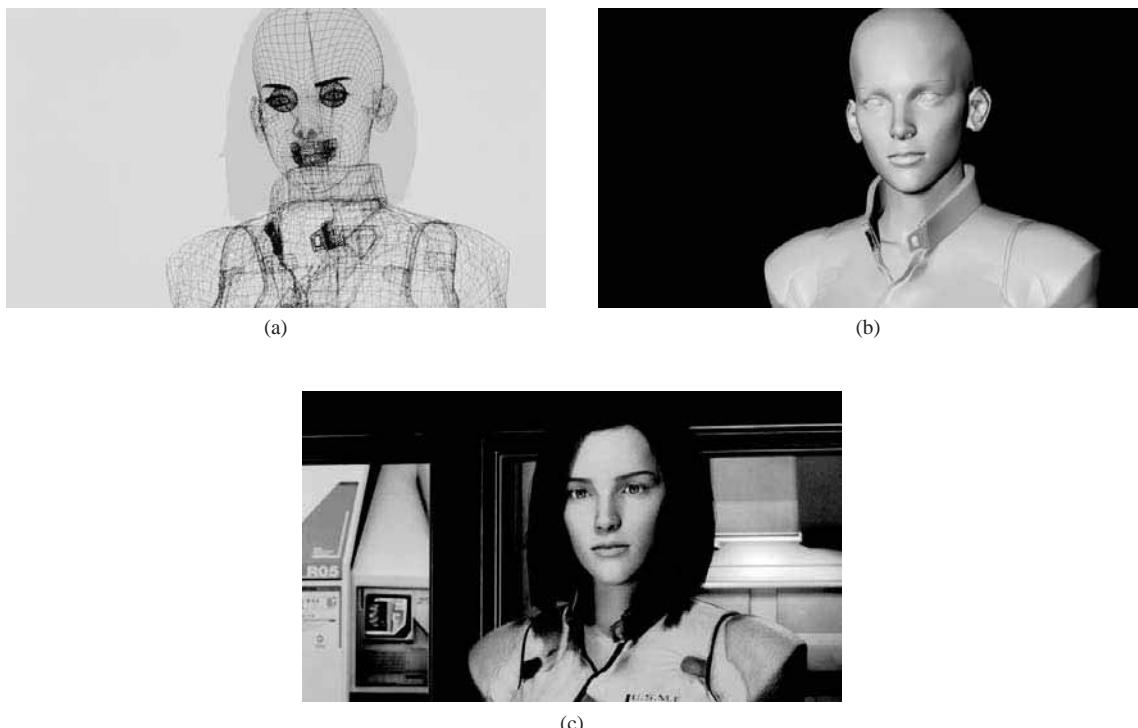
**FIGURA 10.98.** Proyección de un área de píxel sobre una superficie y reflexión del área sobre el mapa de entorno.

información de iluminación correspondiente a las intersecciones entre los rayos y los objetos se almacena en un **mapa de fotones**. Entonces, se aplican métodos de trazado de rayos distribuido utilizando algoritmos incrementales similares a los que se emplean en las representaciones mediante radiosidad.

Las fuentes luminosas pueden ser puntuales, focos direccionales o de cualquier otro tipo. La intensidad asignada a una fuente lumínosa se divide entre sus rayos (fotones) y las direcciones de los rayos se distribuyen aleatoriamente. Una fuente lumínosa puntual se modela generando trayectorias de rayos uniformemente en todas direcciones, a menos que la fuente sea direccional (Sección 10.1). Para otras fuentes luminosas, se seleccionan posiciones aleatorias en la fuente y se generan rayos en direcciones aleatorias. Para las luces brillantes se generan más rayos que para las fuentes luminosas de baja potencia. Además, pueden construirse para las fuentes luminosas *mapas de proyección* que almacenen información binaria sobre si hay o no objetos en cualquier región del espacio. También pueden utilizarse esferas de contorno dentro del algoritmo para proporcionar información acerca de los objetos contenidos en grandes regiones del espacio. Para una escena puede generarse cualquier número de rayos y la precisión de los efectos de iluminación se incrementa a medida que se generan más trayectorias de rayos.

## 10.15 ADICIÓN DE DETALLES A LAS SUPERFICIES

Hasta ahora hemos expuesto las técnicas de representación para la visualización de superficies suaves de los objetos. Sin embargo, la mayoría de los objetos no tienen superficies suaves y homogéneas. Necesitamos las texturas superficiales para modelar de manera precisa objetos tales como paredes de ladrillo, carreteras de grava, alfombras, madera o piel humana. Además, algunas superficies contienen patrones que es preciso tener



**FIGURA 10.99.** Etapas de modelado y representación en el desarrollo del personaje animado Dr. Aki Ross para la película *Final Fantasy: The Spirits Within*: (a) modelo alámbrico de Aki, (b) estructura superficial de la piel y la ropa y (c) figura final representada, incluyendo el pelo y los detalles relativos a la ropa y a las características de la piel. (Cortesía de Square Pictures, Inc. © 2001 FFFP. Todos los derechos reservados.)

en cuenta durante los procedimientos de representación. La superficie de una vasija podría mostrar un diseño pintado, un vaso de agua puede tener grabado el escudo de la familia, una pista de tenis contiene marcas que indican las distintas zonas en que se divide el campo y una autopista de cuatro carriles tiene una serie de líneas divisorias y otras marcas, como huellas de frenazo o manchas de aceite.

La Figura 10.99 ilustra las capas básicas en el modelado y representación de un objeto al que hay que añadir detalles superficiales. Primero puede utilizarse una imagen alámbrica del objeto para ajustar el diseño global. A continuación, se encajan las capas superficiales sobre el contorno del objeto para producir una vista de la estructura con una superficie suave. Después, se añaden los detalles de la superficie a las capas exteriores. Para el ejemplo de la Figura 10.99, los detalles de la superficie incluyen los patrones de la chaqueta (como los pliegues y la textura de la tela) y las características de la piel, como poros o pecas. En la Figura 10.100 se muestran imágenes ampliadas de las características de la piel de este personaje, que han sido generadas por computadora, mientras que la Figura 10.101 muestra las características simuladas de la piel para una persona mayor. En la Figura 10.102 se proporcionan ejemplos adicionales de escenas representadas con detalles superficiales.

Podemos añadir detalles a las superficies utilizando diversos métodos, incluyendo:

- Pegar pequeños objetos, como flores o espinas sobre una superficie mayor.
- Modelar patrones superficiales mediante pequeñas áreas de polígonos.
- Mapear matrices de texturas o procedimientos de modificación de la intensidad sobre una superficie.
- Modificar el vector normal a la superficie para crear relieves localizados.
- Modificar tanto el vector normal a la superficie como el vector tangente a la superficie para mostrar patrones direccionales sobre madera y otros materiales.



(a)



(b)

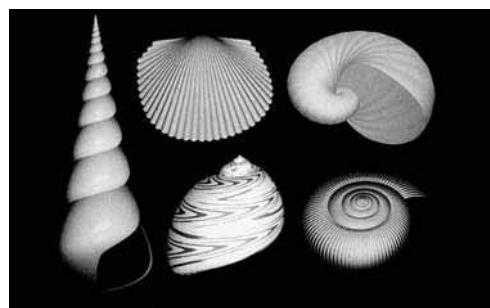
**FIGURA 10.100.** Detalles de la piel para el personaje animado Dr. Aki Ross en la película *Final Fantasy: The Spirits Within*. (Cortesía de Square Pictures, Inc. © 2001 FFFP. Todos los derechos reservados.)



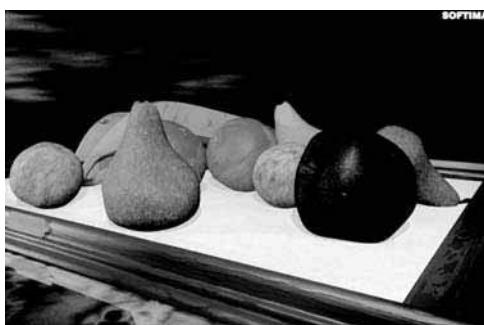
**FIGURA 10.101.** Características faciales y textura superficial de la piel para el personaje animado Dr. Sid, que representa a un hombre de 70 años de edad en la película *Final Fantasy: The Spirits Within*. (Cortesía de Square Pictures, Inc. © 2001 FFFP. Todos los derechos reservados.)



(a)



(b)



(c)



(d)

**FIGURA 10.102.** Escenas que ilustran la generación infográfica de detalles superficiales para diversos objetos: (a) plantas de cactus a las que se les han añadido espinas y flores (cortesía de Deborah R. Fowler, Przemyslaw Prusinkiewicz y Johannes Battjes, University of Calgary. © 1992.), (b) conchas marinas con diversos patrones y superficies en relieve (cortesía de Deborah R. Fowler, Hans Meinhardt y Przemyslaw Prusinkiewicz, University of Calgary. © 1992.), (c) una tabla de frutas (cortesía de SOFTIMAGE, Inc.) y (d) patrones superficiales para piezas de ajedrez y para un tablero de ajedrez, generados con métodos de mapeado de texturas (cortesía de SOFTIMAGE, Inc.).

## 10.16 MODELADO DE LOS DETALLES SUPERFICIALES MEDIANTE POLÍGONOS

---

Un método simple para añadir detalles a una superficie consiste en modelar patrones u otras características superficiales utilizando caras poligonales. Para detalles a gran escala, el modelado mediante polígonos puede proporcionar buenos resultados. Algunos ejemplos de dichos detalles a gran escala serían los cuadrados en un tablero de ajedrez, las líneas divisorias en una autopista, los patrones de las baldosas en un suelo de linóleo, los diseños florales en una alfombra, los paneles de una puerta o los anuncios en el lateral de un camión. También podríamos modelar una superficie irregular con pequeñas caras poligonales aleatoriamente orientadas, siempre y cuando las caras no sean excesivamente pequeñas.

Los polígonos para la aplicación de patrones superficiales se suelen solapar sobre una superficie poligonal mayor y se procesan junto con esa superficie padre. El algoritmo de detección de superficies visibles sólo procesa al polígono padre, pero los parámetros de iluminación de los polígonos de detalle superficial tienen preferencia sobre los del polígono padre. Cuando haya que modelar detalles superficiales intrincados o muy precisos, los métodos basados en polígonos no son prácticos. Por ejemplo, sería difícil modelar con precisión la estructura superficial de uva pasa utilizando caras poligonales.

## 10.17 MAPEADO DE TEXTURAS

---

Un método común para añadir detalles a un objeto consiste en mapear patrones sobre la descripción geométrica del objeto. El patrón de texturas puede definirse mediante una matriz de valores de color o mediante un procedimiento que modifique los colores del objeto. Este método para incorporar detalles de los objetos a una escena se denomina **mapeado de texturas** o **mapeo de patrones** y las texturas pueden definirse como patrones unidimensionales, bidimensionales o tridimensionales. Cualquier especificación de textura se denomina **espacio de textura**, que se referencia mediante **coordenadas de textura** comprendidas en el rango que va de 0 a 1.0.

Las funciones de textura en un paquete gráfico permiten a menudo especificar como opción el número de componentes de color para cada posición de un patrón. Por ejemplo, cada especificación de color en un patrón de textura podría estar compuesta por cuatro componentes RGBA, tres componentes RGB, un único valor de intensidad para un tono de azul, un índice a una tabla de colores o un único valor de luminancia (una media ponderada de las componentes RGB de un color). Cada componente de la descripción de una textura se denomina frecuentemente «texel», pero existe una cierta confusión en el uso de este término. Algunas veces, una posición de un espacio de texturas correspondiente a un conjunto de componentes de color, como por ejemplo una tripla RGB, se denomina texel, mientras que en otras ocasiones se denomina texel a un único elemento de la matriz de texturas, como por ejemplo el valor de la componente roja de un color RGB.

### Patrones de textura lineales

Puede especificarse un patrón de textura unidimensional mediante una matriz de valores de color con un sólo subíndice, definiendo esta matriz una secuencia de colores en un espacio de texturas lineal. Por ejemplo, podríamos definir una lista de 32 colores RGB a la que hiciéramos referencia mediante unos valores de subíndice que fueran de 0 a 95. Los primeros tres elementos de la matriz almacenan las componentes RGB del primer color, los siguientes tres elementos almacenan las componentes RGB del segundo color, etc. Este conjunto de colores, o cualquier subconjunto contiguo de los colores, podría entonces usarse para formar una línea con un cierto patrón a través de un polígono, una banda alrededor de un cilindro o un patrón de color para mostrar un segmento de línea aislado.

Para un patrón lineal, el espacio de texturas se referencia mediante un único valor de la coordenada  $s$ . Para las especificaciones de color RGB, el valor  $s = 0.0$  designa el primer color RGB de la matriz (formado por

tres elementos), el valor  $s = 1.0$  designa las últimas tres componentes de color de RGB y el valor  $s = 0.5$  referencia los tres elementos intermedios de color RGB de la matriz. Por ejemplo, si el nombre de la matriz de textura es `colorArray`, entonces el valor  $s = 0.0$  hace referencia a los tres valores de la matriz `colorArray[0], colorArray[1]` y `colorArray[2]`.

Para mapear un patrón de textura lineal sobre una escena, asignamos un valor de la coordenada  $s$  a una posición espacial y otro valor de la coordenada  $s$  a una segunda posición espacial. La sección de la matriz de colores correspondiente al rayo especificado de coordenadas  $s$  se utiliza entonces para generar una línea multicolor entre las dos posiciones espaciales. Un procedimiento de mapeado de texturas utiliza normalmente una función lineal para calcular las posiciones de la matriz que hay que asignar a los píxeles a lo largo de un segmento lineal. Cuando el número de colores de textura especificados para la línea es pequeño, puede asignarse cada color a un gran conjunto de píxeles, dependiendo de la longitud de la línea. Por ejemplo, si el rango especificado de la coordenada  $s$  abarca un único color RGB (tres elementos de color RGB) dentro de la matriz de texturas, todos los píxeles de la línea se mostrarán con dicho color. Pero si hay que mapear múltiples colores a las distintas posiciones a lo largo de la línea, entonces se asignarán menos píxeles a cada color. Asimismo, puesto que algunos píxeles podrían mapearse a posiciones de la matriz que estuvieran situadas entre sucesivos colores RGB, se pueden usar diversos esquemas para determinar el color que hay que asignar a cada píxel. Un método simple de mapeado de colores consiste en asignar a cada píxel el color de la matriz más próximo. Alternativamente, si un píxel se mapea a una posición que esté comprendida entre los elementos iniciales de la matriz correspondientes a dos colores sucesivos, el color del píxel puede calcularse como una combinación lineal de los dos elementos de color más próximos dentro de la matriz.

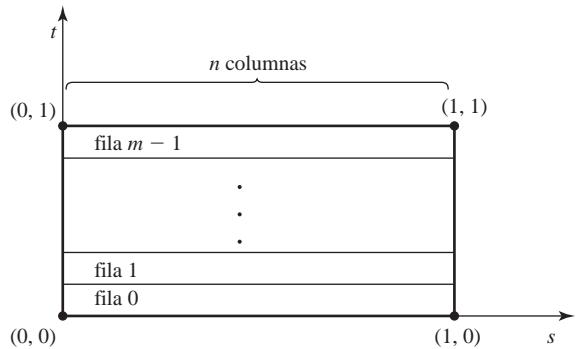
Algunos procedimientos de mapeado de texturas permiten utilizar valores para las coordenadas de texturas que estén fuera del rango de 0 a 1.0. Estas situaciones pueden surgir cuando queramos mapear múltiples copias de una textura sobre un mismo objeto o cuando los valores de  $s$  calculados puedan estar fuera del intervalo unitario. Si queremos permitir valores de las coordenadas de texturas que caigan fuera del rango que va de 0 a 1.0, podemos simplemente ignorar la parte entera de cualquier valor  $s$  determinado. En este caso, el valor  $-3.6$ , por ejemplo, haría referencia a la misma posición dentro del espacio de texturas del valor 0.6 o el valor 12.6. Pero si no queremos permitir valores fuera del rango que va de 0 a 1.0, entonces podemos limitarnos a fijar los valores dentro de este intervalo unitario: cualquier valor calculado que sea inferior a 0 adoptará el valor 0 y a todos los valores calculados que sean superiores a 1.0 se les asignará el valor 1.0.

## Patrones de textura superficial

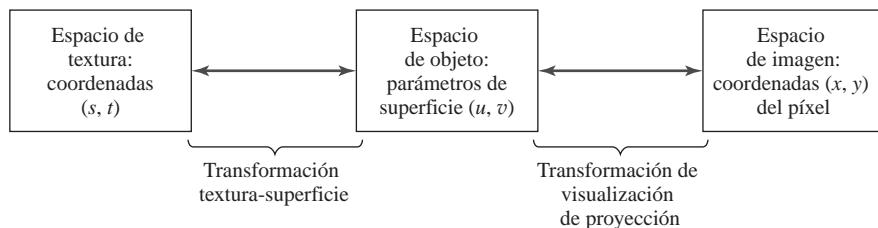
Una textura para un área superficial se define comúnmente mediante un patrón rectangular de color, y las posiciones dentro de este espacio de textura se refieren mediante valores de coordenadas bidimensionales ( $s, t$ ). Las especificaciones para cada color de patrón de texturas pueden almacenarse en una matriz con tres subíndices. Si se define un patrón de texturas con 16 por 16 colores RGB, por ejemplo, entonces la matriz correspondiente a este patrón contendrá  $16 \times 16 \times 3 = 768$  elementos.

La Figura 10.103 ilustra un espacio de textura bidimensional. Los valores correspondientes a  $s$  y  $t$  varían de 0 a 1.0. La primera fila de la matriz enumera los valores de color a lo largo de la parte inferior del patrón de texturas rectangular, mientras que la última fila de la matriz enumera los valores de color correspondientes a la parte superior del patrón. La posición de coordenadas  $(0, 0)$  en el espacio de texturas hace referencia al primer conjunto de componentes de color en la primera posición de la primera fila, mientras que la posición  $(1.0, 1.0)$  hace referencia al último conjunto de componentes de color en la última posición de la última fila de la matriz. Por supuesto, podríamos enumerar los colores de la matriz de texturas de otras formas; si enumeráramos los colores de arriba a abajo, el origen del espacio de textura bidimensional estaría en la esquina superior izquierda del patrón rectangular. Pero colocar el origen del espacio de texturas en la esquina inferior izquierda suele simplificar los procedimientos de mapeado sobre la referencia de coordenadas espaciales de una escena.

Especificamos un mapeado de textura superficial para un objeto utilizando los mismos procedimientos que para mapear una textura lineal sobre la escena. Pueden asignarse las coordenadas  $(s, t)$  del espacio de textu-



**FIGURA 10.103.** Coordenadas bidimensionales del espacio de texturas que hacen referencia a posiciones dentro de una matriz de valores de color que contienen  $m$  filas y  $n$  columnas. Cada posición de la matriz hace referencia a múltiples componentes de color.



**FIGURA 10.104.** Sistemas de coordenadas para el espacio de texturas bidimensional, el espacio de objetos y el espacio de imagen.

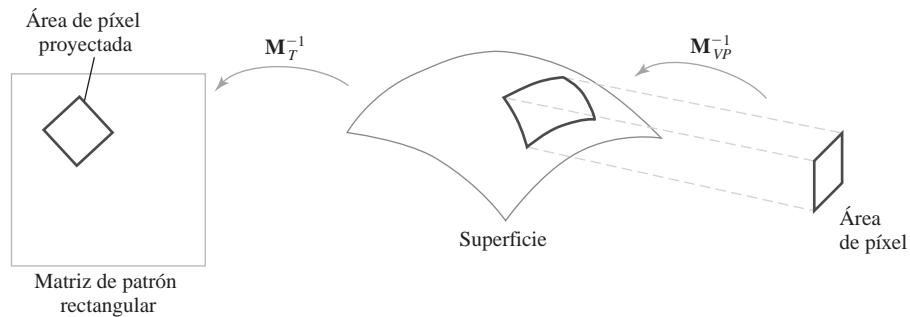
ras correspondientes a las cuatro esquinas de patrón de texturas (Figura 10.103) a cuatro posiciones en el espacio dentro de la escena, utilizándose una transformación lineal para asignar los valores de color a las posiciones de píxel proyectadas para el área espacial designada. También es posible realizar otros tipos de mapeados. Por ejemplo, podríamos asignar tres coordenadas del espacio de texturas a los vértices de un triángulo.

Las posiciones en la superficie de un objeto, como por ejemplo un parche de *spline* cúbica o una sección de esfera, pueden describirse mediante coordenadas *uv* en el espacio de objetos, y las posiciones de píxel proyectadas se refieren en coordenadas cartesianas *xy*. El mapeado de la textura superficial puede llevarse a cabo en una de estas dos formas: o bien mapeamos el patrón de textura sobre la superficie de un objeto y luego sobre el plano de proyección, o podemos mapear cada área de píxel sobre la superficie del objeto y luego mapear este área superficial sobre el espacio de texturas. El mapeado de un patrón de texturas sobre las coordenadas de píxel se denomina en ocasiones *escaneo de textura*, mientras que el mapeado de las coordenadas de píxel sobre el espacio de texturas se denomina *escaneo en orden de píxel*, *escaneo inverso* o *escaneo en orden de imagen*. La Figura 10.104 ilustra las dos posibles secuencias de transformación entre los tres espacios.

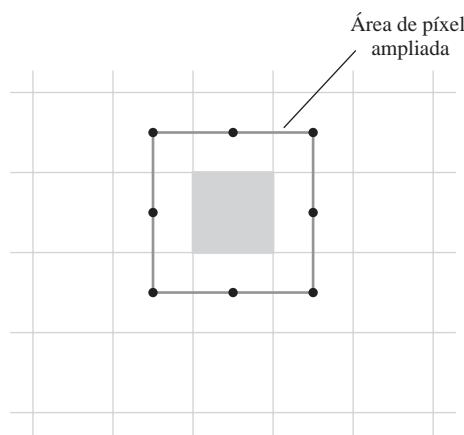
Las transformaciones lineales paramétricas proporcionan un método simple para mapear las posiciones del espacio de texturas sobre el espacio de objetos:

$$\begin{aligned} u &= u(s, t) = a_u s + b_u t + c_u \\ v &= v(s, t) = a_v s + b_v t + c_v \end{aligned} \tag{10.109}$$

La transformación desde el espacio de objetos hasta el espacio de imagen se lleva a cabo concatenando las transformaciones de visualización de proyección. Una desventaja del mapeado desde el espacio de texturas al espacio de píxeles es que un parche de textura seleccionado no suele corresponderse con las fronteras del píxel, lo que requiere efectuar una serie de cálculos para determinar el porcentaje de recubrimiento del píxel. Por tanto, el método de mapeado de texturas más comúnmente utilizado es el mapeado del espacio de píxeles al espacio de texturas (Figura 10.105). Esto evita los cálculos de subdivisión de píxel y permite aplicar fácilmente procedimientos de *antialiasing* (filtrado). Un procedimiento de *antialiasing* efectivo consiste en



**FIGURA 10.105.** Mapeado de texturas mediante proyección de áreas de píxel sobre el espacio de texturas.



**FIGURA 10.106.** Área ampliada para un píxel, que incluye las posiciones centrales de los píxeles adyacentes.

proyectar un área de píxel ligeramente mayor que incluya los centros de los píxeles vecinos, como se muestra en la Figura 10.106, y aplicar una función piramidal para ponderar los valores de intensidad del patrón de texturas. Pero el mapeado del espacio de imagen al espacio de texturas requiere calcular la inversa de la transformación de visualización-proyección  $M_{VP}^{-1}$  y la inversa de la transformación del mapa de texturas  $M_T^{-1}$ . En el siguiente ejemplo, vamos a ilustrar este técnica mapeando un patrón definido sobre una superficie cilíndrica.

### Ejemplo 10.1 Mapeado de una textura superficial

Para ilustrar los pasos del mapeado de textura superficial, vamos a considerar la transferencia del patrón mostrado en la Figura 10.107(a) a una superficie cilíndrica. Los parámetros de superficie son las coordenadas cilíndricas:

$$u = \theta, \quad v = z$$

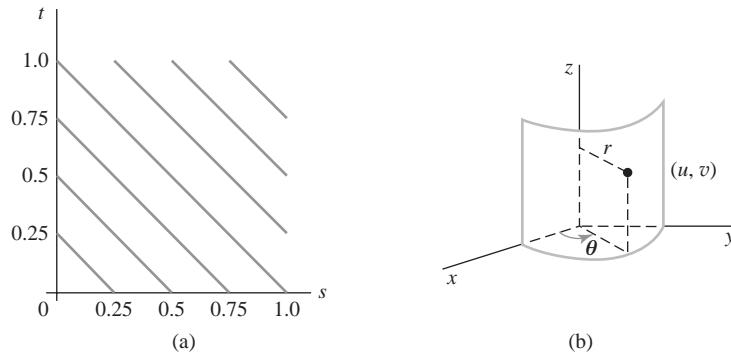
con,

$$0 \leq \theta \leq \pi/2, \quad 0 \leq z \leq 1$$

Y la representación paramétrica para la superficie en el sistema de referencia cartesiano es:

$$x = r \cos u, \quad y = r \sin u, \quad z = v$$

Podemos mapear el patrón matricial sobre la superficie utilizando la siguiente transformación lineal, que transforma las coordenadas del espacio de textura  $(s, t) = (0, 0)$  en la esquina inferior izquierda del elemento de superficie  $(x, y, z) = (r, 0, 0)$ .



**FIGURA 10.107.** Mapeado de un patrón de texturas definido dentro de un cuadrado unitario (a) sobre una superficie cilíndrica (b).

$$u = s\pi/2, \quad v = t$$

A continuación, seleccionamos una posición de visualización y aplicamos la transformación de visualización inversa desde las coordenadas de píxel hasta el sistema de referencia cartesiano de la superficie cilíndrica. Después, las coordenadas de la superficie cartesiana se transfieren a los parámetros de superficie  $uv$  mediante los cálculos

$$u = \tan^{-1}(y/x), \quad v = z$$

y las posiciones de píxel proyectada se mapean sobre el espacio de texturas mediante la transformación inversa

$$s = 2u/\pi, \quad t = v$$

Entonces se promedian los valores de color en la matriz del patrón cubierta por cada área de píxel proyectada con el fin de obtener el color del píxel.

## Patrones de textura volumétricos

Además de los patrones lineales y superficiales, podemos diseñar un conjunto de colores para una serie de posiciones en una región tridimensional del espacio. Estas texturas se denominan a menudo **patrones de textura volumétricos** o **texturas sólidas**. Podemos hacer referencia a una textura sólida utilizando coordenadas tridimensionales para el espacio de texturas ( $s, t, r$ ). Y el espacio de textura tridimensional se define dentro de un cubo unitario, estando las coordenadas de textura comprendidas entre 0 y 1.0.

Un patrón de textura volumétrico puede almacenarse en una matriz de cuatro subíndices, en la que los primeros tres subíndices denotan la posición de una fila, la posición de una columna y una posición de profundidad. El cuarto subíndice se utiliza para hacer referencia a una componente de un color concreto del patrón. Por ejemplo, un patrón de textura sólida RGB con 16 filas, 16 columnas y 16 planos de profundidad podría almacenarse en una matriz con  $16 \times 16 \times 16 \times 3 = 12.288$  elementos.

Para mapear el espacio de texturas completo sobre un bloque tridimensional, asignamos las coordenadas de las ocho esquinas del espacio de texturas a ocho posiciones del espacio de una escena. O bien, podemos mapear una sección plana del espacio de texturas, como por ejemplo un plano de profundidad o una cara del cubo de texturas, sobre un área plana de la escena. Existen muchas otras posibles formas de mapear una textura sólida.

Las texturas sólidas permiten obtener vistas internas, como por ejemplo secciones transversales, para objetos tridimensionales que haya que mostrar con patrones de textura. Así, pueden aplicarse a los ladrillos o a los



**FIGURA 10.108.** Una escena en la que se han modelado las características de los objetos utilizando métodos de texturas sólidas. (Cortesía de Peter Shirley, Computer Science Department, University of Utah.)

objetos de madera unos mismos patrones de textura en toda su extensión espacial. La Figura 10.108 muestra una escena visualizada mediante texturas sólidas para obtener patrones de madera veteada y otros tipos de texturas.

### Patrones de reducción de texturas

En animación y otras aplicaciones, el tamaño de los objetos suelen cambiar a menudo. Para objetos mostrados con patrones de textura, necesitamos entonces aplicar los procedimientos de mapeado de texturas a las dimensiones modificadas del objeto. Cuando el tamaño de un objeto texturado se reduce, el patrón de textura se aplica a una región más pequeña y esto puede hacer que aparezcan dispersiones en las texturas. Para evitar éstos, podemos crear un conjunto de **patrones de reducción de texturas** que se deberán utilizar cuando el tamaño visualizado de los objetos se reduzca.

Normalmente, cada patrón de reducción tiene la mitad del tamaño del patrón anterior. Por ejemplo, si tenemos un patrón bidimensional 16 por 16, podemos definir cuatro patrones adicionales con los tamaños reducidos 8 por 8, 4 por 4, 2 por 2 y 1 por 1. Para cualquier vista de un objeto, podemos entonces aplicar el patrón de reducción apropiado con el fin de minimizar las distorsiones. Estos patrones de reducción se suelen denominar **mapas MIP o mip maps**, donde el término *mip* es un acrónimo de la frase latina *multum in parvo*, que podría traducirse como «mucho en un pequeño objeto».

### Métodos de texturado procedural

Otro método para añadir un patrón de texturas a un objeto consiste en utilizar una definición procedural para las variaciones de color que hay que aplicar. Esta técnica evita los cálculos de transformación implicados en el mapeado de patrones matriciales sobre las descripciones de los objetos. Además, el texturado procedural elimina los requisitos de almacenamiento necesarios cuando hay que aplicar muchos patrones de textura de gran tamaño, y en especial texturas sólidas, a una escena.

Generamos una textura procedural calculando variaciones para las propiedades o características de un objeto. Por ejemplo, las vetas de madera o del mármol pueden crearse para un objeto utilizando funciones



**FIGURA 10.109.** Una escena representada con VG Shaders y modelada con RenderMan utilizando caras poligonales para las facetas de las gemas, superficies cuádricas y parches bicubicos. Además de matrices de texturas, se utilizaron métodos procedimentales para crear la atmósfera vaporosa de la jungla y la cubierta vegetal, que muestra un característico efecto de iluminación. (Cortesía de the VALIS Group. Reimpreso de Graphics Gems III, editado por David Kirk. © 1992 Academic Press, Inc.)

armónicas (curvas sinusoidales) definidos en una región del espacio tridimensional. Entonces, se superponen perturbaciones aleatorias a las variaciones armónicas con el fin de descomponer los patrones simétricos. La escena de la Figura 10.109 fue representada utilizando descripciones procedimentales para patrones que son típicos de las superficies de las piedras, del oro pulido y de las hojas de plátano.

## 10.18 MAPEADO DE RELIEVE

Aunque las matrices de texturas pueden utilizarse para añadir detalles de carácter fino a una superficie, usualmente no son efectivas para modelar la apariencia rugosa de las superficies de algunos objetos tales como naranjas, fresas o pasas. Los detalles relativos a la intensidad luminosa que se proporcionan en una matriz de texturas para este tipo de objetos están definidos de forma independiente de los parámetros de iluminación, como por ejemplo la dirección de la fuente luminosa. Un método mejor para modelar la rugosidad de las superficies consiste en aplicar una función de perturbación a la normal a la superficie y luego usar el vector normal perturbado en los cálculos realizados dentro del modelo de iluminación. Esta técnica se denomina **mapeado de relieve (bump mapping)**.

Si  $\mathbf{P}(u, v)$  representa una posición sobre una superficie paramétrica, podemos obtener la normal a la superficie en dicho punto mediante el cálculo:

$$\mathbf{N}' = \mathbf{P}'_u \times \mathbf{P}'_v \quad (10.110)$$

donde  $\mathbf{P}_u$  y  $\mathbf{P}_v$  son las derivadas parciales de  $\mathbf{P}$  con respecto a los parámetros  $u$  y  $v$ . Para aplicar variaciones a la normal a la superficie, podemos modificar el vector de posición de la superficie sumándole una pequeña función de perturbación, denominada *función de relieve*:

$$\mathbf{P}'(u, v) = \mathbf{P}(u, v) + b(u, v) \mathbf{n} \quad (10.111)$$

Esto añade relieves a la superficie en la dirección del vector unitario normal a la superficie  $\mathbf{n} = \mathbf{N}/|\mathbf{N}|$ . Entonces, la normal a la superficie perturbada se obtiene de la manera siguiente:

$$\mathbf{N}' = \mathbf{P}'_u \times \mathbf{P}'_v \quad (10.112)$$

La derivada parcial de  $\mathbf{P}'$  con respecto a  $u$  es:

$$\begin{aligned} \mathbf{P}'_u &= \frac{\partial}{\partial u}(\mathbf{P} + b\mathbf{n}) \\ &= \mathbf{P}_u + b_u \mathbf{n} + b \mathbf{n}_u \end{aligned} \quad (10.113)$$

Suponiendo que la magnitud de la función de relieve  $b$  sea pequeña, podemos ignorar el último término de la expresión anterior, de modo que:

$$\mathbf{P}'_u \approx \mathbf{P}_u + b_u \mathbf{n} \quad (10.114)$$

De forma similar,

$$\mathbf{P}'_v \approx \mathbf{P}_v + b_v \mathbf{n} \quad (10.115)$$

Y la normal a la superficie perturbada será:



(a)



(b)

**FIGURA 10.110.** Representación del aspecto característico de las superficies rugosas mediante mapeado de relieve. (Cortesía de (a) Peter Shirley, Computer Science Department, Universidad de Utah y (b) SOFTIMAGE, Inc.)



**FIGURA 10.111.** El caballero de cristal de la película *El joven Sherlock Holmes*. Se utilizó una combinación de mapeado de relieve, mapeado de entorno y mapeado de texturas para representar la superficie de la armadura. (Cortesía de Industrial Light & Magic. © 1985 Paramount Pictures/Amblin.)

$$\mathbf{N}' = \mathbf{P}_u \times \mathbf{P}_v + b_v(\mathbf{P}_u \times \mathbf{n}) + b_u(\mathbf{n} \times \mathbf{P}_v) + b_u b_v(\mathbf{n} \times \mathbf{n})$$

Pero  $\mathbf{n} \times \mathbf{n} = 0$ , de modo,

$$\mathbf{N}' = \mathbf{N} + b_v(\mathbf{P}_u \times \mathbf{n}) + b_u(\mathbf{n} \times \mathbf{P}_v) \quad (10.116)$$

El paso final consiste en normalizar  $\mathbf{N}'$  para utilizarla en los cálculos del modelo de iluminación.

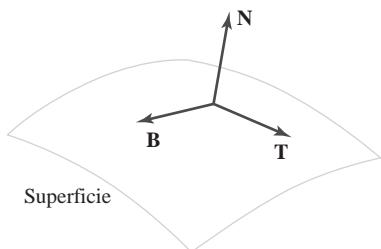
Hay muchas formas en las que podemos especificar la función de relieve  $b(u, v)$ . Podemos definir una expresión analítica, pero los cálculos se reducen si simplemente obtenemos los valores de relieve mediante tablas de sustitución. Con una tabla de relieve, los valores de  $b$  pueden determinarse rápidamente utilizando interpolación lineal y cálculos incrementales. Entonces, las derivadas parciales  $b_u$  y  $b_v$  se aproximan mediante diferencias finitas. La tabla de relieve puede construirse con patrones aleatorios, con patrones de cuadrícula regulares o con formas de caracteres. Los controles aleatorios resultan útiles para modelar una superficie regular, como la de una pasa, mientras que un patrón repetitivo puede utilizarse para modelar la superficie de una naranja, por ejemplo. Para aplicar mecanismos de *antialiasing*, suprimimos las áreas de píxel y promediamos las intensidades calculadas de los subpíxeles.

La Figura 10.110 muestra ejemplos de superficies representadas con mapeado de relieve. En la Figura 10.111 se proporciona un ejemplo de métodos combinados de representación superficial. La armadura del caballero de cristal de la película *El joven Sherlock Holmes* fue representado mediante una combinación de mapeado de relieve, mapeado de entorno y mapeado de texturas. Un mapa de entorno de la escena circundante fue combinado con un mapa de relieve para producir las reflexiones de iluminación de fondo y la apariencia rugosa de la superficie. Después, se añadieron colores adicionales, iluminación superficial, relieves, manchas de suciedad y pliegues para obtener el efecto global mostrado en esta figura.

## 10.19 MAPEADO DEL SISTEMA DE REFERENCIA

Este método para añadir detalle a las superficies es una extensión del mapeado de relieve. En el **mapeado del sistema de referencia (frame mapping)**, perturbamos tanto el vector normal a la superficie  $\mathbf{N}$  como un sistema de coordenadas local (Figura 10.112) asociado a  $\mathbf{N}$ . Las coordenadas locales se definen mediante un vector tangente a la superficie  $\mathbf{T}$  y un vector binormal  $\mathbf{B} = \mathbf{T} \times \mathbf{N}$ .

El mapeado del sistema de referencia se utiliza para modelar superficies anisótropas. Orientamos  $\mathbf{T}$  a lo largo de la «veta» de la superficie y aplicamos perturbaciones direccionales, además de las perturbaciones de



**FIGURA 10.112.** Un sistema de coordenadas local en una posición de la superficie.

relieve en la dirección **N**. De esta forma, podemos modelar patrones de veta de madera, patrones de cruce de las hebras en los tejidos y vetas en el mármol u otros materiales similares. Tanto las perturbaciones de relieve como las direccionales pueden generarse utilizando tablas de sustitución.

## 10.20 FUNCIONES OpenGL DE ILUMINACIÓN Y REPRESENTACIÓN DE SUPERFICIES

---

En OpenGL hay disponibles diversas rutinas para definir fuentes de luz puntuales, para seleccionar los coeficientes de reflexión superficiales y para elegir valores para otros parámetros del modelo básico de iluminación. Además, podemos simular la transparencia y podemos mostrar los objetos utilizando representación plana de las superficies o representación de Gouraud.

### Función OpenGL para fuentes luminosas puntuales

Podemos incluir múltiples fuentes luminosas puntuales en la descripción OpenGL de una escena, teniendo asociadas cada fuente luminosa diversas propiedades, como su posición, tipo, color, atenuación y efectos de foco direccional. Podemos establecer un valor de una propiedad para una fuente luminosa con la función:

```
glLight* (lightName, lightProperty, propertyValue);
```

Se añade un código sufijo igual a **i** o **f** al nombre de la función, dependiendo del tipo de dato al que pertenezca el valor de la propiedad. Para datos vectoriales, también se añade el código de sufijo **v** y entonces el parámetro **propertyValue** será un puntero a una matriz. Puede hacerse referencia a cada fuente luminosa mediante un identificador, y al parámetro **lightName** se le asigna uno de los identificadores simbólicos OpenGL **GL\_LIGHT0**, **GL\_LIGHT1**, **GL\_LIGHT2**, . . . , **GL\_LIGHT7**, aunque algunas implementaciones de OpenGL pueden permitir más de ocho fuentes luminosas. De forma similar, al parámetro **lightProperty** hay que asignarle una de las diez constantes simbólicas de propiedad que admite OpenGL. Después de haber asignado todas las propiedades a una fuente luminosa, podemos activarla mediante el comando:

```
 glEnable (lightName);
```

Sin embargo, también necesitamos activar las rutinas de iluminación OpenGL, lo que hacemos mediante la función:

```
 glEnable (GL_LIGHTING);
```

Entonces las superficies de los objetos se representan utilizando cálculos de iluminación que incluirán las contribuciones de cada una de las fuentes luminosas que hayan sido activadas.

### Especificación de la posición y el tipo de una fuente luminosa en OpenGL

La constante simbólica de propiedad OpenGL para designar la posición de una fuente luminosa es **GL\_POSITION**. En realidad, esta constante simbólica se utiliza para definir dos propiedades de las fuentes luminosas

al mismo tiempo: la posición de la fuente luminosa y el *tipo de la fuente luminosa*. Hay disponibles en OpenGL dos clasificaciones generales de las fuentes luminosas utilizadas para iluminar una escena. Una fuente luminosa puntual puede clasificarse como próxima a los objetos que hay que iluminar (una fuente local) o puede tratarse como si estuviera infinitamente alejada de una escena. Esta clasificación es independiente de la posición que asignemos a la fuente luminosa. Para una fuente luminosa próxima, la luz emitida radia en todas direcciones y la posición de la fuente luminosa se incluye en los cálculos de iluminación. Pero la luz emitida por una fuente distante sólo puede emanar en una dirección y esta dirección se aplica a todas las superficies de la escena, independientemente de la posición que hayamos asignado a la fuente luminosa. La dirección de los rayos emitidos desde una fuente clasificada como distante se calcula como la dirección desde la posición asignada de la línea que une la posición asignada a la fuente luminosa con el origen de coordenadas.

Se utiliza un vector de coma flotante de cuatro elementos para designar tanto el tipo de la fuente lumínosa como los valores de coordenadas que definen su posición. Los primeros tres elementos de este vector proporcionan la posición en coordenadas universales, mientras que el cuarto elemento se utiliza para designar el tipo de la fuente lumínosa. Si asignamos el valor 0.0 al cuarto elemento del vector de posición, la luz se considera como una fuente muy distante (lo que se denomina en OpenGL una luz «direccional») y la posición de la fuente lumínosa se utilizará entonces únicamente para determinar la dirección de los rayos de luz. En caso contrario, se asume que la fuente lumínosa es una fuente puntual local (lo que se denomina en OpenGL una luz «posicional») y la posición de la luz es utilizada por las rutinas de iluminación con el fin de determinar la dirección de los rayos luminosos que inciden sobre cada objeto de la escena. En el siguiente ejemplo de código, la fuente lumínosa 1 está definida como una fuente local en la ubicación (2.0, 0.0, 3.0), mientras que la fuente lumínosa 2 es una fuente distante que emite los rayos luminosos en la dirección y negativa:

```
GLfloat light1PosType [ ] = { 2.0, 0.0, 3.0, 1.0 };
GLfloat light2PosType [ ] = { 0.0, 1.0, 0.0, 0.0 };

glLightfv (GL_LIGHT1, GL_POSITION, light1PosType);
 glEnable (GL_LIGHT1);

glLightfv (GL_LIGHT2, GL_POSITION, light2PosType);
 glEnable (GL_LIGHT2);
```

Si no especificamos una posición y un tipo para una fuente lumínosa, los valores predeterminados son (0.0, 0.0, 1.0, 0.0), lo que indica una fuente distante cuyos rayos luminosos viajan en la dirección z negativa.

La posición de una fuente lumínosa está incluida en la descripción de la escena y se transforma a coordenadas de visualización junto con las posiciones de los objetos; esta transformación se lleva a cabo mediante las matrices de transformación geométrica y transformación de visualización de OpenGL. Por tanto, si queremos conservar una fuente lumínosa en una posición fija relativa a los objetos de una escena, debemos definir su posición después de especificar las transformaciones geométricas y de visualización del programa. Pero si queremos que la fuente lumínosa se mueva a medida que se mueve el punto de vista, definiremos su posición antes de especificar la transformación de visualización y podemos aplicar una traslación o rotación a una fuente lumínosa con el fin de moverla alrededor de una escena estacionaria.

## Especificación de los colores de las fuentes luminosas en OpenGL

A diferencia de una fuente lumínosa real, una fuente en OpenGL tiene tres diferentes propiedades de color RGBA. En este esquema empírico, los tres colores de la fuente lumínosa proporcionan opciones para variar los efectos de iluminación de una escena. Configuraremos estos colores utilizando las constantes simbólicas de propiedad del color `GL_AMBIENT`, `GL_DIFFUSE` y `GL_SPECULAR`. A cada uno de estos colores se le asigna un conjunto de cuatro valores de coma flotante. Las componentes de cada color se especifican en el orden (R, G, B, A) y la componente alpha se utiliza sólo si están activadas las rutinas de mezcla de color. Como cabe suponer a partir de los nombres de las constantes simbólicas de propiedad del color, uno de los colores de la fuente lumínosa contribuye a la luz de fondo (ambiente) de la escena, otro color se utiliza en los cálculos de

iluminación difusa y el tercer color se emplea para calcular los efectos de iluminación especular para las superficies. En realidad, las fuentes luminosas tienen sólo un color, pero podemos utilizar los tres colores de fuente luminosa en OpenGL para crear distintos efectos de iluminación. En el siguiente ejemplo de código, asignamos el color negro como color ambiente para una fuente luminosa local, denominada `GL_LIGHT3`, y asignamos el color blanco a los colores difuso y especular.

```
GLfloat blackColor [ ] = {0.0, 0.0, 0.0, 1.0};
GLfloat whiteColor [ ] = {1.0, 1.0, 1.0, 1.0};

glLightfv (GL_LIGHT3, GL_AMBIENT, blackColor);
glLightfv (GL_LIGHT3, GL_DIFFUSE, whiteColor);
glLightfv (GL_LIGHT3, GL_SPECULAR, whiteColor);
```

Los colores predeterminados para la fuente luminosa 0 son negro para el color ambiente y blanco para los colores difuso y especular. Todas las demás fuentes luminosas tienen como color predeterminado el negro para las propiedades de color ambiente, difuso y especular.

### Especificación de coeficientes de atenuación radial de la intensidad para una fuente luminosa OpenGL

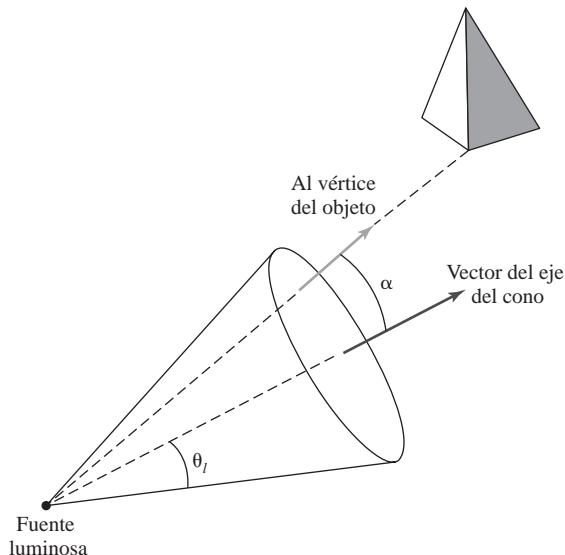
Podemos aplicar una atenuación radial de la intensidad a la luz emitida desde una fuente luminosa local OpenGL, y las rutinas de iluminación OpenGL calcularán esta atenuación utilizando la Ecuación 10.2, siendo  $d_l$  la distancia desde la posición de la fuente luminosa hasta la posición de un objeto. Las tres constantes de propiedad OpenGL para la atenuación radial de la intensidad son `GL_CONSTANT_ATTENUATION`, `GL_LINEAR_ATTENUATION` y `GL_QUADRATIC_ATTENUATION`, que se corresponden con los coeficientes  $a_0$ ,  $a_1$  y  $a_2$  de la Ecuación 10.2. Puede utilizarse un valor entero positivo o un valor positivo de coma flotante para definir cada uno de los coeficientes de atenuación. Por ejemplo, podríamos asignar los valores de los coeficientes de atenuación radial de la forma siguiente:

```
glLightf (GL_LIGHT6, GL_CONSTANT_ATTENUATION, 1.5);
glLightf (GL_LIGHT6, GL_LINEAR_ATTENUATION, 0.75);
glLightf (GL_LIGHT6, GL_QUADRATIC_ATTENUATION, 0.4);
```

Una vez especificados los valores de los coeficientes de atenuación, la función de atenuación radial se aplica a los tres colores (ambiente, difuso y especular) de la fuente luminosa. Los valores predeterminados de los coeficientes de atenuación son  $a_0 = 1.0$ ,  $a_1 = 0.0$  y  $a_2 = 0.0$ . Así, la opción predeterminada es que no haya atenuación radial:  $f_{l,radatten} = 1.0$ . Aunque la atenuación radial puede producir imágenes más realistas, los cálculos consumen mucho tiempo.

### Fuentes luminosas direccionales en OpenGL (focos)

Para fuentes luminosas locales (aquellas que no se considera que están en el infinito), podemos también especificar un efecto direccional o de foco. Esto limita la luz emitida de la fuente a una región del espacio con forma de cono. Definimos la región cónica mediante un vector de dirección según el eje del cono mediante una apertura angular  $\theta_f$  con respecto al eje del cono, como se muestra en la Figura 10.113. Además, podemos especificar un exponente angular de atenuación  $a_f$  para la fuente luminosa, que determinará cuánto decrece la intensidad de la luz a medida que nos alejamos desde el centro del cono hacia la superficie del mismo. A lo largo de cualquier dirección dentro del cono de luz, el factor de atenuación angular es  $\cos^{a_f} \alpha$  (Ecuación 10.5), donde  $\cos \alpha$  se calcula como el producto escalar del vector del eje del cono y del vector que une la fuente luminosa con la posición de un objeto. Calculamos el valor para cada uno de los colores ambiente, difuso y especular en un ángulo  $\alpha$  multiplicando las componentes de intensidad por este factor de atenuación angular. Si  $\alpha > \theta_f$ , el objeto estará fuera del cono de la fuente luminosa y no será iluminado por ésta. Para los rayos de luz que se encuentran dentro del cono, también podemos atenuar radialmente los valores de intensidad.



**FIGURA 10.113.** Un cono circular de luz emitido por una fuente luminosa OpenGL. La extensión angular del cono de luz, medido desde el eje del cono, es  $\theta_l$  y el ángulo desde el eje al vector de dirección de un objeto se designa mediante  $\alpha$ .

Hay tres constantes de propiedad OpenGL para los efectos direccionales: `GL_SPOT_DIRECTION`, `GL_SPOT_CUTOFF` y `GL_SPOT_EXPONENT`. Especificamos la dirección de la luz como un vector en coordenadas universales enteras o de coma flotante. El ángulo del cono  $\theta_l$  se especifica como un valor entero o de coma flotante en grados, y este ángulo puede ser  $180^\circ$  o cualquier valor en el rango que va de  $0^\circ$  a  $90^\circ$ . Cuando el ángulo del cono se hace igual a  $180^\circ$ , la fuente luminosa emite rayos en todas direcciones ( $360^\circ$ ). Podemos definir el valor del exponente de la atenuación de intensidad como un número entero o en coma flotante en el rango comprendido entre 0 y 128. Las siguientes inscripciones especifican los efectos direccionalles para la fuente luminosa número 3 de modo que el eje del cono se encuentra en la dirección  $x$ , el ángulo del cono  $\theta_l$  es  $30^\circ$  y el exponente de atenuación es 2.5.

```
GLfloat dirVector [ ] = {1.0, 0.0, 0.0};

glLightfv (GL_LIGHT3, GL_SPOT_DIRECTION, dirVector);
glLightf (GL_LIGHT3, GL_SPOT_CUTOFF, 30.0);
glLightf (GL_LIGHT3, GL_SPOT_EXPONENT, 2.5);
```

Si no especificamos una dirección para la fuente luminosa, la dirección predeterminada será paralela al eje  $z$  negativo, es decir,  $(0.0, 0.0, -1.0)$ . Asimismo, el ángulo predeterminado del cono será  $180^\circ$  y el exponente de atenuación predeterminado será 0. Así, la opción predeterminada es una fuente luminosa puntual que irradia en todas direcciones, sin ninguna atenuación angular.

### Parámetros de iluminación globales en OpenGL

Pueden especificarse diversos parámetros de iluminación en OpenGL a nivel global. Estos valores se utilizan para controlar la forma en que se llevan a cabo determinados cálculos de iluminación, y un valor de parámetro global se especifica mediante la siguiente función:

```
glLightModel* (paramName, paramValue);
```

Agregamos un código de sufijo igual a `i` o `f`, dependiendo del tipo de dato del valor del parámetro. Y para los datos vectoriales, también agregamos el código de sufijo `v`. Al parámetro `paramName` se le asigna una constante simbólica OpenGL que identifica la propiedad global que hay que configurar, mientras que al parámetro `paramValue` se le asigna un único valor o un conjunto de valores. Utilizando la función `glLightModel`, podemos definir un nivel global de luz ambiente, podemos especificar cómo hay que calcu-

lar los resaltes especulares y podemos decidir si debe aplicarse el modelo de iluminación a las caras posteriores de las superficies poligonales.

Además de un color ambiente para las fuentes de luz individuales, podemos especificar un valor independiente para la iluminación de fondo de OpenGL en forma de parámetro global. Esto proporciona una opción adicional para los cálculos empíricos de iluminación. Para establecer esta opción, utilizamos una constante simbólica `GL_LIGHT_MODEL_AMBIENT`. La siguiente instrucción, por ejemplo, establece la iluminación de fondo general para una escena, asignándola un color azul de baja intensidad (oscuro), con un valor de alpha igual a 1.0:

```
globalAmbient [ ] = { 0.0, 0.0, 0.3, 1.0 );
glLightModelfv ( GL_LIGHT_MODEL_AMBIENT, globalAmbient );
```

Si no especificamos un nivel global de luz ambiente, la opción predeterminada es un color blanco de baja intensidad (gris oscuro), que equivale al valor (0.2, 0.2, 0.2, 1.0).

Los cálculos de las reflexiones especulares requieren determinar diversos vectores, incluyendo el vector  $\mathbf{V}$  que une una posición de una superficie con una posición de visualización. Para acelerar los cálculos de reflexión specular, las rutinas de iluminación de OpenGL pueden utilizar una dirección constante para el vector  $\mathbf{V}$ , independientemente de la posición de la superficie en relación con el punto de visualización. Este vector unitario constante está en la dirección  $z$  positiva, (0.0, 0.0, 1.0), y es el valor predeterminado para  $\mathbf{V}$ , pero si queremos desactivar este valor predeterminado y calcular  $\mathbf{V}$  utilizando la posición de visualización real, que es el origen del sistemas de coordenadas de visualización, podemos utilizar el siguiente comando:

```
glLightModeli ( GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE );
```

Aunque los cálculos de reflexión specular requieren más tiempo cuando utilizamos la posición real de visualización para calcular  $\mathbf{V}$ , lo cierto es que se obtienen imágenes más realistas. Podemos desactivar los cálculos superficiales para el vector  $\mathbf{V}$  utilizando el valor predeterminado `GL_FALSE` (o 0, o 0.0) para el parámetro del observador local.

Cuando se añaden texturas superficiales a los cálculos de iluminación OpenGL, los resaltes de las superficies pueden atenuarse y los patrones de textura pueden verse distorsionados por los términos especulares. Por tanto, como opción, los patrones de textura pueden aplicarse únicamente a los términos no especulares que contribuyen al color de una superficie. Estos términos no especulares incluyen los efectos de la luz ambiente, las emisiones superficiales y las reflexiones difusas. Utilizando esta opción, las rutinas de iluminación OpenGL generan dos colores para cada cálculo de iluminación superficial: un color specular y las contribuciones de color no especulares. Los patrones de textura se combinan únicamente con el color no specular, después de lo cual se combinan los dos colores. Para seleccionar esta opción de dos colores se utiliza la instrucción:

```
glLightModeli ( GL_LIGHT_MODEL_COLOR_CONTROL,
                GL_SEPARATE_SPECULAR_COLOR );
```

No es necesario separar los términos de color si no estamos usando patrones de texturas, y los cálculos de iluminación se realizan de manera más eficiente si no se activa esta opción. El valor predeterminado para esta propiedad es `GL_SINGLE_COLOR`, que no separa el color specular de las otras componentes de color de la superficie.

En algunas aplicaciones, puede que convenga mostrar las superficies posteriores de un objeto. Un ejemplo sería la vista interior en corte transversal de un sólido, en la cual habrá que mostrar algunas superficies posteriores, además de las superficies frontales. Sin embargo, de manera predeterminada, los cálculos de iluminación utilizan las propiedades asignadas de los materiales únicamente para las caras frontales. Para aplicar los cálculos de iluminación tanto a las caras frontales como a las posteriores, utilizando las correspondientes propiedades de los materiales de las caras frontal y posterior, utilizamos el comando:

```
glLightModeli ( GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE );
```

Los vectores normales a la superficie de las caras posteriores serán entonces invertidos y se aplicarán los cálculos de iluminación utilizando las propiedades de los materiales que se hayan asignado a las caras posteriores. Para desactivar los cálculos de iluminación en los dos lados, utilizamos el valor `GL_FALSE` (o 0, o 0.0) en la función `glLightModel`, valor que es el que se usa de manera predeterminada.

## Función OpenGL de propiedad de una superficie

Los coeficientes de reflexión y otras propiedades ópticas de las superficies se configuran utilizando la función:

```
glMaterial* (surfFace, surfProperty, propertyValue);
```

A la función se le añade un código de sufijo `i` o `f`, dependiendo del tipo de dato del valor de la propiedad, y también se agrega el código `v` cuando se suministran propiedades que toman como valor un vector. Al parámetro `surfFace` se le asigna una de las constantes simbólicas `GL_FRONT`, `GL_BACK` o `GL_FRONT_AND_BACK`; el parámetro `surfProperty` es una constante simbólica que identifica un parámetro de la superficie tal como  $I_{\text{surf}}$ ,  $k_a$ ,  $k_d$ ,  $k_s$  o  $n_s$ ; y el parámetro `propertyValue` hay que configurarlo con el correspondiente valor. Todas las propiedades, exceptuando el componente de reflexión especular  $n_s$ , se especifican como valores vectoriales. Para establecer todas las propiedades de iluminación de un objeto utilizamos una secuencia de funciones `glMaterial`, antes de ejecutar los comandos que describen la geometría del objeto.

El valor RGBA para el color de emisión de la superficie,  $I_{\text{surf}}$  se selecciona utilizando la constante simbólica OpenGL de propiedad de la superficie `GL_EMISSION`. Como ejemplo, la siguiente instrucción establece el color de emisión para las superficies frontales asignándole un color gris claro:

```
surfEmissionColor [ ] = {0.8, 0.8, 0.8, 1.0};
glMaterialfv (GL_FRONT, GL_EMISSION, surfEmissionColor);
```

El color predeterminado de emisión de una superficie es el negro, (0.0, 0.0, 0.0, 1.0). Aunque puede asignarse un color de emisión a una superficie, esta emisión no ilumina a otros objetos de la escena. Para hacer eso, debemos definir la superficie como una fuente luminosa utilizando los métodos explicados en la Sección 10.3.

Se utilizan los nombres simbólicos de propiedad OpenGL `GL_AMBIENT`, `GL_DIFFUSE` y `GL_SPECULAR` para asignar valores a los coeficientes de reflexión de la superficie. En el mundo real, los coeficientes ambiente y difuso deberían tener asignado el mismo valor vectorial, y podemos hacer eso utilizando la constante simbólica `GL_AMBIENT_AND_DIFFUSE`. Los valores predeterminados para el coeficiente de ambiente son (0.2, 0.2, 0.2, 1.0), los valores predeterminados para el coeficiente difuso son (0.8, 0.8, 0.8, 1.0) y los valores predeterminados para el coeficiente especular son (1.0, 1.0, 1.0, 1.0). Para definir el exponente de reflexión especular, utilizamos la constante `GL_SHININESS`. Podemos asignar a esta propiedad cualquier valor en el rango comprendido entre 0 y 128 y el valor predeterminado es 0. Como ejemplo, las siguientes instrucciones establecen los valores de los tres coeficientes de reflexión y para el exponente especular. Los coeficientes difuso y de ambiente se configuran de modo que la superficie se muestre con un color azul claro al ser iluminada con luz blanca; la reflexión especular es del color de la luz incidente y el exponente especular tiene asignado un valor de 25.0.

```
diffuseCoeff [ ] = {0.2, 0.4, 0.9, 1.0};
specularCoeff [ ] = {1.0, 1.0, 1.0, 1.0};

glMaterialfv (GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE,
diffuseCoeff);
glMaterialfv (GL_FRONT_AND_BACK, GL_SPECULAR, specularCoeff);
glMaterialf (GL_FRONT_AND_BACK, GL_SHININESS, 25.0);
```

Las componentes de los coeficientes de reflexión también pueden definirse utilizando valores de una tabla de color, para lo cual se proporciona la constante simbólica OpenGL `GL_COLOR_INDEXES`. Los índices de la

tabla de color se asignan como una matriz de tres elementos enteros o de coma flotante, y el valor predeterminado es (0, 1, 1).

## Modelo de iluminación OpenGL

OpenGL calcula los efectos de iluminación superficiales utilizando el modelo básico de iluminación 10.19, con algunas variaciones sobre la forma de especificar ciertos parámetros. El nivel de luz ambiente es la suma de las componentes de ambiente de las fuentes luminosas y del valor global de luz ambiente. Los cálculos de reflexión difusa utilizan las componentes de intensidad difusa de las fuentes luminosas y los cálculos de reflexión specular utilizan la componente de intensidad specular de cada fuente.

Asimismo, el vector unitario  $v$ , especifica la dirección desde una posición de la superficie hasta una posición de visualización, y se le puede asignar el valor constante (0,0,0,0) si no se utiliza la opción de observador local. Para una fuente luminosa situada en el “infinito”, el vector unitario de dirección de la luz  $L$  está en la dirección opuesta a la que se haya asignado a los rayos de luz procedentes de dicha fuente.

## Efectos atmosféricos en OpenGL

Después de haber aplicado el modelo de iluminación con el fin de obtener los colores superficiales, podemos asignar un color a la atmósfera de una escena y combinar los colores superficiales con dicho color de atmósfera. También podemos usar una función de atenuación de la intensidad atmosférica con el fin de simular la visualización de la escena a través de una atmósfera neblinosa o llena de humo. Los diversos parámetros atmosféricos se configuran utilizando la función `glFog` que hemos presentado en la Sección 9.14:

```
 glEnable (GL_FOG);
 glFog* (atmoParameter, paramValue);
```

Se añade un código de sufijo igual a `i` o `f` para indicar el tipo del valor de datos, y con los datos vectoriales se utiliza el código de sufijo `v`.

Para definir un color de atmósfera, asignamos la constante simbólica OpenGL `GL_FOG_COLOR` al parámetro `atmoParameter`. Por ejemplo, podemos hacer que la atmósfera tenga un color gris azulado mediante la instrucción:

```
 GLfloat atmoColor [4] = {0.8, 0.8, 1.0, 1.0};
 glFogfv (GL_FOG_COLOR, atmoColor);
```

El valor predeterminado para el color de la atmósfera es el negro, (0,0,0,0,0,0).

A continuación, podemos elegir la función de atenuación atmosférica que haya que utilizar para combinar los colores de los objetos con el color de la atmósfera. Esto se lleva a cabo utilizando la constante simbólica `GL_FOG_MODE`:

```
 glFogi (GL_FOG_MODE, atmoAttenFunc);
```

Si asignamos al parámetro `atmoAttenFunc` el valor `GL_EXP`, se usará la Ecuación 10.312 como función de atenuación atmosférica. Con el valor `GL_EXP2`, se selecciona la Ecuación 10.32 como función de atenuación de atmósfera. Para cualquiera de ambas funciones exponenciales, podemos seleccionar un valor de densidad de la atmósfera mediante:

```
 glFog (GL_FOG_DENSITY, atmoDensity);
```

Una tercera opción para atenuaciones atmosféricas es la función lineal 9.13 de variación de la intensidad con la profundidad. En este caso, asignamos al parámetro `atmoAttenFunc` el valor `GL_LINEAR`. El valor predeterminado para el parámetro `atmoAttenFunc` es `GL_EXP`.

Una vez seleccionada una función de atenuación atmosférica, esta función se utiliza para calcular la mezcla del color de la atmósfera y del color de la superficie del objeto. OpenGL utiliza la Ecuación 10.33 en sus rutinas atmosféricas para calcular este color de mezcla.

## Funciones de transparencia OpenGL

En OpenGL pueden simularse algunos efectos de transparencia utilizando las rutinas de mezcla de color descritas en la Sección 4.3. Sin embargo, la implementación de la transparencia en un programa OpenGL no suele ser sencilla. Podemos combinar los colores de los objetos para una escena simple que contenga unas pocas superficies opacas y transparentes utilizando el valor de mezcla alpha para especificar el grado de transparencia y procesando las superficies según su orden de profundidad. Pero las operaciones de mezcla de color OpenGL ignoran los efectos de refracción, y el manejo de superficies transparentes en escenas complejas con una diversidad de condiciones de iluminación o con animaciones puede resultar muy complicado. Asimismo, OpenGL no proporciona ninguna funcionalidad para simular la apariencia superficial de un objeto translúcido (como por ejemplo un cristal esmerilado), que dispersa de manera difusa la luz transmitida a través del material semitransparente. Por tanto, para mostrar superficies translúcidas o los efectos de iluminación resultantes de la refracción, necesitamos describir nuestras propias rutinas. Para simular los efectos de iluminación a través de un objeto translúcido, podemos utilizar una combinación de valores de textura superficial y de propiedades del material. Para los efectos de refracción, podemos desplazar las posiciones de píxel para las superficies que se encuentren detrás de un objeto transparente, utilizando la Ecuación 10.29 para calcular el desplazamiento necesario.

Podemos designar como transparentes los objetos de una escena utilizando el parámetro alpha en los comandos de color RGBA de superficie de OpenGL, tal como `glMaterial` y `glColor`. Puede asignarse al parámetro alpha de una superficie el valor del coeficiente de transparencia (Ecuación 10.30) de dicho objeto. Por ejemplo, si especificamos el color de una superficie transparente mediante la función:

```
glColor4f (R, G, B, A);
```

entonces, asignaremos al parámetro alpha el valor  $A = k_t$ . A una superficie completamente transparente se le asignaría el valor alpha  $A = 1.0$  y a una superficie opaca, el valor alpha  $A = 0.0$ .

Una vez asignados los valores de transparencia, activamos las características de mezcla de color de OpenGL y procesamos las superficies, comenzando con los objetos más distantes y siguiendo por orden hasta los objetos más cercanos a la posición de visualización. Con la mezcla de color activada, cada color superficial se combina con los de las superficies solapadas que ya se encuentran en el búfer de imagen, utilizando los valores alpha asignados a la superficie.

Configuramos los factores de mezcla de color de modo que todas las componentes de color de la superficie actual (el objeto «fuente») se multiplican por  $(1 - A) = (1 - k_t)$ , mientras que todas las componentes de color de las posiciones correspondientes del búfer de imagen (el «destino») se multiplican por el factor  $A = k_t$ :

```
 glEnable (GL_BLEND);
 glBlendFunc (GL_ONE_MINUS_SRC_ALPHA, GL_SRC_ALPHA);
```

Los dos colores se mezclan entonces utilizando la Ecuación 10.30, teniendo el parámetro alpha el valor  $k_t$ , siendo los colores del búfer de imagen los correspondientes a una superficie que está detrás del objeto transparente que se esté procesando. Por ejemplo, si  $A = 0.3$ , entonces el nuevo color del búfer de imagen será la suma del 30 por ciento del color actual del búfer de imagen y el 70 por ciento del color de reflexión del objeto, para cada posición de la superficie. (Alternativamente, podríamos utilizar el parámetro de color alpha como factor de opacidad, en lugar de como factor de transparencia. Sin embargo, si asignamos a  $A$  un valor de opacidad, deberemos intercambiar también los dos argumentos en la función `glBlendFunc`.)

Las comprobaciones de visibilidad pueden llevarse a cabo utilizando las funciones de búfer de profundidad de OpenGL de la Sección 9.14. A medida que se procesa cada superficie visible opaca, se almacenan tanto los colores de la superficie como la profundidad de la misma. Pero cuando procesamos una superficie visible transparente, lo único que guardamos son sus colores, ya que la superficie no oculta a las superficies de fondo. Por tanto, cuando procesamos una superficie transparente, ponemos el búfer de profundidad en estado de sólo lectura utilizando la función `glDepthMask`.

Si procesamos todos los objetos en orden de profundidad, el modo de escritura del búfer de profundidad se desactiva y activa a medida que procesamos cada superficie transparente. Alternativamente, podríamos separar las dos clases de objeto, como en el siguiente fragmento de código:

```
glEnable (GL_DEPTH_TEST);
/* Procesar todas las superficies opacas. */
glEnable (GL_BLEND);
glDepthMask (GL_FALSE);
glBlendFunc (GL_ONE_MINUS_SRC_ALPHA, GL_SRC_ALPHA);
/* Procesar todas las superficies transparentes. */
glDepthMask (GL_TRUE);
glDisable (GL_BLEND);
glutSwapBuffers ( );
```

Si no se procesan los objetos transparentes en orden estricto de profundidad, comenzando por los más alejados, esta técnica no acumulará los colores superficiales con precisión en todos los casos. Pero para las escenas simples se trata de un método rápido y efectivo para generar una representación aproximada de los efectos de transparencia.

## Funciones de representación superficial OpenGL

Las superficies pueden mostrarse con las rutinas OpenGL utilizando técnicas de representación superficial de intensidad constante o mecanismos de representación superficial de Gouraud. No se proporcionan rutinas OpenGL para aplicar los mecanismos de representación superficial de Phong, los mecanismos de trazado de rayos ni los métodos de radiosidad. El método de representación se selecciona mediante:

```
glShadeModel (surfRenderingMethod);
```

La representación de superficies con intensidad constante se deselecciona asignando el valor simbólico `GL_FLAT` al parámetro `surfRenderingMethod`. Para el sombreado de Gouraud (la opción predeterminada), utilizamos la constante simbólica `GL_SMOOTH`.

Cuando se aplica la función `glShadeModel` a una superficie curva teselada, como por ejemplo una esfera que esté aproximada mediante una malla poligonal, las rutinas de representación OpenGL utilizan los vectores normales a la superficie en los vértices del polígono para calcular el color del polígono. Las componentes cartesianas de un vector normal a la superficie en OpenGL se especifica mediante el comando:

```
glNormal3* (Nx, Ny, Nz);
```

Los códigos de un sufijo para esta función son `b` (byte), `s` (short), `i` (integer), `f` (float) y `d` (double). Además, añadimos el código de sufijo `v` cuando se proporcionen las componentes del vector mediante una matriz. Los valores de tipo bytes, short e integer se convierten a valores de tipo coma flotante comprendidos entre  $-1.0$  y  $1.0$ . La función `glNormal` define las componentes del vector normal a la superficie como valores de estado que se aplican a todos los comandos  `glVertex` subsiguientes. El vector normal predeterminado está en la dirección  $z$  positiva:  $(0.0, 0.0, 1.0)$ .

Para representación superficial plana, sólo necesitamos una normal a la superficie para cada polígono. El siguiente fragmento de código muestra cómo podemos definir la normal a cada polígono:

```
glNormal3fv (normalVector);
 glBegin (GL_TRIANGLES);
 glVertex3fv (vertex1);
 glVertex3fv (vertex2);
 glVertex3fv (vertex3);
 glEnd ( );
```

Si queremos aplicar el procedimiento de representación superficial de Gouraud al triángulo anterior, necesitaremos especificar un vector normal para cada vértice:

```
glBegin (GL_TRIANGLES);
    glNormal3fv (normalVector1);
    glVertex3fv (vertex1);
    glNormal3fv (normalVector2);
    glVertex3fv (vertex2);
    glNormal3fv (normalVector3);
    glVertex3fv (vertex3);
glEnd ( );
```

Aunque los vectores normales no necesitan especificarse como vectores unitarios, los cálculos se reducirán si definimos como vectores unitarios todas las normales a las superficies. Cualquier normal a la superficie no unitaria será automáticamente convertida a un vector unitario si antes ejecutamos el comando:

```
 glEnable (GL_NORMALIZE);
```

Este comando también renormaliza los vectores de las superficies cuando éstos se han visto modificados por transformaciones geométricas tales como un cambio de escala o una inclinación.

Otra opción disponible es la especificación de una lista de vectores normales que haya que combinar o asociar con una matriz de vértices (Sección 3.17 y Sección 4.3). Las instrucciones para crear una matriz de vectores normales son:

```
 glEnableClientState (GL_NORMAL_ARRAY);
 glEnable (GL_NORMALIZE);
 glNormalPointer (dataType, offset, normalArray);
```

Al parámetro `dataType` se le asigna el valor constante `GL_BYTE`, `GL_SHORT`, `GL_INT`, `GL_FLOAT` (el valor predeterminado) o `GL_DOUBLE`. El número de bytes entre vectores normales sucesivos en la matriz `normalArray` está dado por el parámetro `offset`, que tiene un valor predeterminado de 0.

## Operaciones de semitonos en OpenGL

En algunos sistemas pueden conseguirse diversos efectos de color y de escala de grises utilizando las rutinas de semitonos de OpenGL. Los patrones de aproximación de semitonos y las operaciones correspondientes son dependientes del hardware, y normalmente no tienen ningún efecto en los sistemas que dispongan de capacidades gráficas completas de color. Sin embargo cuando un sistema sólo tiene un pequeño número de bits por píxel, pueden aproximarse las especificaciones de color RGBA mediante patrones de semitono.

Podemos activar las rutinas de semitonos con:

```
 glEnable (GL_DITHER);
```

que es la opción predeterminada, y desactivarlas mediante la función,

```
 glDisable (GL_DITHER);
```

## 10.21 FUNCIONES DE TEXTURAS EN OpenGL

---

OpenGL dispone de un amplio conjunto de funciones de texturas. Podemos especificar un patrón para una línea, para una superficie, para un volumen interior o una región espacial, o como un subpatrón que haya que insertar dentro de otro patrón de texturas. También podemos aplicar y manipular los patrones de texturas de diversas formas. Además, los patrones de texturas pueden utilizarse para simular el mapeado de entorno. Las rutinas de texturas OpenGL sólo pueden utilizarse en el modo de color RGB (RGBA), aunque algunos parámetros pueden configurarse utilizando un índice a una tabla de colores.

## Funciones OpenGL para texturas lineales

Podemos utilizar un comando de la forma siguiente para especificar los parámetros de un patrón de texturas RGBA unidimensional mediante una matriz de colores de una única dimensión:

```
glTexImage1D (GL_TEXTURE_1D, 0, GL_RGBA, nTexColors, 0,
               dataFormat, dataType, lineTexArray);

 glEnable (GL_TEXTURE_1D);
```

Hemos asignado al primer argumento de la función `glTexImage1D` la constante simbólica OpenGL `GL_TEXTURE_1D`, para indicar que estamos definiendo una matriz de texturas para un objeto unidimensional: una línea. Si no estamos seguros de que el sistema vaya a soportar el patrón de texturas con los parámetros especificados, deberemos utilizar la constante simbólica `GL_PROXY_TEXTURE_1D` como principal argumento de `glTexImage1D`. Esto nos permite consultar primero el sistema antes de definir los elementos de la matriz de texturas; hablaremos de los procedimientos de consulta en una sección posterior.

Para el segundo y el quinto argumentos de esta función de ejemplo, utilizamos el valor 0. El primer valor 0 (segundo argumento) significa que esta matriz no es una reducción de otra matriz de texturas mayor. Para el quinto argumento, el valor 0 significa que no queremos que haya un borde alrededor de la textura. Si asignáramos a este quinto argumento el valor 1 (la única otra posibilidad), el patrón de texturas se mostraría con un borde de un píxel a su alrededor, que se utiliza para mezclar el patrón con los patrones de textura adyacentes. Para el tercer argumento, el valor `GL_RGBA` significa que cada color del patrón de texturas está especificado con cuatro valores RGBA. Podríamos habernos limitado a utilizar los tres valores de color RGB, pero los valores RGBA se procesan en ocasiones de manera más eficiente, ya que se alinean con las fronteras de la memoria del procesador. Es posible utilizar muchas otras especificaciones de color, incluyendo un único valor de intensidad o de luminancia. El parámetro `nTexColors`, en cuanto a argumento, debe tener un valor entero positivo que indique el número de colores del patrón lineal de texturas. Puesto que hemos proporcionado un valor 0 para el quinto argumento (el parámetro de borde), el número de colores del patrón de textura debe ser una potencia de 2. Si hubiéramos asignado al quinto argumento el valor 1, entonces el número de colores del patrón de texturas tendría que ser 2 más una potencia de 2. Los dos bordes de color se utilizan para permitir la mezcla de color con los patrones adyacentes. Podemos especificar el patrón de texturas unidimensional con hasta  $64 + 2$  colores y algunas implementaciones OpenGL permiten patrones de textura mayores. Los parámetros que describen los colores de la textura y los colores de borde se almacenan en `lineTexArray`. En este ejemplo, no tenemos ningún borde y cada grupo sucesivo de cuatro elementos de la matriz representa una componente de color del patrón de texturas. Por tanto, el número de elementos de `lineTexArray` es  $4 \times nTexColors$ . Como ejemplo específico, si quisieramos definir un patrón de texturas con 8 colores, la matriz de texturas deberían contener  $4 \times 8 = 32$  elementos.

Los parámetros `dataFormat` y `dataType` son similares a los argumentos de las funciones `glDrawPixels` y `glReadPixels` (Sección 3.19). Asignamos una constante simbólica OpenGL a `dataFormat` para indicar cómo se especifican los valores de color en la matriz de texturas. Por ejemplo, podríamos usar la constante simbólica `GL_BGRA` para indicar que las componentes de color están especificadas en el orden azul, verde, rojo, alpha. Para indicar el tipo de datos BGRA o RGBA, podemos asignar la constante OpenGL `GL_UNSIGNED_BYTE` al parámetro `dataType`. Otros posibles valores que podrían asignarse al parámetro `dataType`, dependiendo del formato de los datos que elijamos, son `GL_INT` y `GL_FLOAT`, entre otros.

Podemos mapear múltiples copias de una textura, o de cualquier subconjunto contiguo de los colores de textura, a un objeto de la escena. Cuando se mapea un grupo de elementos de textura sobre una o más áreas de píxel, las fronteras de los elementos de textura no suelen alinearse con las posiciones de las fronteras de los píxeles. Un área de píxel determinada podría estar contenida dentro de las fronteras de un único elemento de textura RGB (o RGBA), o podría solaparse con varios elementos de textura. Para simplificar los cálculos del ma-peado de texturas, utilizamos las siguientes funciones para proporcionar a cada píxel el color del elemento de textura más próximo:

```
glTexParameteri (GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER,
                 GL_NEAREST);
glTexParameteri (GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER,
                 GL_NEAREST);
```

La primera función es utilizada por las rutinas de texturado cuando una sección del patrón de texturas deba agrandarse para encajar en un rango de coordenadas específico dentro de la escena, mientras que la segunda función se utiliza cuando el patrón de texturas tenga que ser reducido (estas dos operaciones de texturado en OpenGL se denominan magnificación, `MAG` y minificación, `MIN`). Aunque la operación de asignar el color de textura más próximo a un píxel puede llevarse a cabo rápidamente, también puede hacer que aparezcan efectos de *aliasing*. Para calcular el color del píxel como combinación lineal de los colores de textura solapados, sustituimos la constante simbólica `GL_NEAREST` por `GL_LINEAR`. Hay muchos otros valores de parámetro que pueden especificarse con la función `glTexParameter`, y examinaremos dichas opciones en una sección posterior.

La especificación de patrones de textura OpenGL para una escena es en cierto modo similar a la especificación de vectores normales a la superficie, colores RGB u otros atributos. Necesitamos asociar un patrón con un determinado objeto, pero ahora, en lugar de un único valor de color, tenemos una colección de valores de color. Para una especie de texturas unidimensional, los valores de color se refieren mediante una única coordenada *s* que varía entre 0.0 y 1.0 a lo largo del espacio de texturas (Sección 10.16). Así, el patrón de textura se aplica a los objetos de una escena asignando valores de coordenadas de textura a las posiciones de los objetos. Podemos seleccionar un valor concreto de las coordenadas *s* en un espacio de texturas unidimensional mediante la siguiente instrucción:

```
glTexCoord1* (sCoord);
```

Los códigos de sufijo disponibles para esta función son `b` (byte), `s` (short), `i` (integer), `f` (float) y `d` (double), dependiendo del tipo de dato del parámetro `sCoord` que especifica la coordenada de textura. También podemos usar el sufijo `v` si el valor de la coordenada *s* se proporciona mediante una matriz. Al igual que sucede con los parámetros de color y otros similares, la coordenada *s* es un parámetro de estado, que se aplica a todas las posiciones en coordenadas universales que se definen de forma subsiguiente. El valor predeterminado para la coordenada *s* es 0.0.

Para mapear un patrón lineal de textura sobre una serie de posiciones dentro de una escena definida en coordenadas universales, asignamos las coordenadas *s* a los puntos extremos de un segmento lineal. Entonces, los colores de textura pueden aplicarse al objeto de diversas formas, y el método predeterminado que OpenGL utiliza consiste en multiplicar cada valor de color de píxel del objeto por el correspondiente valor de color del patrón de textura. Si el color de la línea es blanco (1.0, 1.0, 1.0, 1.0), que es el color predeterminado para los objetos de una escena, la línea sólo se mostrará con los colores de la textura.

En el siguiente ejemplo, creamos un patrón de textura lineal de cuatro elementos con colores verde y rojo alternantes. Todo el patrón de textura, de 0.0 a 1.0, se asigna entonces a un segmento de línea recta. Puesto que la línea es blanca, de manera predeterminada, se mostrará dentro de la escena con los colores de la textura.

---

```
GLint k;
GLubyte texLine [16]; // Matriz de texturas de 16 elementos.

/* Definir dos elementos verdes para el patrón de textura.
/* Cada color de textura se especifica en cuatro posiciones de la matriz.
*/
for (k = 0; k <= 2; k += 2)
{
    texLine [4*k] = 0;
```

```

    texLine [4*k+1] = 255;
    texLine [4*k+2] = 0;
    texLine [4*k+3] = 255;
}

/* Definir dos elementos rojos para el patrón de textura. */
for (k = 1; k <= 3; k += 2)
{
    texLine [4*k] = 255;
    texLine [4*k+1] = 0;
    texLine [4*k+2] = 0;
    texLine [4*k+3] = 255;
}

glTexParameteri (GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri (GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);

glTexImage1D (GL_TEXTURE_1D, 0, GL_RGBA, 4, 0, GL_RGBA, GL_UNSIGNED_BYTE, texLine);

 glEnable (GL_TEXTURE_1D);

/* Asignar el rango completo de colores de textura a un segmento de línea. */
 glBegin (GL_LINES);
    glTexCoord1f (0.0);
    glVertex3fv (endPt1);
    glTexCoord1f (1.0);
    glVertex3fv (endPt2);
 glEnd ( );

glDisable (GL_TEXTURE_1D);

```

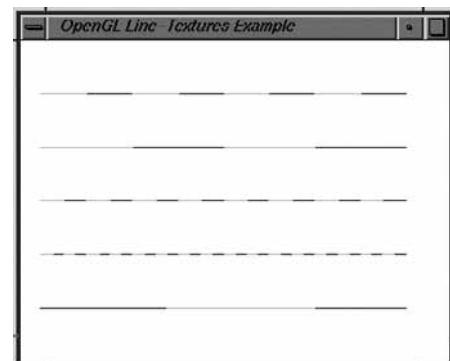
El segmento de línea se muestra con secciones verdes y rojas alternativas. Podemos asignar cualquier valor que queramos a las coordenadas  $s$ . Podemos, por ejemplo, mapear sobre la línea los colores rojo y verde intermedios del patrón de textura con las siguientes instrucciones:

```

glBegin (GL_LINES);
    glTexCoord1f (0.25);
    glVertex3fv (wcPt1);
    glTexCoord1f (0.75);
    glVertex3fv (wcPt2);
glEnd ( );

```

Así, la primera parte de la línea será roja y la segunda mitad verde. También podríamos utilizar valores de  $s$  fuera del rango comprendido entre 0.0 y 1.0. Por ejemplo, si asignáramos a  $s$  el valor  $-2.0$  para un extremo de la línea y el valor  $2.0$  en el otro extremo, el patrón de texturas se mapearía sobre la línea cuatro veces. La línea aparecería entonces con 16 secciones verdes y 16 secciones rojas. Para los valores de coordenada  $s$  situados fuera del intervalo unitario, las partes enteras se ignoran, a no ser que especifiquemos que los valores  $s$  deben fijarse a 0 o a 1.0 cuando se produzca desbordamientos. La Figura 10.114 muestra algunos posibles patrones de línea que pueden visualizarse con la matriz que contiene los valores RGB de dos colores verdes y dos colores rojos.



**FIGURA 10.114.** Ejemplos de mapeado de un patrón de texturas OpenGL de un único subíndice sobre un segmento de línea blanco.

En los patrones de textura OpenGL hay disponible un amplio rango de parámetros y opciones, pero antes de profundizar en estas características de las rutinas de texturas de OpenGL, vamos a ver primero las funciones básicas necesarias para generar patrones de textura bidimensionales y tridimensionales.

## Funciones OpenGL para texturas superficiales

Podemos especificar los parámetros para un espacio de texturas RGBA bidimensional utilizando funciones similares a las empleadas en nuestro ejemplo del patrón de texturas unidimensional:

```
glTexImage2D (GL_TEXTURE_2D, 0, GL_RGBA, texWidth,
              texHeight, 0, dataFormat, dataType, surfTexArray);

 glEnable (GL_TEXTURE_2D);
```

La única diferencia aquí es que debemos especificar tanto una anchura (número de columnas) como una altura (número de filas), para la matriz de texturas de tres subíndices. Tanto la anchura como la altura deben ser una potencia de 2, sin un borde, o 2 más una potencia de 2 en caso de utilizarse un borde. De nuevo, utilizamos componentes de color RGBA, y especificamos en el ejemplo que el patrón no tiene ningún borde y no es una reducción de un patrón de texturas mayor. Por tanto, el tamaño de la matriz almacenada en `surfTexArray` es  $4 \times \text{texWidth} \times \text{texHeight}$ . Para patrones de textura bidimensionales, asignamos a los elementos de la matriz de textura los valores de color de abajo a arriba. Comenzando en la esquina inferior izquierda del patrón de color, especificamos los elementos de la primera fila de la matriz, asignándoles los valores RGBA correspondientes a la fila inferior del espacio de texturas, y especificamos los elementos de la última fila de la matriz asignándoles los valores RGBA correspondientes a la parte superior del espacio de texturas rectangular (Figura 10.103).

Al igual que con el patrón de texturas lineal, a los píxeles de superficie de una escena puede asignárseles el color de textura más próximo a un color de textura interpolado. Seleccionamos cualquiera de estas opciones con las mismas dos funciones `glTexParameter` que ya empleamos para las texturas unidimensionales. Una función especifica la opción que hay que usar cuando se agranda un patrón de texturas para que encaje en un rango de coordenadas y la otra función especifica la opción que hay que emplear con las reducciones de los patrones. Además, un patrón de textura bidimensional puede estirarse en una dirección y comprimirse en la otra. Por ejemplo, las siguientes instrucciones especifican que las rutinas de texturado deben mostrar las posiciones de superficie proyectadas utilizando el color de textura más próximo:

```
glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                 GL_NEAREST);

glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                 GL_NEAREST);
```

Para asignar un color de textura interpolado a los píxeles de las superficies, utilizamos la constante simbólica `GL_LINEAR` en lugar de `GL_NEAREST`.

Una posición de coordenadas en el espacio de texturas bidimensionales se selecciona mediante:

```
glTexCoord2* (sCoord, tCoord);
```

El espacio de texturas está normalizado, de modo que el patrón se referencia mediante valores de coordenadas comprendidos en el rango de 0.0 a 1.0. Sin embargo, podemos utilizar cualquier valor de coordenada de texturas para replicar un patrón a lo largo de una superficie. Las coordenadas de texturas pueden especificarse en varios formatos, indicándose el formato de los datos mediante un código de sufijo igual a b, s, i, f o d. También agregaremos el sufijo v si las coordenadas de textura se especifican mediante una matriz.

Para ilustrar las funciones OpenGL aplicables a un espacio de textura bidimensional, el siguiente segmento de código especifica un patrón 32 por 32 y lo mapea sobre una superficie cuadrilátera. Cada color de textura se especifica mediante cuatro componentes RGBA y el patrón carece de borde.

---

```
GLubyte texArray [32][32][4];

/* Siguiente: asignar las componentes del color de textura a texArray. */
/* Seleccionar la opción de color más próximo. */
glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);

glTexImage2D (GL_TEXTURE_2D, 0, GL_RGBA, 32, 32, 0, GL_RGBA,
              GL_UNSIGNED_BYTE, texArray);

 glEnable (GL_TEXTURE_2D);

/* Asignar el rango completo de colores de textura a un cuadrilátero. */
 glBegin (GL_QUADS);
    glTexCoord2f (0.0, 0.0); glVertex3fv (vertex1);
    glTexCoord2f (1.0, 0.0); glVertex3fv (vertex2);
    glTexCoord2f (1.0, 1.0); glVertex3fv (vertex3);
    glTexCoord2f (0.0, 1.0); glVertex3fv (vertex4);
 glEnd ( );

 glDisable (GL_TEXTURE_2D);
```

---

## Funciones OpenGL para texturas volumétricas

Las funciones para un espacio de texturas tridimensional son simplemente extensiones de las que se emplean para los espacios de texturas bidimensionales. Por ejemplo, podemos especificar una matriz de texturas RGBA de cuatro subíndices y sin ningún borde mediante las funciones:

```
glTexImage3D (GL_TEXTURE_3D, 0, GL_RGBA, texWidth, texHeight,
              texDepth, 0, dataFormat, dataType, volTexArray);

 glEnable (GL_TEXTURE_3D);
```

Los colores de textura RGBA se almacenan en `volTexArray`, que contienen  $4 \times \text{texWidth} \times \text{texHeight} \times \text{texDepth}$  elementos. La anchura, altura y profundidad de la matriz debe ser una potencia de 2 o una potencia de 2 más 2.

Con las siguientes instrucciones, podremos mostrar los píxeles utilizando el color de textura más próximo:

```
glTexParameteri (GL_TEXTURE_3D, GL_TEXTURE_MAG_FILTER,
                 GL_NEAREST);
glTexParameteri (GL_TEXTURE_3D, GL_TEXTURE_MIN_FILTER,
                 GL_NEAREST);
```

Para colores de textura linealmente interpolados, sustituimos el valor `GL_NEAREST` por `GL_LINEAR`.

Las coordenadas de textura tridimensionales se seleccionan mediante

```
glTexCoord3* (sCoord, tCoord, rCoord);
```

Cada posición seleccionada en el espacio de textura se asocia entonces con una posición de coordenadas espaciales dentro de una escena definida en coordenadas universales.

## Opciones de color OpenGL para patrones de texturas

Los elementos de un espacio de texturas pueden especificarse de muchas formas distintas. El tercer argumento de las funciones `glTexImage1D`, `glTexImage2D` y `glTexImage3D` se utiliza para especificar el formato general y el número de componentes de color para cada elemento de un patrón. Hay disponibles casi 40 constantes simbólicas para poder realizar estas especificaciones. Por ejemplo, cada elemento de textura puede ser un conjunto de valores RGBA, un conjunto de valores RGB, un único valor alpha, un único valor de intensidad roja, un único valor de luminancia o un valor de luminancia junto con un valor alpha. Además, algunas constantes también especifican el tamaño en bits. La constante OpenGL `GL_R3_G3_B2`, por ejemplo, especifica un color RGB de un byte (8 bits), con 3 bits asignados a la componente roja, 3 bits asignados a la componente verde y 2 bits a la azul.

El parámetro `dataFormat` de las funciones de textura se utiliza entonces para especificar el formato concreto de los elementos de textura. Podemos seleccionar una cualquiera de entre once constantes simbólicas para este parámetro. Esto nos permite especificar cada elemento de textura como un índice a una tabla de colores, un único valor alpha, un único valor de luminancia, una pareja de valores luminancia-alpha, un único valor de intensidad para una de las componentes RGB, las tres componentes RGB o las cuatro componentes de una especificación RGBA, en el orden BGRA. Al parámetro `dataType` se le asigna un valor tal como `GL_BYTE`, `GL_INT`, `GL_FLOAT` o una constante simbólica que especifique tanto el tipo de dato como el tamaño en bits. Podemos seleccionar un valor de entre un conjunto de 20 constantes simbólicas aplicables al parámetro que especifica el tipo de dato.

## Opciones OpenGL para el mapeado de texturas

Los elementos de textura pueden aplicarse a un objeto de modo que los valores de textura se combinen con los componentes actuales de color del objeto, o pueden emplearse los valores de textura para sustituir el color del objeto. Para seleccionar un método de mapeado de textura se utiliza la función:

```
glTexEnvi (GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
            applicationMethod);
```

Si se asigna al parámetro `applicationMethod` el valor `GL_REPLACE`, entonces el color, la luminancia, la intensidad o el valor alpha de textura sustituirán los correspondientes valores que tenga el objeto. Por ejemplo, un patrón de textura de valores alpha sustituirá los valores alpha del objeto. Se utilizan operaciones similares de sustitución con los patrones de textura especificados con un único valor de luminancia o intensidad. Un patrón de valores de intensidad verde sustituirá las componentes verdes del color del objeto.

Si se asigna el valor `GL_MODULATE` al parámetro `applicationMethod`, lo que se obtendrá es una «modulación» de los valores de color del objeto. Es decir, los valores actuales del objeto se multiplican por los valores de la textura. Los resultados específicos dependerán del formato de los elementos del patrón de textura,

de modo que, por ejemplo, los valores alpha modularán los valores alpha y los valores de intensidad modularán los valores de intensidad. El método de aplicación predeterminado para un patrón de texturas es `GL_MODULATE`. Si el color de un objeto es blanco (el color predeterminado de los objetos), la operación de modulación producirá el mismo resultado que una operación de sustitución, dependiendo de cómo se hayan especificado los elementos del patrón de texturas.

También podemos usar la constante simbólica `GL_DECAL` para las operaciones de mapeado de texturas, lo que hará que se utilicen los valores alpha RGBA como coeficientes de transparencia. En este caso, el objeto se trata como si fuera transparente con el color de textura en segundo plano. Si el patrón de textura sólo contiene valores RGB, sin ninguna componente alpha, el color de textura sustituirá al color del objeto. Asimismo, en algunos casos, como por ejemplo cuando el patrón de textura sólo contiene valores alpha, este modo de aplicación del mapeado de texturas no está definido.

Cuando asignamos la constante `GL_BLEND` al parámetro `application-Method`, las rutinas de texturado realizan una mezcla de colores utilizando un color especificado con la función:

```
glTexEnv* ( GL_TEXTURE_ENV, GL_TEXTURE_ENV_COLOR,
            blendingColor );
```

Agregaremos el sufijo `i` o `f` de acuerdo con el tipo de dato del color de mezcla. Asimismo, se añadirá el sufijo `v` si el color de mezcla se especifica mediante una matriz.

## Envolvimiento de texturas en OpenGL

Cuando los valores de coordenadas en un espacio de texturas caen fuera del rango comprendido entre 0 y 1.0, podemos decidir replicar los patrones de escritos en la matriz de texturas utilizando el comando:

```
glTexParameter* ( texSpace, texWrapCoord, GL_REPEAT );
```

Los patrones se replican utilizando únicamente la parte fraccionaria del valor de la coordenada dentro del espacio de texturas. Al parámetro `texSpace` se le asigna uno de los valores simbólicos `GL_TEXTURE_1D`, `GL_TEXTURE_2D` o `GL_TEXTURE_3D` y el parámetro `texWrap-Coord` designa una coordenada dentro del espacio de texturas utilizando `GL_TEXTURE_WRAP_S`, `GL_TEXTURE_WRAP_T` o `GL_TEXTURE_WRAP_R`.

Para hacer que la coordenada de texturas quede fijada en el límite del intervalo unitario cuando se produce un desbordamiento, utilizamos la constante simbólica `GL_CLAMP` en lugar de `GL_REPEAT`. Si una coordenada de textura tiene un valor superior a 1.0, con esta opción se le asignará el valor 1.0. De forma similar, si una coordenada tiene un valor inferior a 0.0, con esta opción se le asignaría el valor 0.0. Podemos especificar cualquier combinación de repetición o fijación para las coordenadas de un espacio de texturas concreto. La opción predeterminada para todas las coordenadas es `GL_REPEAT`.

## Copia de patrones de texturas OpenGL desde el búfer de imagen

Podemos obtener un patrón original o un subpatrón a partir de los valores almacenados en el búfer de imagen. La siguiente función define un patrón bidimensional para el estado de textura actual utilizando un bloque de valores de píxel RGBA:

```
glCopyTexImage2D ( GL_TEXTURE_2D, 0, GL_RGBA, x0, y0, texWidth,
                  texHeight, 0 );
```

Los dos valores 0 en la lista de argumentos indican de nuevo que este patrón no es una reducción y que no tiene un borde. La posición (`x0`, `y0`) del búfer de imagen, relativa a la esquina inferior izquierda del búfer, hace referencia a la esquina inferior izquierda de un bloque de colores de píxel de tamaño `texWidth` por `texHeight`.

Hay disponible otra función similar para obtener un bloque de colores de píxel como subpatrón de texturas:

```
glCopyTexSubImage2D (GL_TEXTURE_2D, 0, xTexElement,
                     yTexElement, x0, y0, texSubWidth, texSubHeight);
```

Este bloque de valores de píxel se almacena en el patrón actual en la posición correspondiente al elemento de textura especificado (`xTexElement`, `yTexElement`). Los parámetros `texSubWidth` y `texSubHeight` proporcionan el tamaño del bloque de píxeles, cuya esquina inferior izquierda estará en la posición (`x0`, `y0`) del búfer de imagen.

## Matrices de coordenadas de texturas en OpenGL

Al igual que con los datos de color, los vectores normales a la superficie y los indicadores de las aristas de los polígonos, también podemos especificar las coordenadas de textura en listas que pueden combinarse o asociarse con matrices de vértices (Sección 3.17 y Sección 4.3).

```
 glEnableClientState (GL_TEXTURE_COORD_ARRAY);
 glTexCoordPointer (nCoords, dataType, offset, texCoordArray);
```

El parámetro `nCoords` deberá tener el valor 1, 2, 3 o 4, especificando la dimensionalidad del patrón de texturas. El valor predeterminado de 4 se utiliza para referenciar el espacio de texturas según un sistema de coordenadas homogéneas, por lo que la posición en el espacio de texturas se calculará dividiendo los primeros tres valores de coordenadas por el cuarto. Esta forma resulta útil, por ejemplo, cuando el patrón de textura es una fotografía en perspectiva. Al parámetro `dataType` se le asigna el valor constante `GL_SHORT`, `GL_INT`, `GL_FLOAT` (el valor predeterminado) o `GL_DOUBLE`. El desplazamiento en bytes entre las posiciones sucesivas de coordenadas dentro de la matriz `texCoordArray` se especifica mediante el parámetro `offset`, que tiene un valor predeterminado igual a 0.

## Denominación de los patrones de textura OpenGL

A menudo, resulta útil emplear varios patrones de textura en una aplicación, de modo que OpenGL permite crear múltiples patrones de textura nominados. Entonces, simplemente tendremos que indicar qué textura nominada hay que aplicar en cada momento. Este método es mucho más eficiente que invocar la función `glTexImage` cada vez, ya que cada llamada a `glTexImage` requiere volver a crear un patrón, posiblemente a partir de los valores de color contenidos en un archivo de datos. Para asignar un nombre al patrón de texturas, seleccionamos un entero positivo (sin signo) antes de definir el patrón. Como ejemplo, las siguientes instrucciones asignan un nombre al patrón de línea verde y roja de nuestro ejemplo anterior, denominándolo como textura número 3, y luego activan el patrón:

```
 glBindTexture (GL_TEXTURE_1D, 3);
 glTexImage1D (GL_TEXTURE_1D, 0, GL_RGBA, 4, 0, GL_RGBA,
               GL_UNSIGNED_BYTE, texLine);

 glBindTexture (GL_TEXTURE_1D, 3);
```

La primera instrucción `glBindTexture` asigna un nombre al patrón, mientras que la segunda llamada a `glBindTexture` designa el patrón como **estado de textura actual**. Si hemos creado múltiples patrones de textura, podríamos llamar de nuevo a `glBindTexture` con otro nombre de patrón para activar dicha textura con el fin de aplicarla a algún objeto de la escena. Para un patrón bidimensional o tridimensional, cambiaremos el primer argumento de la función `glBindTexture` a `GL_TEXTURE_2D` o `GL_TEXTURE_3D`. Cuando se invoca por primera vez el nombre de una textura, se crea un patrón de textura utilizando los valores predeterminados para los parámetros del patrón.

Podemos borrar uno o más patrones de textura existentes mediante el comando:

```
 glDeleteTextures (nTextures, texNamesArray);
```

El parámetro `nTextures` especifica el número de nombres de patrones enumerados en la matriz `texNamesArray`.

También podemos dejar que sea OpenGL quien seleccione un nombre para el patrón, con el fin de no tener que controlar los nombres que ya han sido utilizados. Por ejemplo,

```
static GLuint texName;

 glGenTextures (1, texName);
 glBindTexture (GL_TEXTURE_2D, texName);
```

Como ejemplo, el siguiente código obtiene una lista de seis nombres de textura no utilizados y emplea uno de ellos para crear un patrón:

```
static GLuint texNamesArray [6];

 glGenTextures (1, texNamesArray [3]);
 glBindTexture (GL_TEXTURE_2D, texNamesArray [3]);
```

En OpenGL hay disponible un comando de consulta para averiguar si un cierto nombre de textura ya está siendo utilizado en algún patrón existente:

```
glIsTexture (texName);
```

Esta función devuelve el valor `GL_TRUE` si `texName` es el nombre de un patrón existente, y en caso contrario devuelve el valor `GL_FALSE`. También se devuelve un valor `GL_FALSE` si `texName = 0` o si se produce un error.

## Subpatrones de textura en OpenGL

Una vez definido un patrón de texturas, podemos crear otro patrón, denominado subpatrón, para modificar cualquier parte del patrón original o la totalidad del mismo. Los valores de textura del subpatrón sustituyen los valores especificados en el patrón original. Usualmente, este proceso es más eficiente que volver a crear una textura con nuevos elementos. Por ejemplo, la siguiente función especifica un conjunto de valores de color RGBA que deben sustituir una sección de una textura bidimensional que no tiene ningún borde y que no es una reducción de un patrón de mayor tamaño:

```
glTexSubImage2D (GL_TEXTURE_2D, 0, xTexElement,
                  yTexElement, GL_RGBA, texSubWidth, texSubHeight,
                  0, dataFormat, dataType, subSurfTexArray);
```

Los parámetros `xTexElement` y `yTexElement` se utilizan para seleccionar una posición de coordenadas entera de un elemento de textura dentro del patrón original, referenciando la posición (0, 0) al elemento de texturas situado en la esquina inferior izquierda del patrón. El subpatrón se pega sobre el patrón original, con su esquina inferior izquierda en la posición (`xTexElement`, `yTexElement`). Los parámetros `TexSubWidth` y `TexSubHeight` proporcionan el tamaño del subpatrón. El número de elementos de color en la matriz `subSurfTexArray` para un patrón de textura RGBA es  $4 \times \text{texSubWidth} \times \text{texSubHeight}$ . Los restantes parámetros son iguales que en la función `glTexImage`, y pueden definirse subpatrones similares para texturas unidimensionales y tridimensionales.

## Patrones de reducción de texturas en OpenGL

Para tamaños de objetos reducidos, podemos utilizar rutinas OpenGL para crear una serie de patrones de reducción de texturas, que se denominan *mipmaps* (Sección 10.17). Una forma de crear una secuencia de patrones de reducción consiste en invocar la función `glTexImage` repetidamente utilizando valores enteros cada vez mayores para el segundo argumento (el «número de nivel») dentro de la función. El patrón original

se referencia mediante el nivel de reducción 0. Un patrón de reducción que tenga la mitad de tamaño que el patrón original tendrá asignado el número de nivel 1, el segundo patrón de reducción a la mitad de tamaño se designará como número de nivel 2, y así sucesivamente para las restantes reducciones. La función `copyTexImage` también genera un patrón de reducción cuando especificamos el número de nivel 1 o superior.

Alternativamente, podemos dejar que OpenGL genere los patrones de reducción automáticamente. Por ejemplo, podríamos utilizar la siguiente función GLU para obtener los patrones de reducción RGBA para una textura de superficial de 16 por 16:

```
gluBuild2DMipmaps (GL_TEXTURE_2D, GL_RGBA, 16, 16, GL_RGBA,
                    GL_UNSIGNED_BYTE, surfTexArray);
```

Esta función generará un conjunto completo de cuatro patrones, con los tamaños reducidos de 8 por 8, 4 por 4, 2 por 2 y 1 por 1. También podemos especificar una reducción seleccionada utilizando la función:

```
gluBuild2DMipmapLevels (GL_TEXTURE_2D, GL_RGBA, 16, 16,
                        GL_RGBA, GL_UNSIGNED_BYTE, 0, minLevel, maxLevel,
                        surfTexArray);
```

Esta función genera patrones de reducción para un rango de números de nivel especificados mediante los parámetros `minLevel` y `maxLevel`. En cada caso, los *mipmaps* se construyen para el patrón de texturas actual, específico en el nivel número 0.

Podemos seleccionar un método para determinar los colores de los píxeles a partir de los patrones de reducción utilizando la función `glTexParameter` y la constante simbólica `GL_TEXTURE_MIN_FILTER`. Como ejemplo, la siguiente función designa el procedimiento de mapeado para un patrón de texturas bidimensional:

```
glTexParameter (GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                GL_NEAREST_MIPMAP_NEAREST);
```

Esta función especifica que las rutinas de textura deben utilizar el patrón de reducción que se ajuste de forma más precisa al tamaño de píxel (`MIPMAP_NEAREST`). Entonces, a los píxeles se les asignará el color del elemento de textura más próximo (`GL_NEAREST`) en dicho patrón de reducción. Con la constante simbólica `GL_LINEAR_MIPMAP_NEAREST`, especificamos una combinación lineal de los colores de textura contenidos en el patrón de reducción más próximo. Con `GL_NEAREST_MIPMAP_LINEAR` (el valor predeterminado), especificamos un color promedio que se calculará a partir de los elementos de textura más próximos en cada uno de los patrones de reducción que se ajusten mejor al tamaño de píxel. Por su parte, `GL_LINEAR_MIPMAP_LINEAR` calcula el color del píxel utilizando una combinación lineal de los colores de textura del conjunto de patrones de reducción que mejor se ajusten en tamaño.

## Bordes de texturas en OpenGL

Cuando se aplican múltiples texturas, o múltiples copias de una misma textura, a un objeto, pueden aparecer efectos de *aliasing* en los bordes de los patrones adyacentes cuando se calculan los colores de píxel interpolando linealmente los colores de textura. Podemos evitar esto incluyendo un borde en cada patrón de texturas, que hará que los colores del borde se ajusten a los colores existentes en la frontera de la textura especificada por el patrón adyacente.

Podemos especificar un color de borde para las texturas de diversas formas. El valor de color de un patrón adyacente puede copiarse en el borde de otro patrón utilizando la función `glTexSubImage`, o bien pueden asignarse directamente los colores de borde en la matriz de texturas especificada con la función `glTexImage`. Otra opción consiste en especificar un color de borde utilizando la rutina `glTexParameter`. Por ejemplo, podemos asignar un color de borde a un patrón bidimensional mediante la función:

```
glTexParameterfv (GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR,
                  borderColor);
```

donde al parámetro `borderColor` se le asigna un conjunto de cuatro componentes de color RGBA. El color de borde predeterminado es el negro, (0.0, 0.0, 0.0, 0.0).

## Texturas proxy en OpenGL

En cualquiera de las funciones `glTexImage`, podemos asignar al primer argumento una constante simbólica denominada proxy de textura. El propósito de esta constante consiste en almacenar la definición del patrón de texturas hasta que veamos si hay suficientes recursos como para poder procesar este patrón. Para un patrón bidimensional, la constante de proxy es `GL_PROXY_TEXTURE_2D`, y hay disponibles constantes similares para los patrones de textura lineales y volumétricos. Una vez especificado el proxy de textura, podemos utilizar `glGetTexLevelFunction` para determinar si se pueden emplear valores específicos de los parámetros.

Como ejemplo de utilización de un proxy de textura, las siguientes instrucciones consultan el sistema para determinar si puede usarse la altura especificada para un patrón bidimensional:

```
GLint texHeight;

glTexImage2D (GL_PROXY_TEXTURE_2D, 0, GL_RGBA12, 16, 16, 0,
              GL_RGBA, GL_UNSIGNED_BYTE, NULL);
glGetTexLevelParameteriv (GL_PROXY_TEXTURE_2D, 0, GL_RGBA12,
                          GL_TEXTURE_HEIGHT, &texHeight);
```

Si el sistema no puede admitir la altura de patrón solicitada (16, en este caso) se devolverá un valor 0 en el parámetro `texHeight`. En caso contrario, el valor devuelto coincidirá con el valor solicitado. Otros parámetros de los patrones pueden consultarse de forma similar utilizando constantes simbólicas tales como `GL_TEXTURE_WIDTH`, `GL_TEXTURE_DEPTH`, `GL_TEXTURE_BORDER` y `GL_TEXTURE_BLUE_SIZE`. En cada caso, si se devuelve un valor 0 querrá decir que el valor solicitado para el parámetro en la función `glTexImage` no puede admitirse. Para valores de datos en coma flotante, tendremos que sustituir el código de sufijo `i` por el código `f`.

Aunque obtengamos una respuesta afirmativa para una textura propuesta, puede que no sea posible almacenar el patrón en memoria. Esto puede suceder cuando haya otro patrón ocupando la memoria disponible.

## Texturado automático de superficies cuádricas

OpenGL dispone de rutinas para generar automáticamente las coordenadas de las texturas en ciertas aplicaciones. Esta característica resulta particularmente útil cuando sea difícil determinar directamente las coordenadas de superficie de un objeto, y hay disponible una función GLU para aplicar estas rutinas de superficies cuádricas.

Para mapear un patrón de textura sobre una superficie cuádrica, primero definimos los parámetros para el espacio de texturas y a continuación invocamos la siguiente función y definimos el objeto cuádrico, como se describe en la Sección 8.6:

```
gluQuadricTexture (quadSurfObj, GL_TRUE)
```

El parámetro `quadSurfObj` en esta función es el nombre del objeto cuádrico. Si queremos desactivar el texturado de la superficie cuádrica, tendremos que cambiar la constante simbólica `GL_TRUE` por `GL_FALSE`.

## Coordenadas de textura homogéneas

Podemos especificar una posición tetradimensional en el espacio de texturas mediante:

```
glTexCoord4* (sCoord, tCoord, rCoord, htexCoord);
```

Las coordenadas de textura se transforman utilizando una matriz 4 por 4 de la misma manera que se transforman las coordenadas de la escena: cada coordenada se divide por el parámetro homogéneo (Sección 5.2).

Así, los valores de las coordenadas de textura  $s$ ,  $t$  y  $r$  en la función anterior se dividen por el parámetro homogéneo  $h_{\text{tex}}$  para generar la posición real dentro del espacio de texturas.

Las coordenadas homogéneas en el espacio de texturas son útiles cuando se combinan múltiples efectos de perspectiva en una misma imagen. Por ejemplo, una vista en perspectiva de un objeto puede incluir un patrón de texturas producido con una transformación diferente de proyección en perspectiva. El patrón de textura puede entonces modificarse utilizando coordenadas homogéneas de textura para ajustar la perspectiva de la misma. Pueden conseguirse muchos otros efectos utilizando coordenadas de texturas homogéneas para manipular el mapeado de texturas.

## Opciones adicionales para texturas en OpenGL

En OpenGL hay disponibles funciones para realizar muchas otras manipulaciones y aplicaciones de texturas. Si obtenemos un patrón de textura (de una fotografía o de otra fuente) cuyo tamaño no sea una potencia de 2, OpenGL proporciona una función para modificar el tamaño del patrón. En algunas implementaciones de OpenGL, hay disponibles rutinas de multitexturizado para pegar múltiples patrones de textura sobre un objeto. El mapeado de entorno puede simularse en OpenGL creando un mapa de textura con la forma de una superficie esférica y pueden generarse automáticamente coordenadas de textura para patrones de entorno esféricos, así como para otras aplicaciones de las texturas.

## 10.22 RESUMEN

---

En general, los objetos son iluminados con energía radiante procedente de las emisiones de luz y de las superficies reflectantes de una escena. Las fuentes luminosas pueden modelarse como objetos puntuales o pueden tener un tamaño finito. Además, las fuentes luminosas pueden ser direccionales, y pueden tratarse como fuentes infinitamente distantes o como fuentes luminosas locales. Normalmente, se suele aplicar una atenuación radial a la luz transmitida, utilizando una función cuadrática inversa de la distancia, y las luces de foco también pueden atenuarse de forma angular. Las superficies reflectantes de una escena son opacas, completamente transparentes o parcialmente transparentes. Los efectos de iluminación se describen en términos de componentes difusa y especular tanto para las reflexiones como para las refracciones.

La intensidad de la luz en una posición de la superficie se calcula utilizando un modelo de iluminación, y el modelo básico de iluminación en la mayoría de los paquetes gráficos utiliza aproximaciones simplificadas de las leyes físicas. Estos cálculos de iluminación proporcionan un valor de intensidad lumínosa para cada componente RGB de la luz reflejada en una posición de la superficie y para la luz transmitida a través de un objeto transparente. El modelo básico de iluminación admite normalmente múltiples fuentes luminosas como emisores puntuales, pero pueden ser fuentes distantes, fuentes locales o luces de foco. La luz ambiente para una escena se describe mediante una intensidad fija para cada componente de color RGB, intensidad que es la misma para todas las superficies. Las reflexiones difusas de una superficie se consideran proporcionales al coseno de la distancia angular con respecto a la dirección de la normal a la superficie. Las reflexiones especulares se calculan utilizando el modelo de Phong. Por su parte, los efectos de transparencia suelen aproximarse utilizando un coeficiente simple de transparencia para un material, aunque pueden modelarse efectos de refracción más precisos utilizando la ley de Snell. Los efectos de sombra correspondientes a las fuentes luminosas individuales pueden añadirse identificando las regiones de una escena que no son visibles desde la fuente luminosa. Asimismo, los cálculos necesarios para obtener las reflexiones de la luz y los efectos de transmisión para los materiales translúcidos no suelen incluirse en el modelo básico de iluminación, aunque podemos modelarlos utilizando métodos que dispersen las componentes de luz difusa.

Los valores de intensidad calculados con un modelo de iluminación se mapean sobre los niveles de intensidad disponibles en el sistema de visualización que se esté utilizando. Los distintos sistemas proporcionan una escala de intensidad logarítmica con el fin de suministrar un conjunto de niveles de intensidad que se

incrementen con diferenciales de brillo percibido de la misma magnitud. Se aplica una corrección gamma a los valores de intensidad para corregir la no linealidad de los dispositivos de visualización. Con los monitores monocromo, podemos utilizar patrones de semitonos y técnicas de aleatorización para simular un rango de valores de intensidad. Las aproximaciones de semitonos pueden utilizarse también para incrementar el número de niveles de intensidad en aquellos sistemas que sean capaces de mostrar más de dos intensidades por píxel. Los métodos de aleatorización ordenada, de difusión de errores y de difusión de puntos se utilizan para simular un rango de intensidades cuando el número de puntos que hay que mostrar en la escena es igual al número de píxeles del dispositivo de visualización.

La representación de superficies en los paquetes gráficos se lleva a cabo aplicando los cálculos del modelo básico de iluminación a procedimientos de línea de exploración que extrapole los valores de intensidad a partir de unos cuantos puntos de la superficie, con el fin de calcular todas las posiciones de píxel proyectadas de la superficie. Con la representación superficial de intensidad constante, también denominada representación plana, se utiliza un color calculado para mostrar todos los puntos de una superficie. La representación superficial plana resulta suficientemente precisa para los poliedros o para las mallas poligonales de aproximación a superficies curvas cuando las posiciones de visualización y de las fuentes luminosas están alejadas de los objetos de una escena. La representación superficial de Gouraud aproxima las reflexiones luminosas producidas en superficies curvas teseladas calculando los valores de intensidad en los vértices de los polígonos e interpolando linealmente dichos valores de intensidad en las caras del polígono. Un procedimiento más preciso, aunque más lento, de representación superficial es el método de Phong, que interpola los vectores normales promediados de los vértices de los polígonos a lo largo de las caras poligonales. Entonces, se utiliza el modelo básico de iluminación para calcular las intensidades superficiales en cada posición de la superficie proyectada, utilizando los valores interpolados para los vectores normales a la superficie. Las técnicas rápidas de representación superficial de Phong utilizan aproximaciones en serie de Taylor para reducir el tiempo de procesamiento dedicado a los cálculos de intensidad.

El trazado de rayos es un método para obtener efectos globales de reflexión especular y de transmisión, trazando rayos luminosos a través de una escena hasta las posiciones de píxel. Los rayos de los píxeles se trazan a través de una escena, rebotando de objeto en objeto a medida que se acumulan las contribuciones de intensidad. Para cada píxel se construye un árbol de trazado de rayos y los valores de intensidad se combinan empezando por los nodos terminales del árbol y subiendo hasta las raíz. Los cálculos de intersección entre los objetos y los rayos en el método de trazado de rayos pueden reducirse mediante métodos de subdivisión espacial que comprueban las intersecciones entre los rayos y los objetos únicamente dentro de determinadas subregiones del espacio total. La técnica de trazado de rayos distribuido emplea múltiples rayos por píxel, asignando aleatoriamente diversos parámetros a los rayos, como la dirección y el tiempo. Esto proporciona un método preciso para modelar el brillo y la translucidez de las superficies, las aperturas finitas de cámara, las fuentes luminosas de tamaño finito, los efectos de sombra y el desenfoque de movimiento.

Los métodos de radiosidad proporcionan un modelado preciso de los efectos de reflexión difusa, calculando la transferencia de energía radiante entre los diversos parches superficiales de una escena. Si utiliza una técnica de refinamiento progresivo para acelerar los cálculos de radiosidad, tomando en consideración la transferencia de energía de un parche superficial en cada pasada. Pueden generarse escenas fotorrealistas utilizando una combinación de métodos de trazado de rayos y de radiosidad.

Un método rápido para aproximar los efectos globales de iluminación es el mapeado de entorno. Con esta técnica, se utiliza una matriz de entorno para almacenar información sobre la intensidad de fondo de una escena. Esta matriz se mapea posteriormente sobre los objetos de una escena basándose en la dirección de visualización especificada.

El mapeado de fotones proporciona un modelo preciso y eficiente para los efectos de iluminación global en escenas complejas. Con esta técnica, se generan rayos aleatorios desde las fuentes luminosas y los efectos de iluminación de cada rayo se almacenan en un mapa de fotones, que separa la información de iluminación de la geometría de la escena. La precisión de los efectos de iluminación se incrementa a medida que aumentamos el número de rayos generados.

Pueden añadirse detalles a las superficies de los objetos utilizando caras poligonales, mapeado de texturas, mapeado de relieve o mapeado del sistema de referencia. Pueden superponerse pequeñas caras poligonales sobre superficies de mayor tamaño con el fin de generar diversos tipos de diseños. Alternativamente, pueden definirse patrones de textura en espacios unidimensionales, bidimensionales y tridimensionales, pudiendo emplearse dichos patrones para añadir una textura a una línea, a una superficie o a un volumen. El mapeado de texturas procedimental utiliza funciones para calcular las variaciones en los efectos de iluminación de los objetos. El mapeado de relieve es un mecanismo para modelar irregularidades de las superficies aplicando una función de relieve que perturba los vectores normales a la superficie. El mapeado del sistema de referencia es una extensión del mapeado de relieve que puede utilizarse para modelar las características de los materiales anisótropos, permitiendo aplicar variaciones horizontales a la superficie, además de las variaciones verticales típicas del mapeado de relieve.

La biblioteca básica de OpenGL contiene un amplio conjunto de funciones para especificar fuentes luminosas puntuales, los diversos parámetros del modelo básico de iluminación, el método de representación superficial que hay que utilizar, las rutinas de aproximación de semitonos que deben emplearse y los patrones matriciales de texturas que hay que aplicar a los objetos. Las Tablas 10.2 y 10.3 proporcionan un resumen de estas funciones OpenGL de iluminación, representación superficial y mapeado de texturas.

**TABLA 10.2. RESUMEN DE FUNCIONES OpenGL PARA ILUMINACIÓN Y REPRESENTACIÓN DE SUPERFICIES.**

<i>Función</i>	<i>Descripción</i>
<code>glLight</code>	Especifica un valor de propiedad de una fuente luminosa.
<code> glEnable (lightName)</code>	Activa una fuente luminosa.
<code>glLightModel</code>	Especifica valores para los parámetros de iluminación globales.
<code>glMaterial</code>	Especifica un valor para un parámetro óptico de una superficie.
<code>glFog</code>	Especifica un valor para un parámetro atmosférico; los efectos atmosféricos se activan con la función <code> glEnable</code> .
<code> glColor4f (R, G, B, A)</code>	Especifica un valor alpha para una superficie con el fin de simular la transparencia. En la función <code>glBlendFunc</code> , hay que asignar el valor <code>GL_SRC_ALPHA</code> al factor origen de la mezcla y el valor <code>GL_ONE_MINUS_SRC_ALPHA</code> al factor de destino de la mezcla.
<code>glShadeModel</code>	Especifica la representación superficial de Gouraud o la representación superficial monocroma.
<code>glNormal3</code>	Especifica un vector normal a la superficie.
<code> glEnable (GL NORMALIZE)</code>	Especifica que las normales a la superficie deben convertirse a vectores unitarios.
<code> glEnableClientState (GL_NORMAL_ARRAY)</code>	Activa las rutinas de procesamiento para una matriz de vectores normales a la superficie.
<code>glNormalPointer</code>	Crea una lista de vectores normales a la superficie que hay que asociar con una matriz de vértices.
<code> glEnable (GL DITHER)</code>	Activa las operaciones para aplicar las técnicas de representación superficial mediante patrones de aproximación de semitonos.

**TABLA 10.3.** RESUMEN DE FUNCIONES DE MAPEADO DE TEXTURAS EN OpenGL.

<i>Función</i>	<i>Descripción</i>
<code>glTexImage1D</code>	Especifica los parámetros para definir un espacio de texturas unidimensional (el textura se activa con <code> glEnable</code> ).
<code>glTexImage2D</code>	Especifica los parámetros para definir un espacio de texturas bidimensional.
<code>glTexImage3D</code>	Especifica los parámetros para definir un espacio de texturas tridimensional.
<code>glTexParameter</code>	Especifica los parámetros para las rutinas de mapeado de texturas.
<code>glTexCoord</code>	Especifica el valor para una coordenada de textura en un espacio de texturas unidimensional, bidimensional, tridimensional o tetradimensional.
<code>glTexEnv</code>	Especifica los parámetros de entorno de las texturas, como el color del mezcla para el mapeado de texturas.
<code>glCopyTexImage</code>	Copia un bloque de colores de píxel del búfer de imagen para utilizarlo como patrón de textura.
<code>glCopyTexSubImage</code>	Copia un bloque de colores de píxel del búfer de imagen para utilizarlo como subpatrón de textura.
<code>glTexCoordPointer</code>	Especifica las coordenadas de textura en una lista asociada con una lista de vértices.
<code>glBindTexture</code>	Asigna un nombre al patrón de texturas y se utiliza también para activar un patrón nominado.
<code>glDeleteTextures</code>	Elimina una lista de texturas nominadas.
<code> glGenTextures</code>	Genera automáticamente nombres para las texturas.
<code>glIsTexture</code>	Comando de consulta para determinar si ya existe una textura nominada.
<code>glTexSubImage</code>	Crea un subpatrón de textura.
<code>gluBuild*Mipmaps</code>	Generación automática de patrones de reducción de texturas para espacios de texturas unidimensionales, bidimensionales o tridimensionales.
<code>gluBuild*MipmapLevels</code>	Generación automática de patrones de reducción de texturas para un nivel especificado en un espacio de texturas unidimensionales, bidimensionales o tridimensionales.
<code>glGetTexLevelParameter</code>	Consulta el sistema para determinar si puede admitirse un determinado valor de un parámetro de textura.
<code>gluQuadricTexture</code>	Activa o desactiva el texturado para superficies cuádricas.

## REFERENCIAS

Los modelos básicos de iluminación y las técnicas de representación superficial se explican en Gouraud (1971) y Phong (1975), Freeman (1980), Bishop y Wiemer (1986), Birn (2000), Akenine-Möller y Haines (2002) y Olano, Hart, Heidrich y McCool (2002). Los algoritmos de implementación para los modelos de ilu-

minación y los métodos de representación se presentan en Glassner (1990), Arvo (1991), Kirk (1992), Heckbert (1994), Paeth (1995) y Sakaguchi, Kent y Cox (2001). Los métodos de aproximación por semitonos se tratan en Velho y Gomes (1991). Para obtener más información sobre los mecanismos de aleatorización ordenada, difusión de errores y difusión de puntos, consulte Knuth (1987).

Los procedimientos de trazado de rayos se tratan en Whitted (1980), Amanatides (1984), Cook, Porter y Carpenter (1984), Kay y Kajiya (1986), Arvo y Kirk (1987), Quek y Hearn (1988), Glassner (1989), Shirley (1990 y 2000) y Koh y Hearn (1992). Los algoritmos relativos a los métodos de radiosidad pueden consultarse en Goral, Torrance, Greenberg y Battaile (1984), Cohen y Greenberg (1985), Cohen, Chen, Wallace y Greenberg (1988), Wallace, Elmquist y Haines (1989), Chen, Rushmeier, Miller y Turner (1991), Dorsey, Sillion y Greenberg (1991), Sillion, Arvo, Westin y Greenberg (1991), He, Heynen, Phillips, Torrance, Salesin y Greenberg (1992), Cohen y Wallace (1993), Lischinski, Tampieri y Greenberg (1993). Schoeneman, Dorsey, Smits, Arvo y Greenberg (1993) y Silicon y Puech (1994). Los algoritmos de mapeado de fotones se explican en Jensen (2001). Los métodos y aplicaciones de mapeado de texturas se analizan en Williams (1983), Segal, Korobkin, van Widenfelt, Foran y Haeberli (1992) y Demers (2002). En Glassner (1995) podrá encontrar una explicación general sobre los temas de propagación de la energía, ecuaciones de transferencia, procesos de representación y percepción humana de la luz y del color.

En Woo, Neider, Davis y Shreiner (1999) se presentan ejemplos adicionales de programación utilizando las funciones de iluminación y representación de OpenGL. También hay disponibles ejemplos de programación para las funciones de iluminación, representación y texturado de OpenGL en el sitio web tutorial de Nate Robins: <http://www.cs.utah.edu/~narobins/opengl.html>. Por último, en Shreiner (2000) puede encontrar un listado completo de las funciones de iluminación y representación de OpenGL.

## EJERCICIOS

---

- 10.1 Escribir una rutina para implementar la Ecuación 10.12 para reflexión difusa utilizando una única fuente de iluminación puntual y un método de representación constante para las caras de un tetraedro. La descripción del objeto se hará mediante tablas de polígonos, incluyendo los vectores normales a la superficie para cada una de las caras poligonales. Otros parámetros adicionales de entrada incluyen la intensidad de luz ambiente, la intensidad de la fuente luminosa y los coeficientes de reflexión superficial. Toda la información de coordenadas puede especificarse directamente en el sistema de referencia de visualización.
- 10.2 Modifique la rutina del Ejercicio 10.1 para representar las caras poligonales de una superficie esférica teselada.
- 10.3 Modifique la rutina del Ejercicio 10.2 para mostrar la superficie esférica utilizando un método de representación superficial de Gouraud.
- 10.4 Modifique la rutina del Ejercicio 10.3 para mostrar la superficie esférica utilizando un método de representación superficial de Phong.
- 10.5 Escribir una rutina para implementar la Ecuación 10.17 para las reflexiones difusa y especular utilizando una única fuente de iluminación puntual y mecanismo de representación superficial de Gouraud para las caras poligonales de una superficie esférica teselada. La descripción del objeto se hará mediante tablas de polígonos, incluyendo los vectores normales a la superficie para cada una de las caras poligonales. Los valores adicionales que se suministraran como entrada incluyen la intensidad de luz ambiente, la intensidad de la fuente luminosa, los coeficientes de reflexión superficial y el parámetro de reflexión especular. Toda la información de coordenadas puede especificarse directamente en el sistema de referencia de visualización.
- 10.6 Modifique la rutina del ejercicio anterior para mostrar las caras poligonales utilizando un método de representación superficial de Phong..
- 10.7 Modifique la rutina del ejercicio anterior para incluir una función de atenuación lineal de la intensidad.
- 10.8 Modifique la rutina del ejercicio anterior para incluir en la escena dos fuentes luminosas.

- 10.9 Modifique la rutina del ejercicio anterior de modo que la superficie esférica se visualice a través de un panel de cristal.
- 10.10 Explique las diferencias que cabría esperar ver en la apariencia de las reflexiones especulares modeladas con  $(\mathbf{N} \cdot \mathbf{H})^{n_s}$ , comparadas con las reflexiones especulares modeladas con  $(\mathbf{V} \cdot \mathbf{R})^{n_s}$ .
- 10.11 Verifique que  $2\alpha = \phi$  en la Figura 10.22 cuando todos los vectores son coplanares, pero que, en general,  $2\alpha \neq \phi$ .
- 10.12 Explique cómo pueden combinarse los diferentes métodos de detección de superficies visibles con un modelo de intensidad para la visualización de un conjunto de poliedros con superficies opacas.
- 10.13 Explique cómo pueden modificarse los diversos métodos de detección de superficies visibles para procesar objetos transparentes. ¿Hay algún método de detección de superficies visibles que no pueda manejar las superficies transparentes?
- 10.14 Especifique un algoritmo, basándose en uno de los métodos de detección de superficies visibles, que permita identificar las áreas de sombra sobre una escena iluminada por una fuente puntual distante.
- 10.15 ¿Cuántos niveles de intensidad pueden mostrarse mediante aproximaciones de semitonos utilizando cuadrículas de píxeles de tamaño  $n$  por  $n$ , si cada píxel puede visualizarse con  $m$  intensidades diferentes?
- 10.16 ¿Cuántas combinaciones de color distintas pueden generarse mediante aproximaciones de semitonos en un sistema RGB de dos niveles con una cuadrícula de 3 por 3 píxeles?
- 10.17 Escriba una rutina para mostrar un conjunto dado de variaciones de intensidad superficial, utilizando aproximaciones de semitonos con cuadrículas de 3 por 3 píxeles y dos niveles de intensidad (0 y 1) por píxel.
- 10.18 Escriba una rutina para generar matrices de aleatorización ordenada utilizando la relación de recurrencia de la Ecuación 10.48.
- 10.19 Escriba un procedimiento para mostrar una matriz dada de valores de intensidad utilizando el método de aleatorización ordenada.
- 10.20 Escriba un procedimiento para implementar el algoritmo de difusión de errores para una matriz dada  $m$  por  $n$  de valores de intensidad.
- 10.21 Escriba un programa para implementar el algoritmo básico de trazado de rayos para una escena que contenga una única esfera situada por encima de un cuadrado de tierra con un patrón ajedrezado. La escena debe iluminarse con una única fuente puntual situada en la posición de visualización.
- 10.22 Escriba un programa para implementar el algoritmo básico de trazado de rayos para una escena que contenga cualquier disposición especificada en esferas y caras poligonales iluminadas por un conjunto dado de fuentes luminosas puntuales.
- 10.23 Escriba un programa para implementar el algoritmo básico de trazado de rayos utilizando métodos de subdivisión espacial para cualquier disposición especificada de esferas y caras poligonales iluminadas por un conjunto dado de fuentes luminosas puntuales.
- 10.24 Escriba un programa para implementar las siguientes características del mecanismo de trazado de rayos distribuido: muestra de píxel con 16 rayos por píxel ajustados mediante fluctuación, direcciones de reflexión distribuidas (brillo), direcciones de refracción distribuidas (translucidez) y fuentes luminosas de tamaño finito.
- 10.25 Diseñe un algoritmo para modelar el desenfoque de movimiento de un objeto móvil utilizando trazado de rayos distribuido.
- 10.26 Implemente el algoritmo básico de radiosidad para representar las superficies interiores de un rectángulo cuando una de las caras interiores del rectángulo es una fuente luminosa.
- 10.27 Diseñe un algoritmo para implementar el método de radiosidad basado en refinamiento progresivo.
- 10.28 Escriba una rutina para transformar un mapa de entorno en la superficie de una esfera.
- 10.29 Escriba un programa para mapear un patrón de textura determinado sobre cualquier cara de un cubo.
- 10.30 Modifique el programa del ejercicio anterior de modo que el patrón sea mapeado sobre una cara de un tetraedro.

- 10.31 Modifique el programa del ejercicio anterior de modo que el patrón sea mapeado sobre una sección especificada de una superficie esférica.
- 10.32 Escriba un programa para mapear un patrón de texturas unidimensional sobre una cara especificada de un cubo, en forma de una banda diagonal.
- 10.33 Modifique el programa del ejercicio anterior de modo que la textura unidimensional sea mapeada sobre la superficie de una esfera, dados dos puntos de la superficie esférica.
- 10.34 Dada una superficie esférica, escriba el procedimiento de mapeado de relieve para simular la superficie rugosa de una naranja.
- 10.35 Escriba una rutina de mapeado de relieve para producir variaciones en las normales a la superficie de acuerdo con cualquier función de relieve especificada.
- 10.36 Escriba un programa OpenGL para mostrar una escena que contenga una esfera y un tetraedros iluminados por dos fuentes luminosas. Una de ellas es una fuente local de color rojo y la otra es una fuente de luz blanca distante. Especifique los parámetros de superficie tanto para la reflexión difusa como especular con representación superficial de Gouraud y aplique una función cuadrática de atenuación de la intensidad.
- 10.37 Modifique el programa del ejercicio anterior de modo que la única fuente local de color rojo se sustituya por dos fuentes de tipo foco: una roja y otra azul.
- 10.38 Modifique el programa del ejercicio anterior para añadir una atmósfera llena de humo a la escena.
- 10.39 Modifique el programa del ejercicio anterior de modo que la escena se visualice a través de un panel de cristal semitransparente.
- 10.40 Escriba un programa OpenGL completo para mostrar un conjunto de líneas diagonales utilizando diversos patrones de textura unidimensional, como en a Figura 10.114.
- 10.41 Escriba un programa utilizando un patrón de texturas bidimensional de OpenGL con el fin de mostrar un patrón ajedrezado blanco y negro sobre un fondo azul.
- 10.42 Modifique el programa del ejercicio anterior de modo que el patrón ajedrezado tenga cuadrados rojos y azules y el fondo sea de color blanco.
- 10.43 Escriba un programa utilizando un patrón de textura bidimensional de OpenGL para mostrar un rectángulo blanco con un conjunto de bandas diagonales rojas equiespaciadas. Defina como color de fondo el azul.
- 10.44 Modifique el programa del ejercicio anterior para mapear el patrón de textura sobre la superficie de una esfera.
- 10.45 Modifique el programa del ejercicio anterior para mapear el patrón de textura sobre la superficie de la tetera GLUT.



# Métodos interactivos de entrada e interfaces gráficas de usuario



Entrada interactiva dentro del entorno de realidad virtual denominado CAVE de NCSA, que está formado por tres paredes verticales, un suelo, un techo y un sistema estereoscópico de proyección. (*Cortesía de National Center for Supercomputing Applications, Universidad de Illinois en Urbana-Champaign.*)

- |   |   |
|---|---|
| <b>11.1</b> Datos de entrada gráficos                           | <b>11.5</b> Entornos de realidad virtual                              |
| <b>11.2</b> Clasificación lógica de los dispositivos de entrada | <b>11.6</b> Funciones OpenGL para dispositivos de entrada interactiva |
| <b>11.3</b> Funciones de entrada para datos gráficos            | <b>11.7</b> Funciones de menú OpenGL                                  |
| <b>11.4</b> Técnicas interactivas de construcción de imágenes   | <b>11.8</b> Diseño de una interfaz gráfica de usuario                 |
|   | <b>11.9</b> Resumen   |

Aunque podemos construir programas y proporcionar datos de entrada utilizando los métodos e instrucciones de programa explicados en los capítulos anteriores, a menudo resulta útil poder especificar interactivamente las entradas gráficas. Durante la ejecución de un programa, por ejemplo, puede que queramos cambiar el punto de vista o la ubicación de un objeto de una escena apuntando una posición de la pantalla, o bien puede que queramos variar los parámetros de una animación utilizando selecciones en un menú. En las aplicaciones de diseño, las coordenadas de los puntos de control para la construcción de *splines* se seleccionan interactivamente y a menudo se construyen imágenes utilizando métodos de dibujo interactivo. Son diversos los tipos de datos utilizados por los programas gráficos y se han desarrollados numerosos métodos de entrada interactiva para procesar dichos datos. Además, las interfaces de los sistemas utilizan ahora ampliamente los gráficos interactivos, incluyendo ventanas de visualización, iconos, menús y un ratón u otros dispositivos de control del cursor.

## 11.1 DATOS DE ENTRADA GRÁFICOS

---

Los programas gráficos utilizan diversos tipos de datos de entrada, como posiciones de coordenadas, valores de atributo, especificaciones de cadenas de caracteres, valores de transformación geométrica, condiciones de visualización y parámetros de iluminación. Muchos paquetes gráficos, incluyendo los estándares ISO y ANSI, proporcionan un amplio conjunto de funciones de entrada para procesar tales datos. Pero los procedimientos de entrada requieren de la interacción con los gestores de ventanas y con dispositivos hardware específicos. De ahí que algunos sistemas gráficos, particularmente aquellos que proporcionan principalmente funciones independientes del dispositivo, incluyen a menudo un número relativamente pequeño de procedimientos interactivos para el tratamiento de datos de entrada. Una organización bastante común para los procedimientos de entrada de un paquete gráfico consiste en clasificar las funciones de acuerdo con el tipo de los datos que cada función tiene que procesar. Este esquema permite introducir cualquier clase de datos mediante cualquier dispositivo físico, como por ejemplo un teclado o un ratón, aunque la mayoría de los dispositivos de entrada suelen manejar mejor algunos tipos de datos que otros.

## 11.2 CLASIFICACIÓN LÓGICA DE LOS DISPOSITIVOS DE ENTRADA

---

Cuando clasificamos las funciones de entrada de acuerdo con el tipo de los datos, los dispositivos utilizados para proporcionar los datos especificados se denominan **dispositivo lógico de entrada** para dicho tipo de datos. Las clasificaciones estándar para los datos de entrada lógicos son:

<b>LOCALIZADOR</b>	- Un dispositivo para especificar una posición de coordenadas.
<b>TRAZO</b>	- Un dispositivo para especificar un conjunto de posiciones de coordenadas.
<b>CADENA</b>	- Un dispositivo para especificar entrada textual.
<b>EVALUADOR</b>	- Un dispositivo para especificar un valor escalar.
<b>ELECCIÓN</b>	- Un dispositivo para elegir una opción de menú.
<b>SELECTOR</b>	- Un dispositivo para seleccionar un componente de una imagen.

## Dispositivos localizadores

La selección interactiva de un punto descrito por sus coordenadas se suele llevar a cabo situando el cursor de la pantalla en una determinada ubicación dentro de una escena, aunque también podrían utilizarse en determinadas aplicaciones otros métodos, como por ejemplo opciones de menú. Podemos utilizar un ratón, un *joystick*, una *trackball*, una *spaceball*, un ratón de bola, un dial, un cursor de mano o un lápiz digitalizador para el posicionamiento del cursor dentro de la pantalla. Y también pueden emplearse diversos botones, teclas o commutadores para indicar las opciones de procesamiento para la ubicación seleccionada.

Los teclados se emplean para los datos de localización de diversas maneras. Un teclado de propósito general suele tener cuatro teclas de control de cursor que mueven el cursor hacia arriba, hacia abajo, hacia la izquierda o hacia la derecha de la pantalla. Con otras cuatro teclas adicionales, podemos también mover el cursor diagonalmente. El movimiento rápido del cursor se consigue manteniendo apretada la tecla de cursor seleccionada. En ocasiones, los teclados incluyen un *joystick*, un *joydisk*, una *trackball* o un ratón de bola para posicionar un cursor en pantalla. En algunas aplicaciones, puede que resulte también conveniente utilizar un teclado para escribir valores numéricos u otros códigos que indiquen los valores de las coordenadas.

También se han utilizado otros dispositivos, como los lápices luminosos, para introducir interactivamente coordenadas en un sistema. Pero los lápices luminosos registran las posiciones de pantalla detectando la luz emitida por los fósforos de la misma, lo que requiere procedimientos especiales de implementación.

## Dispositivos de trazo

Esta clase de dispositivos lógicos se utiliza para introducir una secuencia de coordenadas y los dispositivos físicos empleados para generar las entradas de tipo localizador se utilizan también como dispositivos de trazo. El movimiento continuo de un ratón, de una *trackball*, de un *joystick* o de un cursor de mano se traduce en una serie de coordenadas de entrada. La table gráfica es uno de los dispositivos de trazo más comunes. Puede usarse la activación de botones para colocar la tableta en modo «continuo». A medida que se mueve el cursor por la superficie de la tableta, se genera un flujo de valores de coordenadas. Este procedimiento se utiliza en los sistemas de dibujo para generar imágenes utilizando diversos trazos de pincel. Los sistemas de ingeniería también utilizan este procedimiento para trazar y digitalizar planos.

## Dispositivos de cadena de caracteres

El principal dispositivo físico utilizado para la introducción de cadenas de caracteres es el teclado. Las cadenas de caracteres en las aplicaciones infográficas se utilizan normalmente para etiquetar los dibujos o gráficos.

Pueden usarse también otros dispositivos físicos para generar patrones de caracteres en aplicaciones especiales. Pueden dibujarse caracteres individuales en la pantalla utilizando un dispositivo de trazo o un dispositivo localizador. A continuación, un programa de reconocimiento de patrones interpreta los caracteres utilizando un diccionario almacenado de patrones predefinidos.

## Dispositivos evaluadores

Podemos emplear las entradas de evaluación en los programas gráficos para especificar valores escalares en las transformaciones geométricas, parámetros de visualización y parámetros de iluminación. En algunas apli-

caciones, las entradas de tipo escalar se utilizan también para establecer parámetros físicos tales como la temperatura, la tensión o factores de carácter mecánico.

Un dispositivo físico típico utilizado para proporcionar entradas de evaluación son los paneles de diales de control. Las posiciones de los diales se calibran para generar valores numéricos dentro de un rango predefinido. Una serie de potenciómetros rotatorios convierten la rotación del dial en una tensión correspondiente, que a continuación se traduce en un número comprendido dentro de un rango escalar predefinido, como por ejemplo entre  $-10.5$  y  $25.5$ . En lugar de utilizar diales, en ocasiones se emplean potenciómetros deslizantes para convertir movimientos lineales en valores escalares.

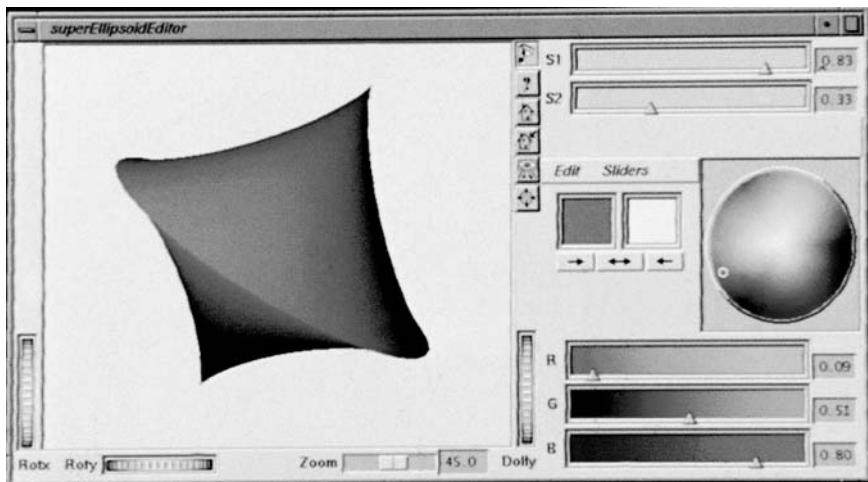
Puede utilizarse como dispositivo evaluador cualquier teclado que disponga de un conjunto de teclas numéricas, aunque los diales y los potenciómetros deslizantes son más eficientes para la introducción rápida de valores.

Los *joysticks*, las *trackballs*, las tabletas y otros dispositivos interactivos pueden adaptarse para introducir valores interpretando la presión o el movimiento del dispositivo en relación con un rango escalar. Para una determinada dirección de movimiento, como por ejemplo de izquierda a derecha, pueden introducirse valores escalares crecientes; el movimiento en la dirección opuesta reduce el valor escalar de entrada. Los valores seleccionados suelen presentarse en pantalla para que el usuario pueda verificarlos.

Otra técnica para proporcionar entradas de evaluación consiste en mostrar representaciones gráficas de deslizadores, botones, diales rotatorios y menús en el monitor de vídeo. La Figura 11.1 ilustra algunas posibilidades para la representación de este tipo de controles. El posicionamiento del cursor mediante un ratón, *joystick*, *spaceball* u otro dispositivo permite seleccionar un valor en uno de esos controles virtuales. Como mecanismo de realimentación para el usuario, los colores seleccionados se muestran mediante barras de color y el valor escalar seleccionado se indica en una pequeña ventana próxima a cada control.

## Dispositivos de elección

Los menús se utilizan normalmente en los programas gráficos para seleccionar opciones de procesamiento, valores de parámetro y formas de los objetos que haya que utilizar para construir una imagen. Los dispositi-



**FIGURA 11.1.** Representación de controles gráficos para la introducción de datos de evaluación. En esta imagen, se proporcionan representaciones de controles deslizantes y de diales para la selección de los valores de los parámetros  $s1$  y  $s2$  de una superelipse, además de para la introducción de componentes de color RGB, ángulos de rotación y parámetros de escala. Alternativamente, puede posicionarse un pequeño círculo sobre la rueda de colores con el fin de seleccionar simultáneamente las tres componentes RGB. También pueden utilizarse las teclas de flecha del teclado y determinados botones con el fin de realizar pequeños cambios en un valor escalar seleccionado.

vos de elección comúnmente utilizados para seleccionar una opción de menú son los dispositivos de posicionamiento del cursor, como por ejemplo un ratón, una *trackball*, el teclado, un panel de control o un panel de botones.

A menudo se utilizan las teclas de función de un teclado o paneles de botones independientes para introducir las selecciones de menú. Cada botón o tecla de función se programa para seleccionar una operación o valor concreto, aunque en ocasiones se incluyen en los dispositivos de entrada botones o teclas preconfigurados.

Para la selección en pantalla de las opciones de menú mostradas, se utiliza un dispositivo de posicionamiento del cursor. Cuando se selecciona una posición ( $x, y$ ) del cursor en pantalla, dicha posición se compara con las extensiones de coordenadas de cada uno de los elementos de menú mostrados. De esta forma, se seleccionará un elemento de menú que tenga límites verticales y horizontales  $x_{\min}, x_{\max}, y_{\min}$  y  $y_{\max}$  de los valores de coordenadas si las coordenadas introducidas satisfacen las desigualdades:

$$x_{\min} \leq x \leq x_{\max}, y_{\min} \leq y \leq y_{\max} \quad (11.1)$$

Para los menús de mayor tamaño en los que se muestre un número relativamente pequeño de opciones, se utiliza comúnmente un panel táctil. La posición seleccionada de la pantalla se compara con las extensiones de coordenadas de las opciones de menú individuales, con el fin de determinar el proceso que hay que ejecutar.

Otros métodos alternativos para la realización de elecciones dentro de una aplicación incluyen el teclado y los dispositivos de entrada de voz. Un teclado estándar puede utilizarse para escribir comandos u opciones de menú. Para este método de introducción de elecciones, resulta útil disponer de algún tipo de formato abreviado; los elementos de menú pueden estar numerados o tener nombres cortos identificativos. Puede utilizarse un esquema de codificación similar con los sistemas de entrada vocal. La entrada vocal resulta particularmente útil cuando el número de opciones es pequeño (20 o menos).

## Dispositivos de selección

Los dispositivos de selección se emplean para seleccionar una parte de una escena que haya que transformar o evitar de alguna manera. Pueden utilizarse diversos métodos distintos para seleccionar un componente de una escena visualizada, y cualquier mecanismo de entrada que se utiliza para este propósito se clasificará como un dispositivo de selección. Normalmente, las operaciones de selección se realizan posicionando el cursor en la pantalla. Utilizando un ratón, un *joystick* o un teclado, por ejemplo, podemos realizar las selecciones posicionando el cursor en la pantalla y presionando un botón o una tecla para registrar las coordenadas del píxel. Esta posición de la pantalla puede entonces usarse para seleccionar un objeto completo, una faceta de una superficie teselada, una arista de un polígono o un vértice. Otros métodos de selección incluyen los esquemas de resalte, los mecanismos de selección de objetos por su nombre o una combinación de algunos de estos métodos.

Utilizando la técnica de posicionamiento del cursor, un procedimiento de selección podría mapear una posición de pantalla seleccionada sobre una ubicación en coordenadas universales utilizando las transformaciones inversas de visualización y geométrica que hubieran sido especificadas para la escena. Entonces, la posición en coordenadas universales puede compararse con la extensión de coordenadas de los objetos. Si la posición seleccionada se encuentra dentro de la extensión de coordenadas de un único objeto, habremos identificado el objeto seleccionado. Entonces, puede usarse el nombre del objeto, sus coordenadas u otra información relativa al mismo para aplicar las operaciones deseadas de transformación o de edición. Pero si la posición seleccionada se encuentra dentro de la extensión de coordenadas de dos o más objetos, será necesario realizar comprobaciones adicionales. Dependiendo del tipo de objeto que haya que seleccionar y de la complejidad de una escena, puede que se requieran varios niveles de búsqueda para identificar el objeto seleccionado. Por ejemplo, si estamos tratando de seleccionar una esfera cuya extensión de coordenadas se solapa con la de algún otro objeto tridimensional, la posición seleccionada podría compararse con las extensiones de coordenadas de las facetas superficiales individuales de los dos objetos. Si esta comprobación no arrojara resultados concluyentes, podrían comprobarse las extensiones de coordenadas de los segmentos de línea individuales.

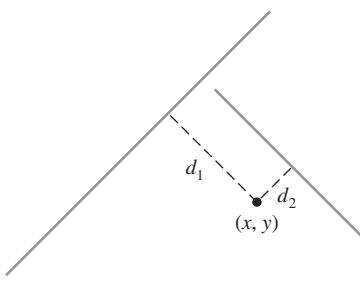
Cuando las comprobaciones de extensión de coordenadas no permitan identificar únicamente el objeto seleccionado, pueden calcularse las distancias desde la posición seleccionada hasta los segmentos de línea individuales. La Figura 11.2 ilustra la selección de una posición que se encuentra dentro de las extensiones de coordenadas de dos segmentos de línea. Para un segmento de línea bidimensional cuyas coordenadas de los extremos sean  $(x_1, y_1)$  y  $(x_2, y_2)$ , el cuadrado de la distancia perpendicular desde una posición seleccionada  $(x, y)$  hasta la línea se calcula como:

$$d^2 = \frac{[\Delta x(y - y_1) - \Delta y(x - x_1)]^2}{\Delta x^2 + \Delta y^2} \quad (11.2)$$

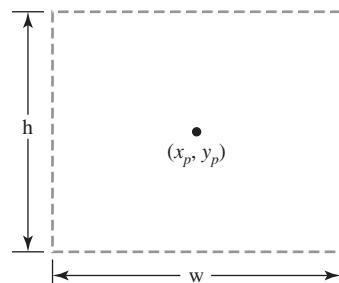
donde  $\Delta x = x_2 - x_1$  y  $\Delta y = y_2 - y_1$ . Se han propuesto también otros métodos, como la comparación de las distancias a los puntos extremos, para simplificar las operaciones de selección de líneas.

Los procedimientos de selección pueden simplificarse si no se llevan a cabo las comprobaciones de extensión de coordenadas para las facetas de superficie y los segmentos de línea de un objeto. Cuando la posición de selección se encuentre dentro de las extensiones de coordenadas de dos o más objetos, los procedimientos de selección pueden simplemente devolver una lista de todos los objetos candidatos.

Otra técnica de selección consiste en asociar una **ventana de selección** con una posición seleccionada del cursor. La ventana de selección estará centrada en la posición del cursor, como se muestra en la Figura 11.3 y se utilizarán procedimientos de recorte para determinar qué objetos se intersectan con la ventana de selección. Para la selección de líneas, podemos asignar valores muy pequeños a las dimensiones  $w$  y  $h$  de la ventana de selección, de modo que sólo intersecte un segmento de línea con la ventana de selección. Algunos paquetes gráficos implementan la selección tridimensional reconstruyendo una escena utilizando las transformaciones de visualización y proyección y empleando la ventana de selección como ventana de recorte. No se muestra ninguna imagen correspondiente a esta reconstrucción, si no que simplemente se aplican los procedimientos de recorte para determinar qué objetos se encuentran dentro del volumen de selección. Entonces, puede devolverse una lista de información para cada objeto contenido en el volumen de selección, con el fin de procesar los distintos objetos. Esta lista puede contener información tal como el nombre del objeto y su rango de profundidad, pudiendo emplearse el rango de profundidad para seleccionar el objeto más cercano dentro del conjunto de objetos contenidos en el volumen de selección. También pueden utilizarse técnicas de resalte para facilitar la selección. Una forma de hacer esto consiste en resaltar sucesivamente aquellos objetos cuyas extensiones de coordenadas se solapen con una posición de selección (o ventana de selección). A medida que se resalta cada objeto, el usuario puede ejecutar una acción de «rechazo» o «aceptación» utilizando los botones del teclado. La secuencia se detendrá cuando el usuario acepte como selección el objeto resaltado. La selección también podría llevarse a cabo simplemente resaltando sucesivamente todos los objetos de la escena sin seleccionar una posición del cursor. Esta secuencia de resalte puede iniciarse mediante un botón o tecla de función, pudiendo emplearse un segundo botón para detener el proceso cuando esté resaltado el objeto deseado. Si hubiera que ir pasando de esta manera a través de una lista muy grande de objetos, pueden



**FIGURA 11.2.** Distancias desde una posición seleccionada a dos segmentos de línea distintos.



**FIGURA 11.3.** Una ventana de selección con centro en  $(x_p, y_p)$ , anchura  $w$  y altura  $h$ .

utilizarse botones adicionales para acelerar el proceso de resalte. Un botón iniciaría un resalte rápido sucesivo de estructuras, un segundo botón se activaría para detener el proceso y un tercer botón se emplearía para ir lentamente hacia atrás en el proceso de resalte. Finalmente, puede presionarse un botón de parada para completar el procedimiento de selección.

Si los componentes de una imagen pueden seleccionarse por su nombre, podría emplearse la entrada de teclado para seleccionar un objeto. Este procedimiento es muy sencillo pero menos interactivo. Algunos paquetes gráficos permiten asignar nombres a los componentes de una imagen a diversos niveles, hasta llegar al nivel de las primitivas individuales. Pueden usarse nombres descriptivos para ayudar al usuario durante el proceso de selección, pero esta técnica tiene sus desventajas: generalmente es más lenta que la selección interactiva en pantalla y el usuario probablemente necesite que se presenten indicaciones en pantalla para recordar los diversos nombres de las estructuras.

## 11.3 FUNCIONES DE ENTRADA PARA DATOS GRÁFICOS

---

Los paquetes gráficos que utilizan la clasificación lógica de los dispositivos de entrada proporcionan diversas funciones para seleccionar dispositivos y clases de datos. Estas funciones permiten al usuario especificar las siguientes opciones:

- El modo de interacción de entrada para el programa gráfico y los dispositivos de entrada. Puede que sean el programa o los dispositivos los que inicien la introducción de los datos, o bien pueden operar ambos simultáneamente.
- La selección de un dispositivo físico que proporcione la entrada dentro de una clasificación lógica concreta (por ejemplo, una tableta utilizada como dispositivo de trazo).
- La selección del tiempo y el dispositivo de entrada para un conjunto concreto de valores de datos.

### Modos de entrada

Algunas funciones de entrada en un sistema gráfico interactivo se utilizan para especificar cómo deben interactuar el programa y los dispositivos de entrada. Un programa podría requerir las entradas en un momento concreto del procesamiento (modo de solicitud) o un dispositivo de entrada podría proporcionar de manera independiente datos de entrada actualizados (modo de muestreo), o el dispositivo podría almacenar de manera independiente todos los datos recopilados (modo de sucesos).

En el **modo de solicitud**, el programa de aplicación es quien inicia la introducción de los datos. Cuando se solicitan valores de entrada, el procesamiento se suspende hasta que se reciben los valores requeridos. Este modo de entrada se corresponde con la operación típica de entrada en un lenguaje de programación general. El programa y los dispositivos de entrada operan de manera alternativa. Los dispositivos se quedan en un estado de espera hasta que se realiza una solicitud de entrada, en cuyo momento el programa espera hasta que se entregan los datos.

En el **modo de muestreo**, el programa de aplicación y los dispositivos de entrada operan de manera independiente. Los dispositivos de entrada pueden estar operando al mismo tiempo que el programa procesa otros datos. Los nuevos valores obtenidos desde los dispositivos de entrada sustituyen a los valores de datos introducidos previamente. Cuando el programa requiera nuevos datos, muestreará los valores actuales que se hayan almacenado a partir de la entrada del dispositivo.

En el **modo de sucesos**, los dispositivos de entrada son quienes inician la introducción de datos en el programa de aplicación. De nuevo, el programa y los dispositivos de entrada operan concurrentemente, pero ahora los dispositivos de entrada suministran datos a una cola de entrada, también llamada cola de sucesos. Todos los datos de entrada se almacenan. Cuando el programa requiere nuevos datos, los extrae de la cola de datos.

Normalmente, puede haber varios dispositivos operando al mismo tiempo en los modos de muestreo y de sucesos. Alguno de ellos pueden estar operando en modo de muestreo mientras que otros operan en modo de sucesos. Por el contrario, en el modo de solicitud, sólo puede haber un dispositivo en cada momento suministrando las entradas.

Otras funciones de la biblioteca de entrada se utilizan para especificar los dispositivos físicos correspondientes a las distintas clases lógicas de datos. Los procedimientos de entrada en un paquete interactivo pueden implicar un procesamiento relativamente complicado para algunos tipos de entrada. Por ejemplo, para obtener una posición en coordenadas universales, los procedimientos de entrada deben procesar una ubicación de pantalla suministrada como entrada, aplicándole las transformaciones de visualización y otras transformaciones hasta llegar a la descripción original en coordenadas universales de la escena. Y este procesamiento también implica obtener información de las rutinas de gestión de las ventanas de visualización.

### Realimentación mediante eco

Normalmente, puede solicitarse a un programa interactivo de entrada que proporcione un eco de realimentación de los datos de entrada y de los parámetros asociados. Cuando se solicita un eco de los datos de entrada, éstos se muestran dentro de un área de pantalla especificada. La realimentación mediante eco puede incluir, por ejemplo, el tamaño de la ventana de selección, la distancia de selección mínima, el tipo y el tamaño de un cursor, el tipo de resalte que habría que emplear durante las operaciones de selección, el rango (mínimo y máximo) de las entradas de evaluación y la resolución (escala) de las entradas de evaluación.

### Funciones de retrollamada

En los paquetes gráficos independientes de los dispositivos, puede proporcionarse un conjunto limitado de funciones de entrada en una biblioteca auxiliar. Los procedimientos de entrada pueden entonces gestionarse como funciones de retrollamada (Sección 2.9) que interactúen con el software del sistema. Estas funciones especifican qué acciones debe tomar un programa cuando tenga lugar un suceso de entrada. Los sucesos de entrada típicos son el movimiento del ratón, la posición de un botón del ratón o la pulsación de un botón del teclado.

## 11.4 TÉCNICAS INTERACTIVAS DE CONSTRUCCIÓN DE IMÁGENES

---

Los paquetes gráficos suelen incorporar diversos métodos interactivos como ayuda para la construcción de imágenes. Puede que se proporcionen rutinas para posicionar objetos, aplicar restricciones, ajustar el tamaño de los objetos y diseñar formas y patrones.

### Métodos básicos de posicionamiento

Podemos seleccionar interactivamente un punto de coordenadas con algún dispositivo señalado que registre una ubicación en pantalla. Cómo se usa la posición dependerá de la opción seleccionada de procesamiento. El punto de coordenadas puede ser el extremo de un segmento de línea o utilizarse para posicionar algún objeto: por ejemplo, la ubicación de pantalla seleccionada podría hacer referencia a una nueva posición que designe el centro de una esfera. O bien, la ubicación puede utilizarse para especificar la posición de una cadena de texto, que podría comenzar en dicha ubicación o estar centrada en ella. Como ayuda adicional de posicionamiento, los valores numéricos correspondientes a las posiciones seleccionadas pueden suministrarse como eco en la pantalla. Utilizando como guía los valores de coordenada suministrados como eco, el usuario podría realizar pequeños ajustes interactivos en los valores de coordenadas utilizando diales, teclas de cursor u otros dispositivos.

## Arrastre de objetos

Otra técnica interactiva de posicionamiento consiste en seleccionar un objeto y arrastrarlo hasta una nueva ubicación. Utilizando un ratón, por ejemplo, posicionamos el cursor sobre el objeto, pulsamos un botón del ratón, movemos el cursor a una nueva posición y liberamos el botón. Entonces, el objeto se mostrará en la nueva ubicación del cursor. Usualmente, el objeto también se suele mostrar en las posiciones intermedias a medida que se mueve el cursor por la pantalla.

## Restricciones

Se denomina restricción a cualquier procedimiento destinado a alterar los valores de coordenadas introducidos con el fin de obtener una orientación o alineación particulares de un objeto. Por ejemplo, puede restringirse un segmento de línea de entrada para que sea horizontal o vertical, como se ilustra en las Figuras 11.4 y 11.5. Para implementar este tipo de restricción, comparamos los valores de coordenada de entrada de los dos puntos extremos. Si la diferencia en los valores  $y$  de los dos extremos es menor que la diferencia en los valores  $x$ , se mostrará una línea horizontal, mientras que en caso contrario se mostrará una línea vertical. La restricción horizontal-vertical resulta útil, por ejemplo, al construir diagramas de red y elimina la necesidad de posicionar de manera precisa las coordenadas de los extremos.

Pueden aplicarse otros tipos de restricciones a las coordenadas de entrada con el fin de producir diversas clases de alineaciones. Pueden restringirse las líneas para que tengan una pendiente concreta, como por ejemplo  $45^\circ$ , y las coordenadas de entrada pueden restringirse para que caigan a lo largo de una serie de trayectorias predefinidas, como por ejemplo arcos circulares.

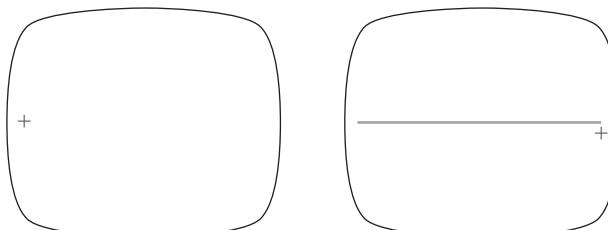
## Cuadrículas

Otro tipo de restricción es una cuadrícula rectangular que puede mostrarse en alguna parte de la pantalla. Si está activada la restricción de cuadrícula, las coordenadas de entrada se redondean a la intersección de cuadrícula más próxima. La Figura 11.6 ilustra el procedimiento de dibujo de una línea utilizando una cuadrícula. Cada una de las posiciones de cursor en este ejemplo se desplaza hasta el punto de intersección de la cuadrícula más cercano, trazándose una línea entre esas dos posiciones de cuadrícula. Las cuadrículas facilitan la construcción de los objetos, ya que puede unirse fácilmente una línea con otra previamente dibujada, seleccionando cualquier posición cerca de la intersección de cuadrícula correspondiente a uno de los extremos de la línea previa. Normalmente, el espacio entre las líneas de cuadrícula puede ajustarse, y también suele ser posible utilizar cuadrículas parciales o cuadrículas con diferente espacio en las diferentes áreas de la pantalla.

## Métodos de banda elástica

Los segmentos de línea y otras formas básicas pueden construirse y posicionarse utilizando métodos de banda elástica que permiten ampliar o contraer interactivamente los tamaños de los objetos. La Figura 11.7 ilustra uno de estos métodos de banda elástica para la especificación interactiva de un segmento de línea. En primer lugar, se selecciona una posición fija de la pantalla para uno de los extremos de la línea. Después, a medida que se mueve el cursor, la línea se muestra desde esa posición inicial hasta la posición actual del cursor. El segundo extremo de la línea será introducido cuando se pulse un botón o una tecla. Utilizando un ratón, construimos una línea de banda elástica mientras pulsamos una de las teclas del ratón; al liberarla, la visualización de la línea será completada.

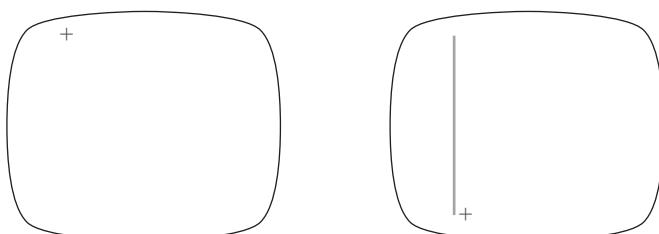
Podemos utilizar métodos de banda elástica para construir rectángulos, círculos y otros objetos. La Figura 11.8 ilustra la construcción de un rectángulo mediante un método de banda elástica y la Figura 11.9 muestra la construcción de un círculo. Podemos implementar estos mecanismos de construcción de banda elástica de diversas maneras; por ejemplo, la forma y el tamaño de un rectángulo pueden ajustarse moviendo de manera independiente únicamente la arista superior del rectángulo, o la arista inferior o una de las aristas laterales.



Se selecciona la posición del primer extremo

Se selecciona la posición del segundo extremo según un trayecto horizontal aproximado

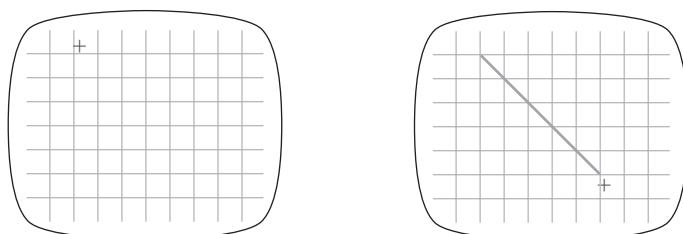
**FIGURA 11.4.** Restricción horizontal de líneas.



Se selecciona la posición del primer extremo

Se selecciona la posición del segundo extremo según un trayecto vertical aproximado

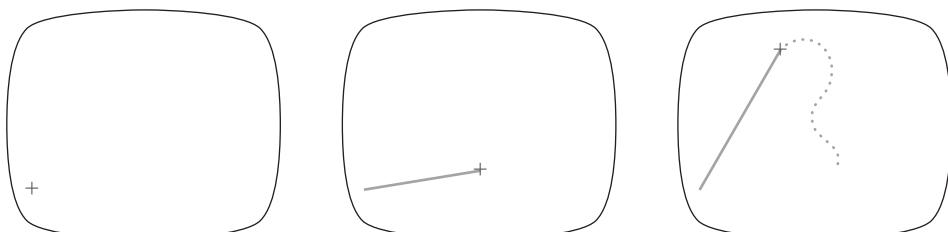
**FIGURA 11.5.** Restricción vertical de líneas.



Se selecciona la posición del primer extremo cerca de una intersección de la cuadrícula

Se selecciona una posición cerca de una segunda intersección de la cuadrícula

**FIGURA 11.6.** Construcción de un segmento de línea en el que los extremos están restringidos de forma que sólo pueden colocarse en las posiciones de intersección de la cuadrícula.

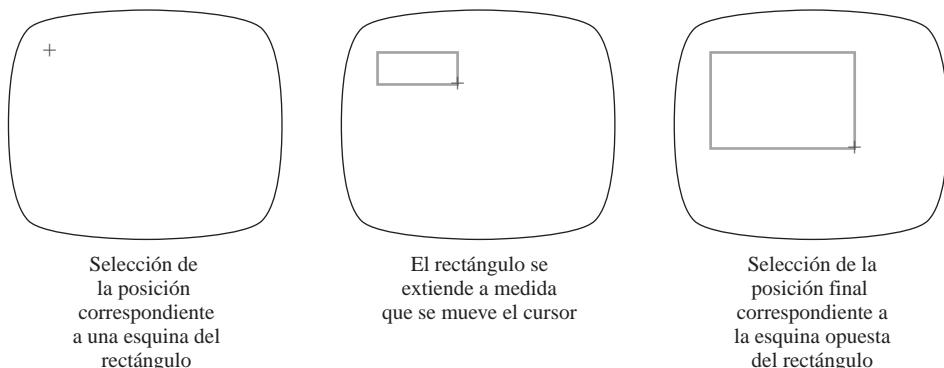


Selección del primer extremo de la línea

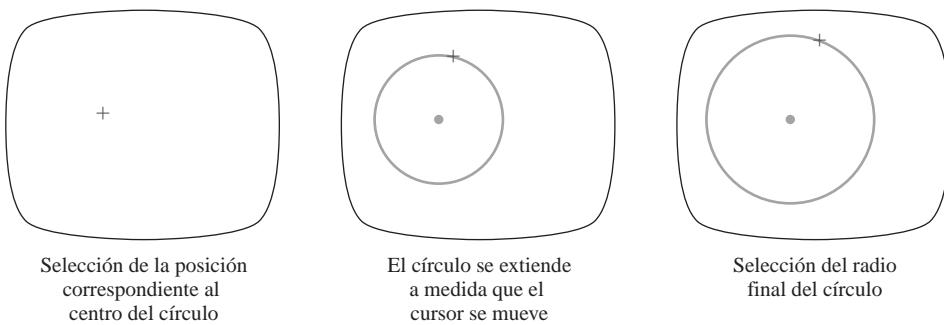
A medida que el cursor se mueve, una línea se extiende a partir del punto inicial

La línea sigue la posición del cursor hasta que se selecciona el segundo extremo

**FIGURA 11.7.** Método de banda elástica para la construcción y posicionamiento de un segmento de línea recta.



**FIGURA 11.8.** Método de banda elástica para la construcción de un rectángulo.

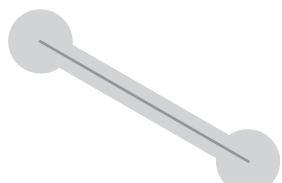


**FIGURA 11.9.** Construcción de un círculo utilizando un método de banda elástica.

## Campo de gravedad

En la construcción de figuras, en ocasiones surge la necesidad de conectar líneas en las posiciones de algunos extremos que no se encuentran situados en las intersecciones de la cuadrícula. Puesto que el posicionamiento exacto del cursor de pantallas sobre el punto de conexión puede resultar difícil, los paquetes gráficos pueden incluir procedimientos que conviertan cualquier posición de entrada situada cerca del segmento de línea a una posición sobre una línea utilizando un área de *campo de gravedad* alrededor de la línea. Cualquier posición seleccionada dentro del campo de gravedad de una línea se moverá («gravitará») hasta la posición más cercana situada sobre la línea. En la Figura 11.10 se ilustra mediante una región sombreada el área del campo de gravedad situado alrededor de una línea.

Los campos de gravedad alrededor de los extremos de la línea están agrandados para hacer que le resulte más fácil al diseñador conectar las líneas por los extremos. Las posiciones seleccionadas en una de las áreas circulares del campo de gravedad se verán atraídas hasta el extremo correspondiente a dicha área. El tamaño de los campos de gravedad se selecciona de forma que sea lo suficientemente grande como para ayudar en el posicionamiento, pero lo suficientemente pequeño como para reducir las posibilidades de solapamiento con otras líneas. Si se muestran muchas líneas, las áreas de gravedad pueden solaparse y puede resultar difícil



**FIGURA 11.10.** Campo de gravedad alrededor de una línea. Cualquier punto seleccionado en el área sombreada se desplaza a una posición sobre la línea.

especificar los puntos correctamente. Normalmente, los sistemas no muestran en pantalla las fronteras de estos campos de gravedad.

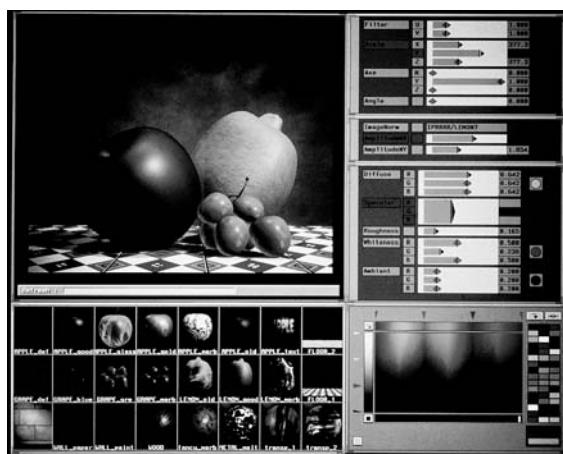
### Métodos interactivos de dibujo

Las opciones para la realización de bocetos y dibujos son de distintos tipos. Pueden generarse líneas rectas, polígonos y círculos utilizando los métodos explicados en las secciones previas. También pueden proporcionarse opciones de dibujo de curvas utilizando formas curvas estándar, como arcos circulares y *splines*, o empleando procedimientos de trazado a mano alzada. Las *splines* se construyen interactivamente especificando un conjunto de puntos de control o un boceto a mano alzada que proporcione la forma general de la curva. Entonces, el sistema hará encajar el conjunto de puntos con una curva polinómica. En el dibujo a mano alzada, las curvas se generan siguiendo la trayectoria de un lápiz sobre una tableta gráfica o el trayecto del cursor de pantalla sobre un monitor de vídeo. Una vez visualizada una curva, el diseñador puede alterar la forma de la curva ajustando las posiciones de puntos seleccionados situados a lo largo del trayecto curvo.

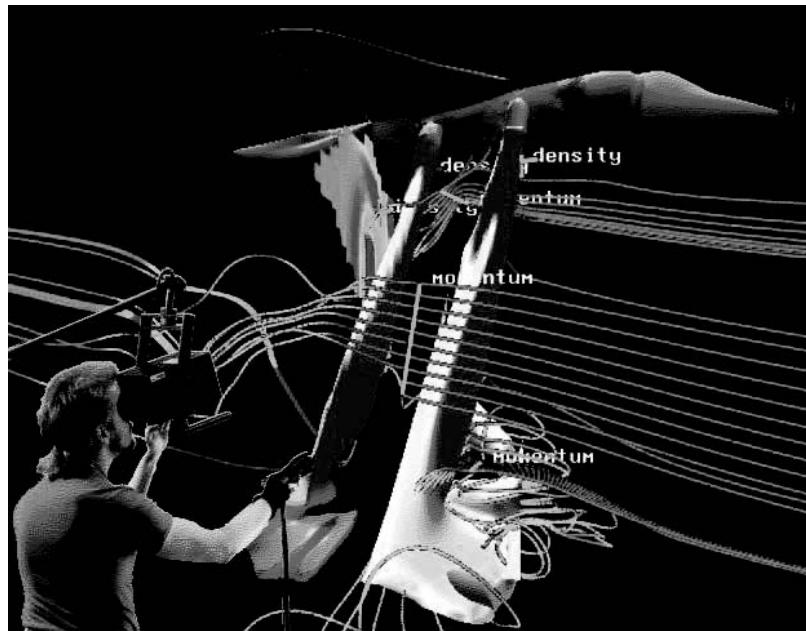
Los paquetes de dibujo suelen también incluir opciones para el ajuste de las anchuras de las líneas, de los estilos de las líneas y de otros atributos. Estas opciones se implementan mediante los métodos analizados en la Sección 4.5. En muchos sistemas hay también disponibles distintos estilos de pincel, patrones de pincel, combinaciones de colores, formas de objetos y patrones de textura superficial; todas estas opciones suelen ser muy comunes en los sistemas que funcionan como estaciones de trabajo para dibujo artístico. Algunos sistemas de dibujo varían el ancho de la línea y los trazos de pincel de acuerdo con la presión que la mano del artista ejerce sobre el lápiz. La Figura 11.11 muestra un sistema de ventanas y menús utilizado con un paquete de dibujo que permite a un artista seleccionar variaciones de una forma de objeto especificada, así como seleccionar diferentes texturas superficiales y condiciones de iluminación para la escena.

## 11.5 ENTORNOS DE REALIDAD VIRTUAL

En la Figura 11.12 se ilustra un entorno típico de realidad virtual. La entrada interactiva se lleva a cabo en este entorno mediante un electroguante (Sección 2.4), que es capaz de agarrar y desplazar los objetos que se muestran en una escena virtual. La escena generada por la computadora se muestra mediante un sistema de visiocasco (Sección 2.1) en forma de proyección estereográfica. Una serie de dispositivos de seguimiento calculan la posición y orientación del visiocasco y del electroguante en relación con las posiciones de los objetos de la escena. Con este sistema, un usuario puede moverse a través de la escena y reordenar las posiciones de los objetos con su electroguante.



**FIGURA 11.11.** Una captura de pantalla que muestra un tipo de interfaz de un paquete para dibujo artístico. (Cortesía de Thomson Digital Image.)



**FIGURA 11.12.** Utilizando un visiocasco estéreo, denominado BOOM (Fake Space Labs, Inc.) y un electroguante Dataglove (VPL, Inc.), un investigador manipula interactivamente una serie de sondas dentro del turbulento flujo que existe alrededor de un avión Harrier a reacción. El software ha sido desarrollado por Steve Bryson; y los datos han sido proporcionados por Harrier. (Cortesía de Sam Uselton, NASA Ames Research Center.)

Otro método para generar escenas virtuales consiste en mostrar proyecciones estereográficas sobre un monitor de barrido, visualizando las dos imágenes estereográficas en ciclos de refresco alternativos. Entonces, la escena puede observarse mediante unas gafas estereográficas. De nuevo, la manipulación interactiva de los objetos puede llevarse a cabo con un electroguante, utilizando un dispositivo de seguimiento para monitorear la posición y orientación del guante en relación con la posición de los objetos de la escena.

## 11.6 FUNCIONES OpenGL PARA DISPOSITIVOS DE ENTRADA INTERACTIVA

La entrada interactiva desde cualquier tipo de dispositivo en un programa OpenGL se gestiona mediante las rutinas GLUT (Utility Toolkit), porque estas rutinas necesitan comunicarse con un sistema de gestión de ventanas. En GLUT, disponemos de funciones para aceptar la entrada procedente de dispositivos estándar, como el teclado o el ratón, así como de tabletas, ratones de bola tridimensionales, cajas de botones y diales. Para cada dispositivo, especificamos un procedimiento (la función de retrollamada) que será el que haya que invocar cuando tenga lugar cualquier suceso de entrada en ese dispositivo. Estos comandos GLUT se insertan en el procedimiento `main` junto con las demás instrucciones GLUT. Además, puede usarse una combinación de funciones de la biblioteca básica y de la biblioteca GLU con la función de ratón de GLUT para capturar entradas.

### Funciones de ratón GLUT

Utilizamos la siguiente función para especificar («registrar») un procedimiento que será el que haya que llamar cuando el puntero del ratón se encuentre dentro de una ventana de visualización y un botón del ratón sea presionado o liberado:

```
glutMouseFunc (mouseFcn);
```

Este procedimiento de retrollamada para control del ratón, al que hemos denominado `mouseFcn`, tiene cuatro argumentos:

```
void mouseFcn (GLint button, GLint action, GLint xMouse,
               GLint yMouse)
```

Al parámetro `button` se le asigna una constante simbólica GLUT que denota uno de los tres botones del ratón, mientras que al parámetro `action` se le asigna una constante simbólica que especifica qué acción del botón podemos utilizar para disparar el suceso de activación del ratón. Los valores permitidos para `button` son `GLUT_LEFT_BUTTON`, `GLUT_MIDDLE_BUTTON` y `GLUT_RIGHT_BUTTON` (si sólo disponemos de un ratón de dos botones, usaremos únicamente las designaciones correspondientes al botón izquierdo y al botón derecho; con un ratón de un único botón, el único valor que podemos asignar al parámetro `button` es `GLUT_LEFT_BUTTON`). Al parámetro `action` se le pueden asignar las constantes `GLUT_DOWN` o `GLUT_UP`, dependiendo de si queremos iniciar una acción cuando pulsamos el botón del ratón o cuando lo liberemos. Cuando se invoca el procedimiento `mouseFcn`, se devuelve la ubicación del cursor del ratón dentro de la ventana de visualización, en forma de una pareja de coordenadas (`xMouse`, `yMouse`). Esta ubicación es relativa a la esquina superior izquierda de la ventana de visualización, de modo que `xMouse` será la distancia en píxeles desde el borde izquierdo de la ventana de visualización e `yMouse` será la distancia en píxeles desde el borde superior de la ventana de visualización.

Activando un botón del ratón mientras el cursor se encuentra dentro de la ventana de visualización, podemos seleccionar una posición para mostrar una primitiva, como por ejemplo un único punto, un segmento de línea o un área de relleno. También podemos utilizar un ratón como dispositivo de selección, comparando la posición en pantalla devuelta con las extensiones de coordenadas de los objetos visualizados en la escena. Sin embargo, OpenGL proporciona otras rutinas para utilizar el ratón como dispositivo de selección, y hablaremos de esas rutinas en una sección posterior.

Como ejemplo simple de utilización de la rutina `glutMouseFunc`, el siguiente programa dibuja un punto rojo, con un tamaño de punto igual a 3, en la posición del cursor dentro de la ventana de visualización; para dibujar el punto, debemos presionar el botón izquierdo del ratón. Puesto que el origen de coordenadas para las funciones primitivas OpenGL es la esquina inferior izquierda de la ventana de visualización, será necesario invertir el valor `yMouse` devuelto en el procedimiento `mousePtPlot`.

```
#include <GL/glut.h>

GLsizei winWidth = 400, winHeight = 300; // Tamaño inicial de la ventana
                                            // de visualización.

void init (void)
{
    glClearColor (0.0, 0.0, 1.0, 1.0) // Establecer azul como color ventana de
                                         // visualización.

    glMatrixMode (GL_PROJECTION);
    gluOrtho2D (0.0, 200.0, 0.0, 150.0);
}

void displayFcn (void)
{
    glClear (GL_COLOR_BUFFER_BIT); // Borrar ventana de visualización.

    glColor3f (1.0, 0.0, 0.0);    // Establecer rojo como color de punto.
    glPointSize (3.0);           // Definir tamaño de punto 3.0.
}
```

```

void winReshapeFcn (GLint newWidth, GLint newHeight)
{
    /* Reinicializar parámetros de proyección y visor */
    glViewport (0, 0, newWidth, newHeight);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ( );
    gluOrtho2D (0.0, GLdouble (newWidth), 0.0, GLdouble (newHeight));

    /* Reinicializar parámetros de tamaño ventana de visualización. */
    winWidth = newWidth;
    winHeight = newHeight;
}

void plotPoint (GLint x, GLint y)
{
    glBegin (GL_POINTS);
    glVertex2i (x, y);
    glEnd ( );
}

void mousePtPlot (GLint button, GLint action, GLint xMouse, GLint yMouse)
{
    if (button == GLUT_LEFT_BUTTON && action == GLUT_DOWN)
        plotPoint (xMouse, winHeight - yMouse);

    glFlush ( );
}

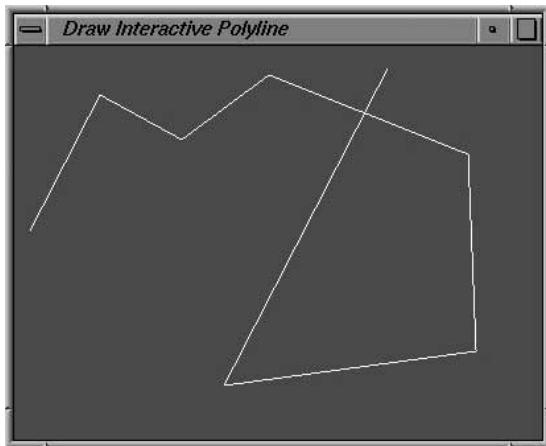
void main (int argc, char** argv)
{
    glutInit (&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    glutInitWindowPosition (100, 100);
    glutInitWindowSize (winWidth, winHeight);
    glutCreateWindow ("Mouse Plot Points");

    init ( );
    glutDisplayFunc (displayFcn);
    glutReshapeFunc (winReshapeFcn);
    glutMouseFunc (mousePtPlot);

    glutMainLoop ( );
}

```

El siguiente ejemplo de programa utiliza la entrada de ratón para seleccionar la posición de un extremo de un segmento de línea recta. En el programa se conectan segmentos de línea seleccionados para ilustrar la construcción interactiva de una polilínea. Inicialmente, deben seleccionarse con el botón izquierdo del ratón dos ubicaciones dentro de la ventana de visualización para generar el primer segmento de línea. Cada una de las posiciones subsiguientes que se seleccione agrega otro segmento a la polilínea. En la Figura 11.13 se proporciona una salida de ejemplo de este programa.



**FIGURA 11.13.** Una salida de ejemplo del procedimiento interactivo polyline de control con el ratón.

```
#include <GL/glut.h>

GLsizei winWidth = 400, winHeight = 300; // Tamaño inicial ventana visualización.
GLint endPtCtr = 0; // Inicializar contador de puntos
                     // extremos de líneas.

class scrPt {
public:
    GLint x, y;
};

void init (void)
{
    glClearColor (0.0, 0.0, 1.0, 1.0) // Establecer azul como color ventana de
                                       // visualización.

    glMatrixMode (GL_PROJECTION);
    gluOrtho2D (0.0, 200.0, 0.0, 150.0);
}

void displayFcn (void)
{
    glClear (GL_COLOR_BUFFER_BIT);
}

void winReshapeFcn (GLint newWidth, GLint newHeight)
{
    /* Reinicializar parámetros de proyección y visor */
    glViewport (0, 0, newWidth, newHeight);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ( );
    gluOrtho2D (0.0, GLdouble (newWidth), 0.0, GLdouble (newHeight));
    /* Reinicializar parámetros de tamaño de la ventana de visualización. */
    winWidth = newWidth;
    winHeight = newHeight;
}
```

```

void drawLineSegment (scrPt endPt1, scrPt endPt2)
{
    glBegin (GL_LINES);
        glVertex2i (endPt1.x, endPt1.y);
        glVertex2i (endPt2.x, endPt2.y);
    glEnd ( );
}

void polyline (GLint button, GLint action, GLint xMouse, GLint yMouse)
{
    static scrPt endPt1, endPt2;

    if (ptCtr == 0) {
        if (button == GLUT_LEFT_BUTTON && action == GLUT_DOWN) {
            endPt1.x = xMouse;
            endPt1.y = winHeight - yMouse;
            ptCtr = 1;
        }
        else
            if (button == GLUT_RIGHT_BUTTON) // Salir del programa.
                exit (0);
    }
    else
        if (button == GLUT_LEFT_BUTTON && action == GLUT_DOWN) {
            endPt2.x = xMouse;
            endPt2.y = winHeight - yMouse;
            drawLineSegment (endPt1, endPt2);

            endPt1 = endPt2;
        }
        else
            if (button == GLUT_RIGHT_BUTTON) // Salir del programa.
                exit (0);

        glFlush ( );
    }
}

void main (int argc, char** argv)
{
    glutInit (&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    glutInitWindowPosition (100, 100);
    glutInitWindowSize (winWidth, winHeight);
    glutCreateWindow ("Draw Interactive Polyline");

    init ( );
    glutDisplayFunc (displayFcn);
    glutReshapeFunc (winReshapeFcn);
    glutMouseFunc (polyline);

    glutMainLoop ( );
}

```

Otra rutina de ratón de GLUT que podemos utilizar es:

```
glutMotionFunc (fcnDoSomething);
```

Esta rutina invoca `fcnDoSomething` cuando se mueve el ratón dentro de la ventana de visualización con uno o más botones activados. La función que se invoca en este caso tiene dos argumentos:

```
void fcnDoSomething (GLint xMouse, GLint yMouse)
```

donde (`xMouse`, `yMouse`) es la ubicación del ratón en la ventana de visualización, relativa a la esquina superior izquierda, cuando se mueve el ratón con un botón pulsado.

De forma similar, podemos realizar alguna acción cuando movamos el ratón dentro de la ventana de visualización sin presionar un botón:

```
glutPassiveMotionFunc (fcnDoSomethingElse);
```

De nuevo, la ubicación del ratón se devuelve a `fcnDoSomethingElse` como la posición de coordenadas (`xMouse`, `yMouse`), relativa a la esquina superior izquierda de la ventana de visualización.

## Funciones de teclado GLUT

Con la entrada de teclado, utilizamos la siguiente función para especificar el procedimiento que haya que invocar cuando se pulse una tecla:

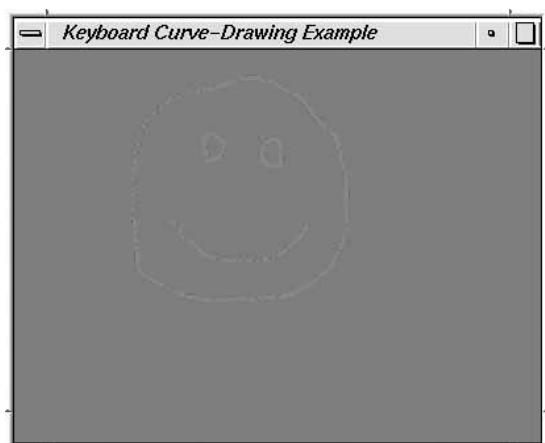
```
glutKeyboardFunc (keyFcn);
```

El procedimiento especificado tiene tres argumentos:

```
void keyFcn (GLubyte key, GLint xMouse, GLint yMouse)
```

Al parámetro `key` se le asigna un valor de tipo carácter o el correspondiente código ASCII. La ubicación del ratón dentro de la ventana de visualización se devuelve como la posición (`xMouse`, `yMouse`), relativa a la esquina superior izquierda de la ventana de visualización. Cuando se pulsa una tecla designada, podemos utilizar la ubicación del ratón para iniciar alguna acción, independientemente de si hay o no pulsado algún botón del ratón.

En el siguiente código, presentamos un procedimiento simple de dibujo de curvas utilizando la entrada de teclado. Se genera una curva a mano alzada moviendo el ratón dentro de la ventana de visualización mientras se mantiene apretada la tecla «`c`». Esto hace que se muestre una secuencia de puntos rojos en cada una de las posiciones registradas del ratón. Moviendo lentamente el ratón, podemos obtener una línea curva continua. Los botones del ratón no tienen ningún efecto en este ejemplo. En la Figura 11.14 se proporciona una salida de ejemplo del programa.



**FIGURA 11.14.** Una salida de ejemplo que muestra un dibujo a mano alzada generado mediante el procedimiento `curveDrawing`.

```

#include <GL/glut.h>

GLsizei winWidth = 400, winHeight = 300; // Tamaño inicial ventana visualización.

void init (void)
{
    glClearColor (0.0, 0.0, 1.0, 1.0); // Establecer azul como color ventana de
visualización.

    glMatrixMode (GL_PROJECTION);
    gluOrtho2D (0.0, 200.0, 0.0, 150.0);
}

void displayFcn (void)
{
    glClear (GL_COLOR_BUFFER_BIT); // Borrar ventana de visualización.

    glColor3f (1.0, 0.0, 0.0); // Establecer rojo como color de punto.
    glPointSize (3.0); // Especificar tamaño de punto 3.0.
}

void winReshapeFcn (GLint newWidth, GLint newHeight)
{
    /* Reinicializar parámetros de proyección y visor */
    glViewport (0, 0, newWidth, newHeight);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    gluOrtho2D (0.0, GLdouble (newWidth), 0.0, GLdouble (newHeight));

    /* Reinicializar parámetros de tamaño de la ventana de visualización. */
    winWidth = newWidth;
    winHeight = newHeight;
}

void plotPoint (GLint x, GLint y)
{
    glBegin (GL_POINTS);
    glVertex2i (x, y);
    glEnd ();
}

/* Mover el cursor mientras se pulsa la tecla c activa el dibujo de curvas
a mano alzada. */
void curveDrawing (GLubyte curvePlotKey, GLint xMouse, GLint yMouse)
{
    GLint x = xMouse;
    GLint y = winHeight - yMouse;
    switch (curvePlotKey)
    {
        case 'c':
            plotPoint (x, y);
    }
}

```

```
        break;
    default:
        break;
    }
    glFlush ( );
}

void main (int argc, char** argv)
{
    glutInit (&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    glutInitWindowPosition (100, 100);
    glutInitWindowSize (winWidth, winHeight);
    glutCreateWindow ("Keyboard Curve-Drawing Example");

    init ( );
    glutDisplayFunc (displayFcn);
    glutReshapeFunc (winReshapeFcn);
    glutKeyboardFunc (curveDrawing);

    glutMainLoop ( );
}
```

Para las teclas de función, teclas de flecha y otras teclas de propósito especial, podemos utilizar el comando:

```
glutSpecialFunc (specialKeyFcn);
```

El procedimiento especificado tiene los mismos tres argumentos:

```
void specialKeyFcn (GLint specialKey, GLint xMouse,  
                     GLint yMouse)
```

pero ahora al parámetro `specialKey` se le asigna una constante simbólica GLUT de valor entero. Para seleccionar una tecla de función, utilizamos una de las constantes `GLUT_KEY_F1` a `GLUT_KEY_F12`. Para las teclas de cursor, usamos constantes del estilo de `GLUT_KEY_UP` y `GLUT_KEY_RIGHT`. Otras teclas pueden designarse mediante `GLUT_KEY_PAGE_DOWN`, `GLUT_KEY_HOME` y otras constantes similares para las teclas de RePág, Fin e Insert. Las teclas de retroceso, Supr y Esc pueden designarse mediante la rutina `glutKeyboardFunc` utilizando sus códigos ASCII, que son 8, 127 y 27, respectivamente.

El siguiente fragmento de código es un programa interactivo que ilustra el uso del ratón, el teclado y las teclas de función. La entrada de ratón se utiliza para seleccionar una ubicación para la esquina inferior izquierda de un cuadrado rojo. La entrada de teclado se emplea para cambiar la escala del cuadrado y podemos obtener un nuevo cuadrado con cada clic del botón izquierdo del ratón.

```
void init (void)
{
    glClearColor (0.0, 0.0, 1.0, 1.0) // Establecer azul como color ventana de
visualización.

    glMatrixMode (GL_PROJECTION);
    gluOrtho2D (0.0, 200.0, 0.0, 150.0);
}

void displayFcn (void)
{
    glClear (GL_COLOR_BUFFER_BIT); // Borrar ventana de visualización.

    glColor3f (1.0, 0.0, 0.0); // Establecer rojo como color de relleno.
}

void winReshapeFcn (GLint newWidth, GLint newHeight)
{
    /* Reinicializar parámetros de proyección y visor */
    glViewport (0, 0, newWidth, newHeight);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ( );
    gluOrtho2D (0.0, GLdouble (newWidth), 0.0, GLdouble (newHeight));

    /* Reinicializar parámetros de tamaño de la ventana de visualización. */
    winWidth = newWidth;
    winHeight = newHeight;
}

/* Mostrar un cuadrado rojo con un tamaño seleccionado de arista. */
void fillSquare (GLint button, GLint action, GLint xMouse, GLint yMouse)
{
    GLint x1, y1, x2, y2;

    /* Usar botón izquierdo del ratón para seleccionar una posición para
     * la esquina inferior izquierda del cuadrado.
     */
    if (button == GLUT_LEFT_BUTTON && action == GLUT_DOWN)
    {
        x1 = xMouse;
        y1 = winHeight - yMouse;
        x2 = x1 + edgeLength;
        y2 = y1 + edgeLength;
        glRecti (x1, y1, x2, y2);
    }
    else
        if (button == GLUT_RIGHT_BUTTON) // Utilizar botón derecho del ratón para
salir.
            exit (0);

        glFlush ( );
}
```

```
/* Utilizar teclas 2, 3 y 4 para agrandar el cuadrado. */
void enlargeSquare (GLubyte sizeFactor, GLint xMouse, GLint yMouse)
{
    switch (sizeFactor)
    {
        case '2':
            edgeLength *= 2;
            break;
        case '3':
            edgeLength *= 3;
            break;
        case '4':
            edgeLength *= 4;
            break;
        default:
            break;
    }
}
/* Utilizar teclas de función F2 y F4 para los factores de reducción 1/2 y 1/4. /
void reduceSquare (GLint reductionKey, GLint xMouse, GLint yMouse)
{
    switch (reductionKey)
    {
        case GLUT_KEY_F2:
            edgeLength /= 2;
            break;
        case GLUT_KEY_F3:
            edgeLength /= 4;
            break;
        default:
            break;
    }
}

void main (int argc, char** argv)
{
    glutInit (&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    glutInitWindowPosition (100, 100);
    glutInitWindowSize (winWidth, winHeight);
    glutCreateWindow ("Display Squares of Various Sizes");

    init ( );
    glutDisplayFunc (displayFcn);
    glutReshapeFunc (winReshapeFcn);
    glutMouseFunc (fillSquare);
    glutKeyboardFunc (enlargeSquare);
    glutSpecialFunc (reduceSquare);

    glutMainLoop ( );
}
```

## Funciones GLUT para tabletas gráficas

Usualmente, la tableta se activa solamente cuando el cursor del ratón se encuentra dentro de la ventana de visualización. Entonces, puede registrarse un suceso de botón para la entrada de la tableta gráfica mediante:

```
glutTabletButtonFunc (tabletFcn);
```

y los argumentos para la función invocada son similares a los del caso del ratón:

```
void tabletFcn (GLint tabletButton, GLint action,
                GLint xTablet, GLint yTablet)
```

Podemos designar un botón de la tableta mediante un identificador entero tal como 1, 2, 3, etcétera, y la acción correspondiente al botón se especifica, de nuevo, mediante GLUT\_UP o GLUT\_DOWN. Los valores devueltos xTablet e yTablet son las coordenadas de la tableta. Podemos determinar el número de botones de la tableta disponibles mediante el comando:

```
glutDeviceGet (GLUT_NUM_TABLET_BUTTONS);
```

El movimiento de un cursor o un lápiz en la tableta se procesa mediante la siguiente función:

```
glutTabletMotionFunc (tabletMotionFcn);
```

donde la función invocada tiene la forma:

```
void tabletMotionFcn (GLint xTablet, GLint yTablet)
```

Los valores devueltos xTablet e yTablet proporcionan las coordenadas sobre la superficie de la tableta.

## Funciones GLUT para una *spaceball*

Podemos utilizar la siguiente función para especificar una operación que debe llevarse a cabo cuando se active un botón de la *spaceball* para una ventana de visualización seleccionada:

```
glutSpaceballButtonFunc (spaceballFcn);
```

La función de retrollamada tiene dos parámetros:

```
void spaceballFcn (GLint spaceballButton, GLint action)
```

Los botones de la *spaceball* se identifican mediante los mismos valores enteros que para una tableta, y al parámetro action se le asigna el valor GLUT\_UP o GLUT\_DOWN. Podemos determinar el número de botones de la *spaceball* disponibles mediante una llamada a glutDeviceGet utilizando el argumento GLUT\_NUM\_SPACEBALL\_BUTTONS.

El movimiento de traslación de una *spaceball* cuando el ratón se encuentra en la ventana de visualización se registra mediante la llamada de función:

```
glutSpaceballMotionFunc (spaceballTranlFcn);
```

Las distancias de traslación tridimensional se pasan a la función invocada, como por ejemplo:

```
void spaceballTranslFcn (GLint tx, GLint ty, GLint tz)
```

Estas distancias de traslación están normalizadas en el rango comprendido entre -1000 y 1000. De forma similar, una rotación de la *spaceball* se registra mediante:

```
glutSpaceballRotateFunc (spaceballRotFcn);
```

Los ángulos de rotación tridimensionales estarán entonces disponibles también para la función de retrollamada:

```
void spaceballRotFcn (GLint thetaX, GLint thetaY, GLint thetaZ)
```

## Funciones GLUT para cajas de botones

La entrada procedente de una caja de botones se obtiene mediante la siguiente instrucción:

```
glutButtonBoxFunc (buttonBoxFcn);
```

La activación de un botón puede pasarse a la función invocada mediante los correspondientes parámetros:

```
void buttonBoxFcn (GLint button, GLint action);
```

Los botones se identifican mediante valores enteros y la acción realizada por el botón se especifica como: GLUT\_UP o GLUT\_DOWN.

## Funciones GLUT para diales

La rotación de un dial puede registrarse mediante la siguiente rutina:

```
glutDialsFunc (dialsFcn);
```

En este caso, utilizamos la función de retrollamada para identificar el dial y obtener el ángulo de rotación:

```
void dialsFcn (GLint dial, GLint degreeValue);
```

Los diales se designan mediante valores enteros y la rotación del dial se devuelve como un valor entero que especifica los grados.

## Operaciones de selección en OpenGL

En un programa OpenGL, podemos seleccionar objetos interactivamente apuntando a las posiciones de pantalla. Sin embargo, estas operaciones de selección en OpenGL no son particularmente sencillas.

Básicamente, realizamos la selección utilizando una ventana de selección especificada, con el fin de generar un nuevo volumen de visualización. Asignamos identificadores enteros a los objetos de una escena y los identificadores de aquellos objetos que intercepten con el nuevo volumen de visualización se almacenarán en una matriz que actúa como búfer de selección. Así, para utilizar las funciones de selección OpenGL, necesitamos implementar los siguientes procedimientos dentro del programa:

- Crear y visualizar una escena.
- Seleccionar una posición en pantalla y, dentro de la función de retrollamada del botón, llevar a cabo los siguientes pasos:
  - Configurar un búfer de selección.
  - Activar las operaciones de selección (modo de selección).
  - Inicializar una pila de nombres de identificación para los identificadores de los objetos.
  - Guardar la matriz actual de las transformaciones de visualización y geométrica.
  - Especificar una ventana de selección para la entrada del ratón.
  - Asignar identificadores a los objetos y volver a procesar la escena utilizando el nuevo volumen de visualización (la información de selección se almacenará entonces en el búfer de selección).
  - Restaurar la matriz original de transformación de visualización y geométrica.
  - Determinar el número de objetos que se han seleccionado y volver al modo de representación normal.
  - Procesar la información de selección.

También podemos utilizar una modificación de estos procedimientos para seleccionar objetos sin necesidad de utilizar el ratón para la entrada interactiva. Esto se lleva a cabo especificando los vértices del nuevo volumen de visualización, en lugar de designar una ventana de selección.

Podemos definir una matriz que actúe como búfer de selección mediante el comando:

```
glSelectBuffer (pickBufferSize, pickBuffer);
```

El parámetro `pickBuffer` designa una matriz de enteros con `pickBufferSize` elementos. La función `glSelectBuffer` debe invocarse antes de activar las operaciones OpenGL de selección (modo de selección). Para cada objeto seleccionado mediante una única entrada de selección se registrará la correspondiente información dentro del búfer de selección, en forma de valores enteros. En el búfer de selección pueden almacenarse varios registros de información, dependiendo del tamaño y de la ubicación de la ventana de selección. Cada registro del búfer de selección contiene la información siguiente:

- (1) La posición del objeto en la pila, que es el número de identificadores que hay en la pila de nombres, incluyendo la propia posición del objeto seleccionado.
- (2) La profundidad mínima del objeto seleccionado.
- (3) La profundidad máxima del objeto seleccionado.
- (4) La lista de identificadores contenida en la pila de nombres, desde el primer identificador (el de la parte inferior) hasta el identificador correspondiente al objeto seleccionado.

Los valores enteros de profundidad almacenados en el búfer de selección son los valores originales, comprendidos en el rango de 0 a 1.0, multiplicados por  $2^{32} - 1$ .

Las operaciones de selección OpenGL se activan mediante:

```
glRenderMode (GL_SELECT);
```

Esto hace que el sistema se configure en modo de selección, lo que significa que se procese la escena a través de la pipeline de visualización, pero el resultado no se almacena en el búfer de imagen. En lugar de ello, se almacena en el búfer de selección un registro de información para cada objeto que hubiera sido visualizado en el modo normal de representación. Además, este comando devuelve el número de objetos seleccionados, que será igual al número de registros de información contenidos en el búfer de selección. Para volver al modo de representación normal (el predeterminado) invocamos la rutina `glRenderMode` utilizando el argumento `GL_RENDER`. Una tercera opción es el argumento `GL_FEEDBACK`, que almacena las coordenadas de los objetos y otras informaciones en un búfer de realimentación sin visualizar los objetos. Este modo de realimentación se utiliza para obtener información acerca de los tipos de las primitivas, los atributos y otros parámetros asociados con los objetos de una escena.

Utilizamos la siguiente instrucción para activar la pila de nombres basados en identificadores enteros para las operaciones de selección:

```
glInitNames ( );
```

La pila de identificadores está inicialmente vacía y sólo puede utilizarse en el modo de selección. Para almacenar un valor entero sin signo en la pila, podemos invocar la siguiente función:

```
glPushName (ID);
```

Esto coloca el valor del parámetro `ID` en la parte superior de la pila y empuja el nombre previamente colocado en la primera posición hasta la siguiente posición de la pila. También podemos simplemente sustituir la posición superior de la fila utilizando:

```
glLoadName (ID);
```

pero no podemos emplear este comando para almacenar un valor en una pila vacía. Para eliminar el elemento superior de la pila de identificadores, se utiliza el comando:

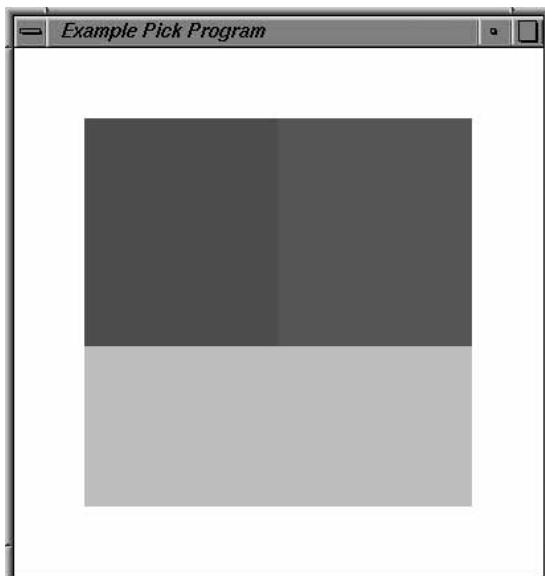
```
glPopName ( );
```

Para definir una ventana de selección dentro de un visor especificado se utiliza la siguiente función GLU:

```
gluPickMatrix (xPick, yPick, widthPick, heightPick, vpArray);
```

Los parámetros `xPick` e `yPick` proporcionan la ubicación en coordenadas de pantalla de doble precisión correspondiente al centro de la ventana de selección, siendo esas coordenadas relativas a la esquina inferior izquierda del visor. Cuando se introducen estas coordenadas mediante la entrada de ratón, las coordenadas del ratón se especifican de forma relativa a la esquina superior izquierda, por lo que será necesario invertir el valor `yMouse` de entrada. Los valores de doble precisión para la anchura y la altura de la ventana de selección se especifican mediante los parámetros `widthPick` y `heightPick`. El parámetro `vpArray` designa una matriz de enteros que contiene las coordenadas y el tamaño del visor actual. Podemos obtener los parámetros del visor utilizando la función `glGetIntegerv` (Sección 6.4). Esta ventana de selección se utiliza entonces como ventana de recorte para construir un nuevo volumen de visualización para las transformaciones de visualización. La información relativa a los objetos que intersecten este nuevo volumen de visualización se almacenará en el búfer de selección.

Vamos a ilustrar las operaciones de selección OpenGL en el siguiente programa, que visualiza los tres rectángulos de color que se muestran en la Figura 11.15. Para este ejemplo de selección, utilizamos una ventana de selección 5 por 5, proporcionando el centro de la ventana de selección mediante la entrada de ratón. Por tanto, necesitaremos invertir el valor `yMouse` de entrada utilizando la altura del visor, que es el cuarto elemento de la matriz `vpArray`. Asignamos al rectángulo rojo el valor ID = 30, al rectángulo azul el identificador ID = 10 y al rectángulo verde el identificador ID = 20. Dependiendo de la posición de entrada del ratón, podemos no seleccionar ningún rectángulo, seleccionar uno, seleccionar dos o seleccionar los tres a la vez. Los identificadores de los rectángulos se introducen en la pila de identificadores según su orden de color: rojo, azul, verde. Por tanto, cuando procesemos un rectángulo seleccionado, podemos utilizar su identificador o su número de posición dentro de la pila. Por ejemplo, si el número de posición en la pila, que es el primer elemento en el registro de selección, es 2, entonces habremos seleccionado el rectángulo azul y habrá dos identificadores de rectángulo especificados al final del registro. Alternativamente, podríamos utilizar la última entrada del registro, que será el identificador del objeto seleccionado. En este programa de ejemplo, simplemente enumeraremos el contenido del búfer de selección. Los rectángulos están definidos en el plano *xy*, de modo que todos los valores de profundidad son 0. En el Ejemplo 11.1 se proporciona una muestra de la salida para una posición de entrada del ratón que se encuentre cerca de la frontera entre los rectángulos rojo y verde. No proporcionamos ningún mecanismo para terminar el programa, por lo que puede procesarse cualquier número de entradas de ratón.



**FIGURA 11.15.** Los tres rectángulos de color mostrados por el programa de selección de ejemplo.

```
#include <GL/glut.h>
#include <stdio.h>

const GLint pickBuffSize = 32;

/* Establecer tamaño inicial ventana visualización. */
GLsizei winWidth = 400, winHeight = 400;

void init (void)
{
    /* Especificar blanco como color de la ventana de visualización. */
    glClearColor (1.0, 1.0, 1.0, 1.0);
}

/* Definir 3 rectángulos y sus identificadores asociados. */
void rects (GLenum mode)
{
    if (mode == GL_SELECT)
        glPushName (30); // Rectángulo rojo.
    glColor3f (1.0, 0.0, 0.0);
    glRecti (40, 130, 150, 260);
    if (mode == GL_SELECT)
        glPushName (10); // Rectángulo azul.
    glColor3f (0.0, 0.0, 1.0);
    glRecti (150, 130, 260, 260);

    if (mode == GL_SELECT)
        glPushName (20); // Rectángulo verde.
    glColor3f (0.0, 1.0, 0.0);
    glRecti (40, 40, 260, 130);
}

/* Imprimir el contenido del búfer de selección para cada selección del ratón. */
void processPicks (GLint nPicks, GLuint pickBuffer [ ])
{
    GLint j, k;
    GLuint objID, *ptr;

    printf (" Number of objects picked = %d\n", nPicks);
    printf ("\n");
    ptr = pickBuffer;

    /* Proporcionar como salida todos los elementos en cada registro de selección.
    */
    for (j = 0; j < nPicks; j++) {
        objID = *ptr;

        printf (" Stack position = %d\n", objID);
    }
}
```

```

ptr++;

printf (" Min depth = %g,", float (*ptr/0x7fffffff));
ptr++;

printf (" Max depth = %g\n", float (*ptr/0x7fffffff));
ptr++;

printf (" Stack IDs are: \n");
for (k = 0; k < objID; k++) {
    printf (" %d ",*ptr);
    ptr++;
}
printf ("\n\n");
}

void pickRects (GLint button, GLint action, GLint xMouse, GLint yMouse)
{
    GLuint pickBuffer [pickBuffSize];
    GLint nPicks, vpArray [4];

    if (button != GLUT_LEFT_BUTTON || action != GLUT_DOWN)
        return;

    glSelectBuffer (pickBuffSize, pickBuffer); // Designar el búfer de selección.
    glRenderMode (GL_SELECT);                // Activar las operaciones de selección.
    glInitNames ();                         // Inicializar la pila de identificadores
de objetos.

/* Guardar matriz de visualización actual. */
    glMatrixMode (GL_PROJECTION);
    glPushMatrix ( );
    glLoadIdentity ( );

    /* Obtener los parámetros para el visor actual. Definir
     * una ventana de selección 5 por 5 e invertir el valor yMouse de entrada
     * utilizando la altura del visor, que es el cuarto
     * elemento de vpArray.
     */
    glGetIntegerv (GL_VIEWPORT, vpArray);
    gluPickMatrix (GLdouble (xMouse), GLdouble (vpArray [3] - yMouse),
                  5.0, 5.0, vpArray);

    gluOrtho2D (0.0, 300.0, 0.0, 300.0);
    rects (GL_SELECT); // Procesar los rectángulos en el modo de selección.

    /* Restaurar la matriz de visualización original. */
    glMatrixMode (GL_PROJECTION);
    glPopMatrix ( );
}

```

```
glFlush ( );

/* Determinar el número de objetos seleccionados y volver
 * al modo de representación normal.
 */
nPicks = glRenderMode (GL_RENDER);

processPicks (nPicks, pickBuffer); // Procesar los objetos seleccionados.

glutPostRedisplay ( );
}

void displayFcn (void)
{
    glClear (GL_COLOR_BUFFER_BIT);
    rects (GL_RENDER); // Visualizar los rectángulos.
    glFlush ( );
}

void winReshapeFcn (GLint newWidth, GLint newHeight)
{
    /* Reinicializar parámetros de proyección y visor. */
    glViewport (0, 0, newWidth, newHeight);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ( );

    gluOrtho2D (0.0, 300.0, 0.0, 300.0);
    glMatrixMode (GL_MODELVIEW);

    /* Reinicializar parámetros de tamaño de la ventana de visualización. */
    winWidth = newWidth;
    winHeight = newHeight;
}

void main (int argc, char** argv)
{
    glutInit (&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    glutInitWindowPosition (100, 100);
    glutInitWindowSize (winWidth, winHeight);
    glutCreateWindow ("Example Pick Program");

    init ( );
    glutDisplayFunc (displayFcn);
    glutReshapeFunc (winReshapeFcn);
    glutMouseFunc (pickRects);

    glutMainLoop ( );
}
```

**Ejemplo 11.1 Salida de ejemplo del procedimiento pickrects**

```

Number of objects picked 5 2

Stack position 5 1

Min depth 5 0, Max depth 5 0
Stack IDs are:
30

Stack position 5 3
Min depth 5 0, Max depth 5 0
Stack IDs are:
30 10 20

```

## 11.7 FUNCIONES DE MENÚ OpenGL

Además de las rutinas correspondientes a los dispositivos de entrada, GLUT contiene diversas funciones para añadir menús emergentes simples a los programas. Con estas funciones, podemos especificar y acceder a diversos menús y a sus submenús asociados. Los comandos GLUT de menú se incluyen en el procedimiento `main` junto con las demás funciones GLUT.

### Creación de un menú GLUT

Podemos crear un menú emergente mediante la instrucción:

```
glutCreateMenu (menuFcn);
```

donde el parámetro `menuFcn` es el nombre de un procedimiento que haya que invocar cuando se seleccione una entrada del menú. Este procedimiento tiene un argumento, que será el valor entero correspondiente a la posición de una opción seleccionada:

```
void menuFcn (GLint menuItemNumber)
```

El valor entero pasado mediante el parámetro `menuItemNumber` se utiliza entonces en la función `menuFcn` para realizar algún tipo de operación. Cuando se crea un menú, se le asocia con la ventana de visualización actual.

Una vez designada la función de menú que haya que invocar cuando se seleccione un elemento del menú, deberemos especificar las opciones que haya que enumerar dentro del menú. Podemos hacer esto con una serie de instrucciones que indiquen el nombre y la posición de cada opción. Estas instrucciones tienen la forma general:

```
glutAddMenuEntry (charString, menuItemNumber);
```

El parámetro `charString` especifica el texto que hay que visualizar en el menú, mientras que el parámetro `menuItemNumber` indica la ubicación de dicha entrada dentro del menú. Por ejemplo, las siguientes instrucciones crean un menú con dos opciones:

```
glutCreateMenu (menuFcn);
```

```
glutAddMenuEntry ("First Menu Item", 1);
glutAddMenuEntry ("Second Menu Item", 2);
```

A continuación, debemos especificar el botón del ratón que hay que utilizar para seleccionar una opción del menú. Esto se realiza mediante:

```
glutAttachMenu (button);
```

donde al parámetro `button` se le asigna una de las tres constantes simbólicas GLUT que hacen referencia a los botones izquierdo, central y derecho del ratón.

Para ilustrar la creación y utilización de un menú GLUT, el siguiente programa proporciona dos opciones para visualizar el relleno interior de un triángulo. Inicialmente, el triángulo está definido con dos vértices blancos, un vértice rojo y un color de relleno determinado por la interpolación de los colores de los vértices. Utilizamos la función `glShadeModel` (Secciones 4.14 y 10.20) para seleccionar un relleno del polígono que puede ser un color homogéneo o una interpolación (representación de Gouraud) de los colores de los vértices. En este programa creamos un menú que nos permite elegir entre las dos opciones utilizando el botón derecho del ratón, cuando el cursor del ratón se encuentre dentro de la ventana de visualización. Este menú emergente se muestra con la esquina superior izquierda situada en la posición del cursor del ratón, como se ilustra en la Figura 11.16. Las opciones de menú se resaltarán cuando situemos el cursor del ratón sobre ellas. La opción resaltada puede entonces seleccionarse liberando el botón derecho. Si se selecciona la opción «*Solid-Color Fill*», el triángulo se llenará con el color especificado para el último vértice (que es el rojo). Al final del procedimiento de visualización del menú, `fillOption`, incluimos un comando `glutPostRedisplay` (Sección 6.4) para indicar que es necesario redibujar el triángulo cuando se visualice el menú.

---

```
#include <GL/glut.h>

GLsizei winWidth = 400, winHeight = 400; // Tamaño inicial ventana visualización.

GLfloat red = 1.0, green = 1.0, blue = 1.0; // Color inicial del triángulo:
blanco.

GLenum fillMode = GL_SMOOTH; // Relleno inicial del polígono: interpolación de
colores.

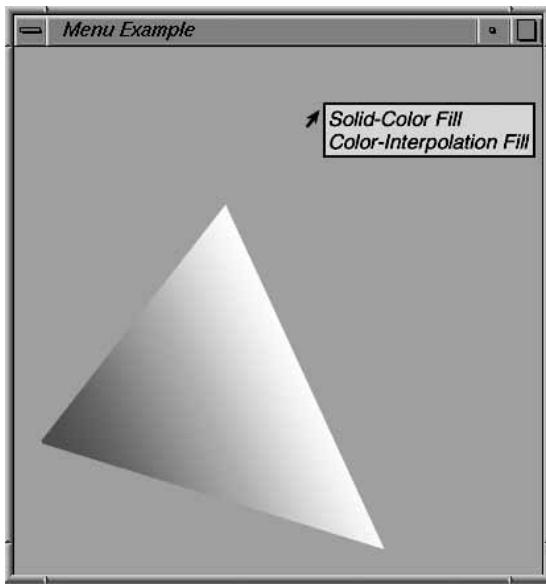
void init (void)
{
    glClearColor (0.6, 0.6, 0.6, 1.0); // Establecer gris como color ventana de
visualización.

    glMatrixMode (GL_PROJECTION);
    gluOrtho2D (0.0, 300.0, 0.0, 300.0);
}

void fillOption (GLint selectedOption)
{
    switch (selectedOption) {
        case 1: fillMode = GL_FLAT; break; // Representación plana de superficies.
        case 2: fillMode = GL_SMOOTH; break; // Representación de Gouraud.
    }
    glutPostRedisplay ( );
}

void displayTriangle (void)
```

```
{  
    glClear (GL_COLOR_BUFFER_BIT);  
  
    glShadeModel (fillMode);          // Establecer método de relleno para triángulo.  
    glColor3f (red, green, blue); // Establecer color para los primeros dos vértices.  
  
    glBegin (GL_TRIANGLES);  
        glVertex2i (280, 20);  
        glVertex2i (160, 280);  
        glColor3f (red, 0.0, 0.0); // Asignar rojo como color del último vértice.  
        glVertex2i (20, 100);  
    glEnd ( );  
  
    glFlush ( );  
}  
  
void reshapeFcn (GLint newWidth, GLint newHeight)  
{  
    glViewport (0, 0, newWidth, newHeight);  
  
    glMatrixMode (GL_PROJECTION);  
    glLoadIdentity ( );  
    gluOrtho2D (0.0, GLfloat (newWidth), 0.0, GLfloat (newHeight));  
    displayTriangle ( );  
    glFlush ( );  
}  
  
void main (int argc, char **argv)  
{  
    glutInit (&argc, argv);  
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);  
    glutInitWindowPosition (200, 200);  
    glutInitWindowSize (winWidth, winHeight);  
    glutCreateWindow ("Menu Example");  
  
    init ( );  
    glutDisplayFunc (displayTriangle);  
  
    glutCreateMenu (fillOption); // Crear menú emergente.  
    glutAddMenuEntry ("Solid-Color Fill", 1);  
    glutAddMenuEntry ("Color-Interpolation Fill", 2);  
  
    /* Seleccionar opción de menú utilizando el botón derecho del ratón. */  
    glutAttachMenu (GLUT_RIGHT_BUTTON);  
  
    glutReshapeFunc (reshapeFcn);  
  
    glutMainLoop ( );  
}
```



**FIGURA 11.16.** Menú OpenGL emergente mostrado por el programa de ejemplo de gestión de menús.

## Creación y gestión de múltiples menús GLUT

Cuando se crea un menú, se lo asocia con la ventana de visualización actual (Sección 6.4). Podemos crear múltiples menús para una misma ventana de visualización y también crear diferentes menús para las diferentes ventanas. A medida que se crea cada menú, se le asigna un identificador entero, comenzando con el valor 1 para el primer valor creado. El identificador entero correspondiente a un menú es devuelto por la rutina `glutCreateMenu`, y podemos registrar dicho valor mediante una instrucción como la siguiente:

```
menuID = glutCreateMenu (menuFcn);
```

El menú recién creado se convertirá en el **menú actual** para la ventana de visualización actual. Para activar un menú en la ventana de visualización actual, se utiliza la instrucción:

```
glutSetMenu (menuID);
```

Este menú se convertirá entonces en el menú actual, que emergerá dentro de la ventana de visualización cuando se pulse el botón del ratón que se haya asociado a dicho menú.

Podemos eliminar un menú mediante el comando:

```
glutDestroyMenu (menuID);
```

Si el menú especificado es el menú actual de una ventana de visualización, entonces dicha ventana no tendrá ningún menú asignado como menú actual, aun cuando existan otros menús.

Para obtener el identificador del menú actual de la ventana de visualización actual se utiliza la siguiente función:

```
currentMenuID = glutGetMenu ( );
```

Se devolverá un valor 0 si no hay ningún menú para esta ventana de visualización o si el menú actual anterior ha sido eliminado mediante la función `glutDestroyMenu`.

## Creación de submenús GLUT

Podemos asociar un submenú con un menú creando primero un submenú mediante `glutCreateMenu`, junto con una lista de subopciones, y luego especificando el submenú como opción adicional dentro del menú prin-

cipal. Podemos añadir el submenú a la lista de opciones de un menú principal (o de otro submenú) utilizando una secuencia de instrucciones tal como:

```

submenuID = glutCreateMenu (submenuFcn);
glutAddMenuEntry ("First Submenu Item", 1);

.

.

.

glutCreateMenu (menuFcn);
glutAddMenuEntry ("First Menu Item", 1);

.

.

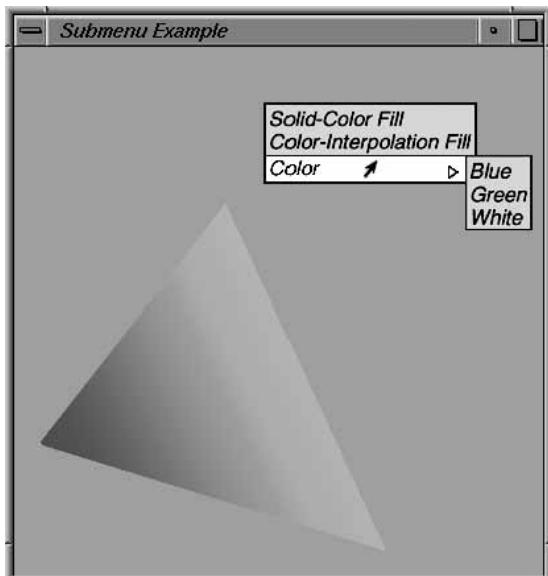
.

glutAddSubMenu ("Submenu Option", submenuID);

```

La función `glutAddSubMenu` puede también utilizarse para añadir el submenú al menú actual.

Vamos a ilustrar la creación de un submenú en el siguiente programa. Este programa, que es una modificación del programa anterior de gestión de menús, muestra un submenú que proporciona tres opciones de color (azul, verde y blanco) para los primeros dos vértices del triángulo. El menú principal se visualizará ahora con tres opciones, y la tercera de ellas incluye un símbolo de flecha para indicar que aparecerá un submenú desplegable cuando se resalte dicha opción, como se muestra en la Figura 11.17. Se incluye una función `glutPostRedisplay` tanto al final de la función correspondiente al menú principal, como de la función correspondiente al submenú.



**FIGURA 11.17.** El menú emergente principal OpenGL y el correspondiente submenú mostrados por el programa de ejemplo de definición de submenús.

---

```

#include <GL/glut.h>

GLsizei winWidth = 400, winHeight = 400; // Tamaño inicial ventana visualización.

GLfloat red = 1.0, green = 1.0, blue = 1.0; // Valores de color iniciales.
GLenum renderingMode = GL_SMOOTH;           // Método inicial de relleno.

void init (void)

```

```

{
    glClearColor (0.6, 0.6, 0.6, 1.0); // Establecer gris como color ventana
                                         // visualización.

    glMatrixMode (GL_PROJECTION);
    gluOrtho2D (0.0, 300.0, 0.0, 300.0);
}

void mainMenu (GLint renderingOption)
{
    switch (renderingOption) {
        case 1: renderingMode = GL_FLAT; break;
        case 2: renderingMode = GL_SMOOTH; break;
    }
    glutPostRedisplay ( );
}

/* Establecer valores de color de acuerdo con la opción del submenú
seleccionada. */
void colorSubMenu (GLint colorOption)
{
    switch (colorOption) {
        case 1:
            red = 0.0; green = 0.0; blue = 1.0;
            break;
        case 2:
            red = 0.0; green = 1.0; blue = 0.0;
            break;
        case 3:
            red = 1.0; green = 1.0; blue = 1.0;
    }
    glutPostRedisplay ( );
}

void displayTriangle (void)
{
    glClear (GL_COLOR_BUFFER_BIT);

    glShadeModel (renderingMode); // Establecer método de relleno para
                                  // el triángulo.
    glColor3f (red, green, blue); // Establecer el color de los primeros
                                  // dos vértices.

    glBegin (GL_TRIANGLES);
    glVertex2i (280, 20);
    glVertex2i (160, 280);
    glColor3f (1.0, 0.0, 0.0); // Seleccionar rojo como color del último vérti-
ce.
    glVertex2i (20, 100);
    glEnd ( );
    glFlush ( );
}

```

```

void reshapeFcn (GLint newWidth, GLint newHeight)
{
    glViewport (0, 0, newWidth, newHeight);

    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ( );
    gluOrtho2D (0.0, GLfloat (newWidth), 0.0, GLfloat (newHeight));

    displayTriangle ( );
    glFlush ( );
}

void main (int argc, char **argv)
{
    GLint subMenu;           // Identificador para el submenú.

    glutInit (&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    glutInitWindowPosition (200, 200);
    glutInitWindowSize (winWidth, winHeight);
    glutCreateWindow ("Submenu Example");

    init ( );
    glutDisplayFunc (displayTriangle);

    subMenu = glutCreateMenu (colorSubMenu);
    glutAddMenuEntry ("Blue", 1);
    glutAddMenuEntry ("Green", 2);
    glutAddMenuEntry ("White", 3);

    glutCreateMenu (mainMenu); // Crear menú principal emergente.
    glutAddMenuEntry ("Solid-Color Fill", 1);
    glutAddMenuEntry ("Color-Interpolation Fill", 2);
    glutAddSubMenu ("Color", subMenu);

    /* Seleccionar opción del menú con el botón derecho del ratón. */
    glutAttachMenu (GLUT_RIGHT_BUTTON);
    glutReshapeFunc (reshapeFcn);

    glutMainLoop ( );
}

```

## Modificación de los menús GLUT

Si queremos cambiar el botón del ratón utilizado para seleccionar una opción del menú, primero tenemos que cancelar la asociación actual de botón y luego asociar el botón nuevo. Podemos cancelar una asociación de botón para el menú actual mediante:

```
glutDetachMenu (mouseButton);
```

Al parámetro `mouseButton` se le asigna la constante GLUT que identifique el botón izquierdo, derecho o central que hubiera sido previamente asociado con el menú.

También pueden modificarse las opciones definidas dentro de un menú existente. Por ejemplo, podemos borrar una opción del menú actual mediante la función:

```
glutRemoveMenuItem (itemNumber);
```

donde al parámetro `itemNumber` se le asigne el valor entero de la opción de menú que haya que borrar.

Otras rutinas GLUT nos permiten modificar los nombres o el estado de los elementos de un menú existente. Por ejemplo, podemos utilizar estas rutinas para cambiar el nombre con el que se visualiza una opción de menú, para cambiar el número de elemento asignado a la opción o para transformar una opción en un submenú.

## 11.8 DISEÑO DE UNA INTERFAZ GRÁFICA DE USUARIO

---

Casi todas las aplicaciones software suelen incluir hoy en día una interfaz gráfica, compuesta por ventanas de visualización, iconos, menús y otras características que sirven para ayudar al usuario a aplicar el software a un problema concreto. Se incluyen diálogos interactivos especializados para que las opciones de programación puedan seleccionarse utilizando términos familiares dentro de un determinado campo, como por ejemplo el diseño arquitectónico y de ingeniería, el dibujo, los gráficos empresariales, la geología, la economía, la química o la física. Otras consideraciones que suelen tenerse en cuenta al diseñar una interfaz de usuario son la adaptación a los diversos niveles de experiencia de los usuarios, la coherencia, el tratamiento de errores y la realimentación.

### El diálogo con el usuario

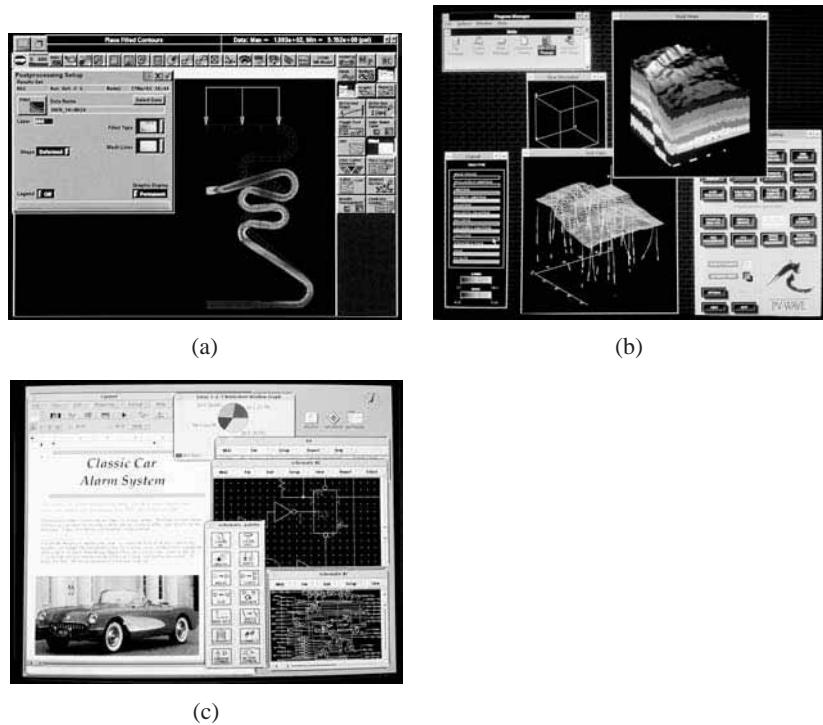
En cualquier aplicación, el *modelo de usuario* sirve como base para el diseño del procedimiento de diálogo en que se basa la aplicación, escribiéndose qué es lo que el sistema debe hacer y qué operaciones hay disponibles. Ese modelo indica el tipo de los objetos que pueden mostrarse y la manera en que se los puede manipular. Por ejemplo, si el sistema va a ser usado como herramienta para el diseño arquitectónico, el modelo describe cómo puede utilizarse el paquete software para construir y visualizar vistas de edificios posicionando las paredes, las puertas, las ventanas y otros componentes. Un paquete de diseño de interiores podría incluir un conjunto de tipos de muebles, junto con las operaciones para colocar y eliminar diferentes objetos en un determinado plano de una vivienda. Y un programa de diseño de circuitos proporciona símbolos eléctricos o lógicos y las operaciones de colocación necesarias para añadir o borrar elementos de un diagrama.

Toda la información del procedimiento de diálogo con el usuario se presenta en el lenguaje de la aplicación. En un paquete de diseño arquitectónico, esto quiere decir que todas las interacciones se describirán exclusivamente en términos arquitectónicos, sin hacer referencia a estructuras de datos concretas, a términos de infografía o a otros conceptos que puedan no resultar familiares para un arquitecto.

### Ventanas e iconos

La Figura 11.18 muestra ejemplos de interfaces gráficas típicas. Se utilizan representaciones visuales tanto para los objetos que hay que manipular en la aplicación como para las acciones que hay que realizar sobre los objetos de la aplicación.

Además de las operaciones estándar de gestión de las ventanas de visualización, como son las de apertura, cierre, posicionamiento y cambio de tamaño, también hacen falta otras operaciones para trabajar con los deslizadores, botones, iconos y menús. Algunos sistemas pueden soportar múltiples gestores de ventanas con el fin de utilizar diferentes estilos de ventana, cada uno con su propio gestor, que podrían estar estructurados de forma especial para una aplicación concreta.



**FIGURA 11.18.** Ejemplos de disposiciones de pantalla que utilizan ventanas de visualización, menús e iconos. (Cortesía de (a) Intergraph Corporation; (b) Visual Numerics, Inc.; (c) Sun Microsystems.)

Los iconos que representan objetos tales como paredes, puertas, ventanas y elementos de circuito se suelen denominar **iconos de aplicación**. Los iconos que representan acciones, como por ejemplo rotar, magnificar, cambiar de escala, recortar o pegar, se denominan **iconos de control** o **iconos de comando**.

### Adaptación a los distintos niveles de experiencia

Usualmente, las interfaces gráficas interactivas proporcionan diversos métodos para la selección de acciones. Por ejemplo, podría especificarse una opción apuntando a un ícono, accediendo a un menú desplegable o emergente o escribiendo un comando en el teclado. Esto permite que el paquete sea utilizado por usuarios con distintos niveles de experiencia.

Para los usuarios menos experimentados, una interfaz con unas pocas operaciones fácilmente comprensibles y con mensajes informativos detallados resulta más efectiva que otra con un conjunto de operaciones grande y exhaustivo. Un conjunto simplificado de menús y de opciones es fácil de aprender y de recordar y el usuario puede concentrarse en la aplicación en lugar de hacerlo en los detalles de la interfaz. Las operaciones simples de tipo apuntar y hacer clic son a menudo las más sencillas para los usuarios poco experimentados de un paquete de aplicación. Por tanto, las interfaces suelen proporcionar algún tipo de mecanismo para enmascarar la complejidad del paquete software, de modo que los principiantes puedan utilizar el sistema sin verse abrumados por un exceso de detalles.

Los usuarios experimentados, por el contrario, lo que normalmente desean es velocidad. Esto significa menos mensajes indicativos y más entrada desde el teclado o mediante múltiples clics de los botones del ratón. Las acciones se seleccionan mediante teclas de función o mediante la combinación simultánea de varias teclas normales, ya que los usuarios experimentados son capaces de recordar estos atajos utilizados para acceder a las acciones más comunes.

De forma similar, las facilidades de ayuda pueden diseñarse con distintos niveles de modo que los principiantes puedan embarcarse en un diálogo detallado, mientras que los usuarios más experimentados pueden reducir o eliminar los mensajes indicativos. Las facilidades de ayuda también pueden incluir una o más aplicaciones de ejemplo, que proporcionen a los usuarios una introducción a las capacidades y al modo de utilización del sistema.

## Coherencia

Una consideración de diseño importante en una interfaz es la coherencia. Una cierta forma de ícono debe tener siempre el mismo significado, en lugar de utilizarse para representar diferentes acciones u objetos dependiendo del contexto. Otros ejemplos de coherencia que podríamos citar son colocar los menús en las mismas posiciones relativas, de modo que el usuario no tenga que andar buscando una opción concreta; utilizar siempre la misma combinación de teclas para una acción y emplear siempre la misma codificación de color, de modo que un cierto color no tenga diferentes significados en las distintas situaciones.

## Minimización de la memorización

Las operaciones de la interfaz deben estar también estructuradas de modo que sean fáciles de comprender y de recordar. Los formatos de comando extraños, complicados, incoherentes y abreviados conducen a la confusión y a una reducción en las capacidades de aplicar el software de manera efectiva. Por ejemplo, una tecla o botón utilizados para todas las operaciones de borrado son más fáciles de recordar que un conjunto de diferentes teclas que se empleen para diferentes tipos de procedimientos de borrado.

Los iconos y sistemas de ventana también pueden organizarse para reducir las necesidades de memorización. Los diferentes tipos de información pueden separarse en diferentes ventanas, de modo que el usuario pueda identificar y seleccionar fácilmente los elementos. Los iconos deben diseñarse de manera que sus formas sean fácilmente reconocibles y estén relacionadas con los objetos y acciones de la aplicación. Para seleccionar una acción concreta, el usuario debe poder seleccionar el ícono que «recuerde» a esa acción.

## Cancelación de acciones y tratamiento de errores

Otra característica común de las interfaces es algún tipo de mecanismo para deshacer una secuencia de operaciones, lo que permite al usuario explorar las capacidades del sistema, sabiendo que los efectos de un error pueden corregirse. Normalmente, los sistemas pueden hoy en día deshacer varias operaciones, permitiendo así al usuario reiniciar el sistema en alguna acción especificada. Para aquellas acciones que no pueden deshacerse, como por ejemplo cerrar una aplicación sin guardar los cambios, el sistema suele pedir al usuario que confirme la operación solicitada.

Además, unos adecuados mensajes de diagnóstico y de error ayudan al usuario a determinar la causa de los errores. Las interfaces pueden tratar de minimizar los errores anticipando ciertas acciones que pudieran conducir a un error, y puede proporcionarse una advertencia a los usuarios si están solicitando acciones ambiguas o incorrectas, como por ejemplo tratar de aplicar un procedimiento a múltiples objetos de aplicación.

## Realimentación

Otra característica importante de las interfaces es que deben responder a las acciones de los usuarios, particularmente en el caso de los usuarios no experimentados. A medida que se introduce cada acción, debería darse algún tipo de respuesta. En caso contrario, el usuario puede comenzar a preguntarse qué es lo que el sistema está haciendo y si debe volver a introducir los datos.

La realimentación puede proporcionarse de muchas formas, como por ejemplo resaltando un objeto, mostrando un ícono o un mensaje o visualizando una opción de menú seleccionada en un color distinto. Cuando el procesamiento de una acción solicitada sea largo, la visualización de un mensaje parpadeante, de un reloj,

de un reloj de arena o de algún otro indicador de progreso tiene una importancia crucial. También puede ser posible que el sistema muestre resultados parciales a medida que se completan, de modo que la imagen final se construye de elemento en elemento. El sistema podría también permitir a los usuarios introducir otros comandos u otros datos mientras se está procesando una instrucción.

Se utilizan diseños estándar de ciertos símbolos para determinados tipos de realimentación. Para indicar un error, se utiliza a menudo un aspa, una cara con las cejas fruncidas o un puño con el pulgar hacia abajo, mientras que para indicar que se está procesando una acción suele emplearse algún tipo de símbolo relacionado con el tiempo o alguna señal parpadeante que indique que el sistema «está trabajando». Este tipo de realimentación puede ser muy efectiva con un usuario experto, pero es posible que los principiantes necesiten una realimentación más detallada que no sólo indique claramente qué es lo que el sistema está haciendo, sino también lo que el usuario debe introducir a continuación.

La claridad es otro aspecto de gran importancia en la realimentación. Las respuestas deben ser claramente comprensibles, pero no tan detalladas que interrumpan la concentración del usuario. Con las teclas de función, la realimentación puede proporcionarse en forma de un clic audible o iluminando la tecla que ha sido pulsada. La realimentación sonora tiene la ventaja de que no ocupa espacio de pantalla y no distrae la atención del usuario del área de trabajo. También puede utilizarse un área de mensajes fija, de modo que el usuario siempre sepa dónde buscar los mensajes, aunque en otras ocasiones pueden que sea ventajoso colocar los mensajes de realimentación en el área de trabajo, cerca del cursor. Los mensajes de realimentación pueden también mostrarse en diferentes colores, para distinguirlos de los restantes objetos visualizados.

La realimentación de eco resulta útil en muchas ocasiones, particularmente para la entrada desde teclado, de modo que puedan detectarse rápidamente los errores. Las entradas de botones y diales pueden proporcionarse de la misma como eco. Los valores escalares seleccionados con diales o con reguladores virtuales (representados en pantalla) suelen mostrarse como eco en la pantalla para que el usuario pueda comprobar la precisión de los valores introducidos. La selección de puntos de coordenadas puede también proporcionarse como eco mediante un cursor o un símbolo que aparezca en la posición seleccionada. Para proporcionar un eco más preciso de las posiciones seleccionadas, los valores de las coordenadas podrían también mostrarse en la pantalla.

## 11.9 RESUMEN

---

La entrada de los programas gráficos puede provenir de muchos tipos distintos de dispositivos hardware, pudiendo haber más de un dispositivo proporcionando la misma clase general de datos de entrada. Las funciones de entrada de los paquetes gráficos se suelen diseñar para que sean independientes del hardware, adoptando una clasificación lógica de los dispositivos de entrada. Entonces, los dispositivos se especifican de acuerdo con el tipo de entrada gráfica. Los seis tipos de dispositivos lógicos utilizados en los estándares ISO y ANSI son los dispositivos localizadores, de trazado, de cadena, evaluadores, de elección y de selección. Los dispositivos localizadores introducen una única posición de coordenadas. Los dispositivos de trazado proporcionan como entrada un flujo de coordenadas. Los dispositivos de cadena proporcionan como entrada cadenas de texto. Los dispositivos evaluadores se utilizan para introducir valores escalares. Los dispositivos de elección se utilizan para elegir opciones de menú. Por su parte, los dispositivos selectores permiten seleccionar componentes de la escena. Los paquetes gráficos independientes de los dispositivos incluyen un conjunto limitado de funciones de entrada que se definen en una biblioteca auxiliar.

Para las funciones de entrada se utilizan comúnmente tres modos distintos. El modo de solicitud coloca la entrada bajo control del programa de aplicación. El modo de muestreo permite que los dispositivos de entrada y el programa operen concurrentemente. El modo de sucesos permite que los dispositivos de entrada inicien la introducción de datos y controlen el procesamiento de los mismos. Una vez elegido un modo para una clase de dispositivo lógico y el dispositivo físico concreto que hay que utilizar para introducir dicha clase de datos, se utilizan las funciones de entrada para introducir los valores de los datos dentro del programa. Un pro-

grama de aplicación puede introducir simultáneamente varios dispositivos de entrada físicos que operen en diferentes modos.

Los métodos interactivos de construcción de imágenes se utilizan comúnmente en diversas aplicaciones, incluyendo los paquetes de diseño y de dibujo. Estos métodos proporcionan a los usuarios la capacidad de especificar las posiciones de los objetos, restringir los objetos para que adopten orientaciones o alineaciones predefinidas y dibujar o pintar interactivamente objetos en una escena. Se utilizan métodos tales como las cuadrículas, los campos de gravedad y los métodos de banda elástica como ayuda durante el posicionamiento y durante la realización de otras operaciones de generación de imágenes.

Las interfaces gráficas de usuario son ahora una característica estándar de las aplicaciones software. El software se basa en un procedimiento de diálogo diseñado a partir del modelo del usuario, que describe el propósito y la función del paquete de aplicación. Todos los elementos de ese procedimiento de diálogo se presentan en el lenguaje de la aplicación.

Los sistemas de ventanas proporcionan una interfaz típica, con procedimientos para manipular las ventanas de visualización, los menús y los iconos. Pueden diseñarse sistemas de gestión de ventanas generales que soporten múltiples gestores de ventanas.

Las principales preocupaciones a la hora de diseñar un procedimiento de diálogo con el usuario son la facilidad de uso, la claridad y la flexibilidad. Específicamente, las interfaces gráficas se diseñan tratando de mantener la coherencia en las interacciones con el usuario y de adaptarse a los diferentes niveles de experiencia de los distintos usuarios. Además, las interfaces se diseñan para tratar de reducir los esfuerzos de memorización por parte del usuario, para proporcionar una realimentación suficiente y para proporcionar unas capacidades adecuadas de cancelación de acciones y de tratamiento de errores.

En GLUT (Utility Toolkit), hay disponibles funciones de entrada para dispositivos interactivos tales como ratones, tabletas, spaceballs, cajas de botones y cajas de diales. Además, GLUT proporciona una función para aceptar una combinación de valores de entrada procedentes de un ratón y un teclado. Las operaciones de selección pueden realizarse utilizando funciones de la biblioteca GLU y de la biblioteca básica OpenGL. También podemos mostrar menús emergentes y submenús utilizando un conjunto de funciones de la biblioteca GLUT. En las Tablas 11.1 y 11.2 se proporciona un resumen de las funciones de entrada y de menús de OpenGL.

**TABLA 11.1. RESUMEN DE FUNCIONES DE ENTRADA OpenGL.**

<i>Función</i>	<i>Descripción</i>
glutMouseFunc	Especifica una función de retrollamada para el ratón que será invocada cuando se pulse un botón del ratón.
glutMotionFunc	Especifica una función de retrollamada para el ratón que será invocada cuando se mueva el cursor del ratón mientras está pulsado un botón.
glutPassiveMotionFunc	Especifica una función de retrollamada para el ratón que será invocada cuando se mueva el cursor del ratón sin presionar un botón.
glutKeyboardFunc	Especifica una función de retrollamada para el teclado que será invocada cuando se pulse una tecla estándar.
glutSpecialFunc	Especifica una función de retrollamada para el teclado que será invocada cuando se pulse una tecla de propósito especial (por ejemplo, una tecla de función).
glutTabletButtonFunc	Especifica una función de retrollamada para una tableta que será invocada cuando se pulse un botón de la tableta mientras el cursor del ratón se encuentra en una ventana de visualización.

*Continúa*

**TABLA 11.1.** RESUMEN DE FUNCIONES DE ENTRADA OpenGL. (*Cont.*)

<i>Función</i>	<i>Descripción</i>
glutTabletMotionFunc	Especifica una función de retrollamada para una tableta que será invocada cuando se mueva el cursor o el lápiz de la tableta mientras el cursor del ratón se encuentra en una ventana de visualización.
glutSpaceballButtonFunc	Especifica una función de retrollamada para una spaceball que será invocada cuando se pulse un botón de la spaceball mientras el cursor del ratón se encuentra en una ventana de visualización, o bien utilizando algún otro método de activación de ventanas de visualización.
glutSpaceballMotionFunc	Especifica una función de retrollamada para una <i>spaceball</i> que será invocada cuando se produzca un movimiento de traslación de la <i>spaceball</i> para una ventana de visualización activada.
glutSpaceballRotateFunc	Especifica una función de retrollamada para una <i>spaceball</i> que será invocada cuando se produzca un movimiento giratorio de la <i>spaceball</i> para una ventana de visualización activada.
glutButtonBoxFunc	Especifica una función de retrollamada para cajas de botones que será invocada cuando se pulse un botón.
glutDialsFunc	Especifica una función de retrollamada para un dial que será invocada cuando se gire un dial.
glSelectBuffer	Especifica el tamaño y el nombre del búfer de selección.
glRenderMode	Activa las operaciones de selección utilizando el argumento GL_SELECT. Esta función también se utiliza para activar el modo de representación normal o el modo de realimentación.
glInitNames	Activa la pila de nombres identificadores de los objetos.
glPushName	Introduce un identificador de un objeto en la pila de identificadores.
glLoadName	Sustituye el identificador situado en la parte superior de la pila por otro valor especificado.
glPopName	Elimina el elemento superior de la pila de identificadores.
gluPickMatrix	Define una ventana de selección y construye un nuevo volumen de visualización para las operaciones de selección.

**TABLA 11.2.** RESUMEN DE FUNCIONES DE MENÚ OpenGL.

<i>Función</i>	<i>Descripción</i>
glutCreateMenu	Crea un menú emergente y especifica el procedimiento que habrá que invocar cuando se seleccione un elemento del menú; se asigna un identificador entero al menú creado.
glutAddMenuEntry	Especifica una opción que hay que incluir en un menú emergente.
glutAttachMenu	Especifica el botón del ratón que se utilizará para seleccionar las opciones de menú.

*Continúa*

**TABLA 11.2.** RESUMEN DE FUNCIONES DE MENÚ OpenGL. (*Cont.*)

<i>Función</i>	<i>Descripción</i>
glutSetMenu	Especifica el menú actual para la ventana de visualización actual.
glutDestroyMenu	Especifica el identificador de un menú que se quiere eliminar.
glutGetMenu	Devuelve el identificador del menú actual asociado a la ventana actual.
glutAddSubMenu	Especifica un submenú que hay que incluir en otro menú, debiendo haberse definido el submenú indicado mediante la rutina glutCreateMenu.
glutDetachMenu	Cancela la asociación especificada de botón de ratón para el menú actual.
glutRemoveMenuItem	Borra una opción especificada en el menú actual.

## REFERENCIAS

La evolución del concepto de los dispositivos de entrada lógicos (o virtuales) se analiza en Wallace (1976) y en Rosenthal, Michener, Pfaff, Kessener y Sabin (1982). Puede encontrar implementaciones para diversos procedimientos de entrada en Glassner (1990), Arvo (1991), Kirk (1992), Heckbert (1994) y Paeth (1995). También puede encontrar ejemplos tradicionales de programación utilizando la entrada de ratón y de teclado en Woo, Neider, Davis y Shreiner (1999). Para ver una lista completa de funciones de la biblioteca básica OpenGL y de la biblioteca GLU, consulte Shreiner (2000). Las funciones de entrada y de menús de GLUT se tratan con detalle en Kilgard (1996). Puede encontrar directrices para el diseño de interfaces de usuario en Schneiderman (1986), Apple (1987), Bleser (1988), Brown y Cunningham (1989), Digital (1989), OSF/MOTIF (1989) y Laurel (1990). Para obtener información sobre el sistema XWindow, consulte Young (1990) y Cutler, Gilly y Reilly (1992).

## EJERCICIOS

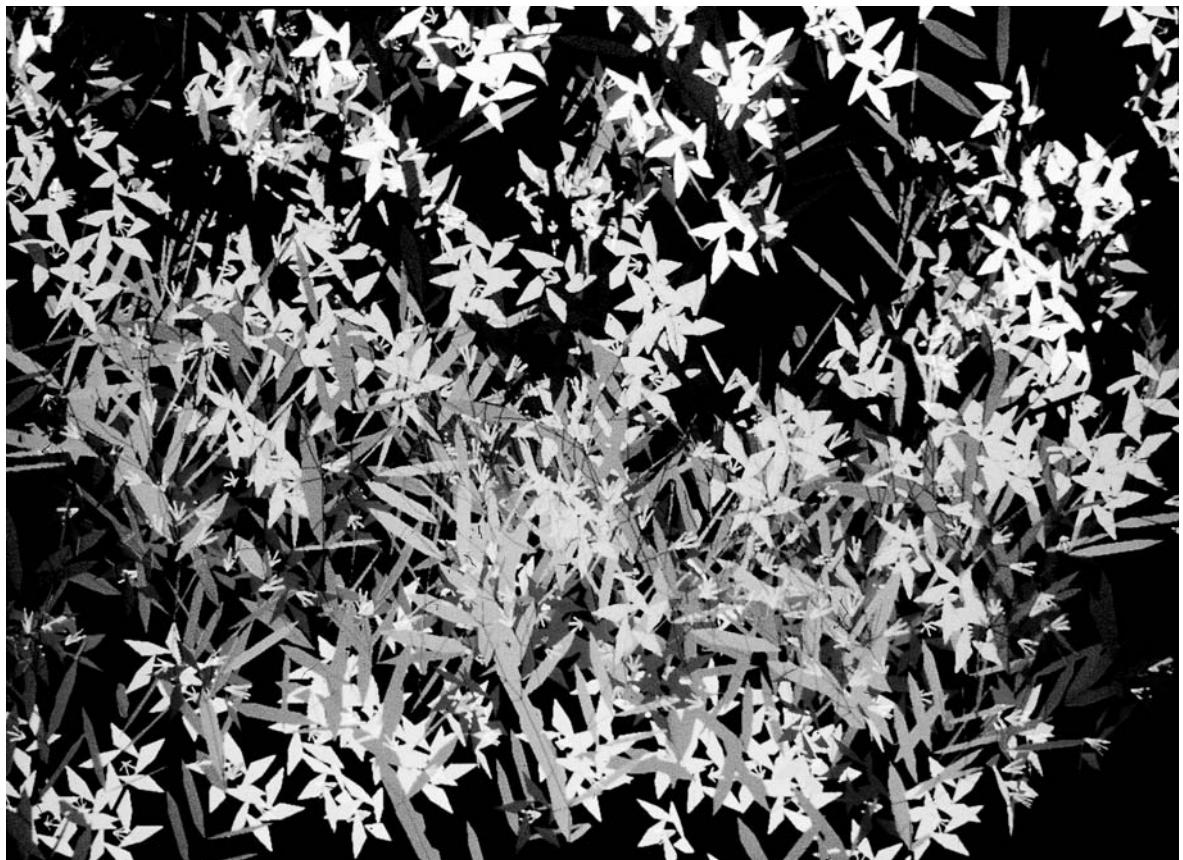
- 11.1 Diseñe un algoritmo que permita situar objetos en la pantalla utilizando un dispositivo localizador. Hay que presentar un menú de objetos con formas geométricas al usuario, el cual seleccionará un objeto y la posición donde hay que colocarlo. El programa debe permitir la colocación de un número arbitrario de objetos, hasta que se proporcione una señal de «terminación».
- 11.2 Amplíe el algoritmo del ejercicio anterior para poder cambiar la escala de los objetos y girarlos antes de colocarlos. Las opciones de transformación y los parámetros de transformación deben presentarse al usuario como opciones de menú.
- 11.3 Defina un procedimiento para dibujar interactivamente objetos utilizando un dispositivo de trazado.
- 11.4 Explique los métodos que podrían emplearse en un procedimiento de reconocimiento de patrones para hacer corresponder los caracteres de entrada con una biblioteca almacenada de formas.
- 11.5 Escriba una rutina que muestre una escala lineal y un deslizador en la pantalla y que permita seleccionar valores numéricos posicionando el deslizador a lo largo de la escala. Debe proporcionarse como eco el valor numérico seleccionado, dentro de un recuadro que se muestre cerca de la escala lineal.
- 11.6 Escriba una rutina que muestre una escala circular y un puntero o un deslizador que puedan moverse por el círculo con el fin de seleccionar ángulos (en grados). Debe proporcionarse como eco el valor angular seleccionado, dentro de un recuadro que se muestre cerca de la escala circular.

- 11.7 Escriba un programa de dibujo que permita a los usuarios crear una imagen mediante un conjunto de segmentos lineales dibujados entre puntos especificados. Las coordenadas de los segmentos de línea individuales se seleccionarán mediante un dispositivo localizador.
- 11.8 Escriba un programa de dibujo que permita crear imágenes mediante segmentos de línea recta dibujados entre puntos especificados. Defina un campo de gravedad alrededor de cada línea de la imagen, como ayuda para conectar las nuevas líneas con las líneas existentes.
- 11.9 Modifique el programa de dibujo del ejercicio anterior para poder restringir las líneas horizontal o verticalmente.
- 11.10 Escriba un programa de dibujo que pueda mostrar un patrón de cuadrícula opcional, de modo que las posiciones de pantalla seleccionadas se ajusten para que coincidan con intersecciones de la cuadrícula. El programa debe proporcionar capacidades de dibujo de líneas, seleccionándose los extremos de las líneas mediante un dispositivo localizador.
- 11.11 Escriba una rutina que permita al diseñador crear una imagen dibujando líneas rectas mediante un método de banda elástica.
- 11.12 Diseñe un programa de dibujo que permita construir líneas rectas, rectángulos y círculos utilizando métodos de banda elástica.
- 11.13 Escriba un procedimiento que permita al usuario seleccionar componentes de una escena bidimensional. Las extensiones de coordenadas de cada objeto deben almacenarse y utilizarse para identificar el objeto seleccionado a partir de la posición del cursor suministrada como entrada.
- 11.14 Desarrolle un procedimiento que permita al usuario diseñar una imagen a partir de un menú de formas básicas visualizadas, arrastrando cada forma seleccionada hasta su posición de destino mediante un dispositivo de selección.
- 11.15 Diseñe una implementación de las funciones de entrada para el modo de solicitud.
- 11.16 Diseñe una implementación de las funciones de entrada para el modo de muestreo.
- 11.17 Diseñe una implementación de las funciones de entrada para el modo de sucesos.
- 11.18 Diseñe un procedimiento para implementar funciones de entrada para los modos de solicitud, muestreo y sucesos.
- 11.19 Amplíe el programa OpenGL de dibujo de puntos de la Sección 11.6 para incluir un menú que permita al usuario seleccionar el tamaño del punto y el color del punto.
- 11.20 Amplíe el programa OpenGL de polilíneas de la Sección 11.6 para incluir un menú que permita al usuario seleccionar los atributos de la línea: tamaño, color y anchura.
- 11.21 Modifique el programa del ejercicio anterior para que se pueda elegir un patrón de textura para la polilínea.
- 11.22 Escriba un programa OpenGL interactivo para mostrar un rectángulo de 100 por 100 píxeles en cualquier posición que se suministre como entrada, dentro de una ventana de visualización. La posición de entrada será el centro del rectángulo. Incluya un menú de opciones de color para mostrar el rectángulo con un color homogéneo.
- 11.23 Modifique el programa del ejercicio anterior de modo que se rechace la posición de entrada si no se puede mostrar todo el rectángulo dentro de la ventana de visualización.
- 11.24 Modifique el programa del ejercicio anterior para incluir un menú de opciones de textura para el rectángulo. Utilice como mínimo dos patrones de textura.
- 11.25 Diseñe un programa OpenGL interactivo para mostrar una cadena de caracteres de entrada en cualquier posición dentro de una ventana de visualización. La posición de entrada será la posición de inicio del texto.
- 11.26 Escriba un programa OpenGL interactivo para posicionar un único objeto bidimensional en cualquier posición dentro de una ventana de visualización. El objeto se seleccionará de entre un menú de formas básicas, que deberá incluir (como mínimo) un cuadrado, un círculo y un triángulo.
- 11.27 Modifique el programa del ejercicio anterior para permitir que se visualice cualquier composición de los anteriores objetos bidimensionales, seleccionándose cada objeto del menú hasta que se elija la opción de menú que permita abandonar el programa.

- 11.28 Modifique el programa del ejercicio anterior para permitir cambiar la escala de los objetos o girarlos. Las operaciones de transformación geométrica deberán enumerarse en un menú.
- 11.29 Escriba un programa OpenGL interactivo para posicionar un único objeto tridimensional dentro de una ventana de visualización. El objeto se seleccionará en un menú de sólidos GLUT alámbricos, compuesto por ejemplo de una esfera, un cubo y un cilindro, y deberá estar centrado el objeto en la posición que se indique como entrada.
- 11.30 Modifique el programa del ejercicio anterior para poder visualizar los objetos en modo sólido o alámbrico. Para la visualización de objetos sólidos, incluya una fuente de luz puntual en la posición de visualización y utilice los parámetros predeterminados de iluminación y de sombreado superficial.
- 11.31 Escriba un programa para implementar las operaciones de selección OpenGL para una escena tridimensional que contenga varios objetos. Para cada selección, cree una pequeña ventana de selección y traiga al frente el objeto más distante contenido dentro de dicha ventana de selección.
- 11.32 Escriba un programa OpenGL interactivo para mostrar una curva de Bézier cónica bidimensional. Las posiciones de los cuatro puntos de control se seleccionarán mediante el ratón.
- 11.33 Modifique el programa del ejercicio anterior para mostrar una curva de Bézier cuyo grado será seleccionable, pudiendo ser de grado tres, cuatro o cinco.
- 11.34 Escriba un programa OpenGL interactivo para mostrar una B-spline cónica bidimensional. Los parámetros de la *spline* se proporcionarán como entrada y los puntos de control se seleccionarán con el ratón.
- 11.35 Escriba un programa OpenGL interactivo para mostrar un parche superficial cúbico de Bézier. Las coordenadas *x* e *y* de los puntos de control podrán seleccionarse con el ratón y la coordenada *z* puede proporcionarse como una altura por encima de un plano de tierra.
- 11.36 Seleccione alguna aplicación gráfica con la que esté familiarizado y defina un modelo de usuario que pueda servir como base para el diseño de una interfaz de usuario para las aplicaciones gráficas utilizadas en dicha área.
- 11.37 Enumere las posibles facilidades de ayuda que pueden proporcionarse en una interfaz de usuario y explique qué tipos de ayuda son más apropiados para los distintos niveles de experiencia de los usuarios.
- 11.38 Resuma los métodos utilizados para el tratamiento de errores y para la cancelación de operaciones. ¿Qué métodos son adecuados para un principiante? ¿Qué métodos son mejores para un usuario experimentado?
- 11.39 Enumere los posibles formatos de presentación de menús a un usuario y explique en qué circunstancias puede ser apropiado para cada uno.
- 11.40 Explique las alternativas de realimentación existentes en relación con los diversos niveles de experiencia de los usuarios.
- 11.41 Enumere las funciones que debe proporcionar un gestor de ventanas para gestionar la disposición de pantalla cuando existen múltiples ventanas solapadas.
- 11.42 Diseñe un paquete de gestión de ventanas.
- 11.43 Diseñe una interfaz de usuario para un programa de dibujo.
- 11.44 Diseñe una interfaz de usuario para un paquete de modelado jerárquico de dos niveles.



# Modelos y aplicaciones del color



Una escena de flores generada por computadora, modelada con diversas combinaciones de color y una forma básica de pétalo. (*Cortesía de Przemyslaw Prusinkiewicz, University of Calgary. © 1987.*)

- |   |  |
|---|--|
| <b>12.1</b> Propiedades de la luz                             | <b>12.6</b> Los modelos de color CMY y CMYK    |
| <b>12.2</b> Modelos de color                                  | <b>12.7</b> El modelo de color HSV             |
| <b>12.3</b> Primarios estándar y diagrama cromático           | <b>12.8</b> El modelo de color HLS             |
| <b>12.4</b> El modelo de color RGB                            | <b>12.9</b> Selección y aplicaciones del color |
| <b>12.5</b> El modelo de color YIQ y los modelos relacionados | <b>12.10</b> Resumen                           |

Nuestras explicaciones sobre el color hasta este momento se han concentrado en el modelo de color RGB, que es el que se utiliza para generar las imágenes en los monitores de vídeo. Hay varios otros modelos de color que también son útiles en las aplicaciones infográficas. Algunos de los modelos se utilizan para describir la salida de color a través de impresoras y trazadoras gráficas. Otros se emplean para transmitir y almacenar información de color y otros, en fin, se usan para proporcionar a los programas una interfaz más intuitiva para el manejo de los parámetros de color.

## 12.1 PROPIEDADES DE LA LUZ

---

Como hemos indicado en los capítulos anteriores, la luz exhibe muchas características distintas y podemos describir las propiedades de la luz en diferentes formas en diferentes contextos. Físicamente, podemos caracterizar la luz como energía radiante, pero también necesitamos otros conceptos si queremos describir nuestra percepción de la luz.

### El espectro electromagnético

En términos físicos, el color es una radiación electromagnética dentro de una estrecha banda de frecuencias. Algunos de los otros grupos de frecuencias del espectro electromagnético son los de las ondas de radio, las microondas, las ondas infrarrojas y los rayos X. La Figura 12.1 muestra los rangos de frecuencia aproximados para estos distintos tipos de radiación electromagnética.

Cada valor de frecuencia dentro de la región visible del espectro electromagnético se corresponde con un **colorpectral** distinto. En el extremo de baja frecuencia (aproximadamente  $3.8 \times 10^{14}$  hercios) se encuentran los colores rojos, mientras que en el extremo de alta frecuencia (aproximadamente  $7.9 \times 10^{14}$  hercios) encontramos los colores violetas. En la realidad, el ojo humano también es sensible a algunas frecuencias de las bandas infrarroja y ultravioleta. Los colores especiales van variando desde las tonalidades de rojo, a través del naranja y el amarillo (en el extremo de baja frecuencia), pasando luego por el verde, el azul y el violeta (en el extremo alto).

En el modelo ondulatorio de la radiación electromagnética, la luz puede describirse como un campo electromagnético que oscila de manera transversal y se propaga a través del espacio. Los campos eléctricos y magnéticos oscilan en direcciones perpendiculares entre sí y a la dirección de propagación. Para cada color spectral, la frecuencia de oscilación de la magnitud del campo está dada por la frecuencia  $f$ . La Figura 2.2 ilustra las oscilaciones variables en el tiempo de la magnitud del campo eléctrico dentro de un plano. El tiempo entre dos posiciones consecutivas de la onda que tienen la misma amplitud se denomina *periodo*  $T = 1/f$  de la onda. Y la distancia que la onda viaja entre el comienzo de una oscilación y el comienzo de la siguiente se denomina *longitud de onda*  $\lambda$ . Para un color spectral (una onda monocromática), la longitud de onda y

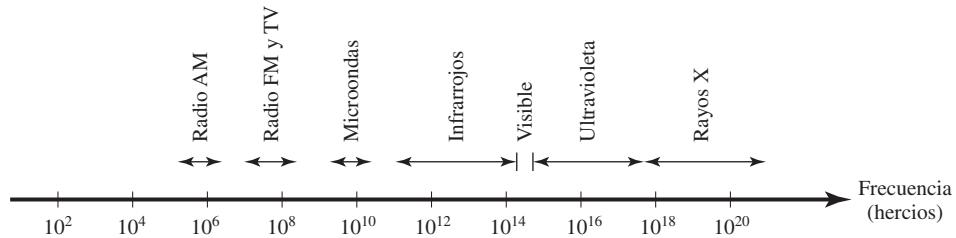


FIGURA 12.1. Espectro electromagnético.

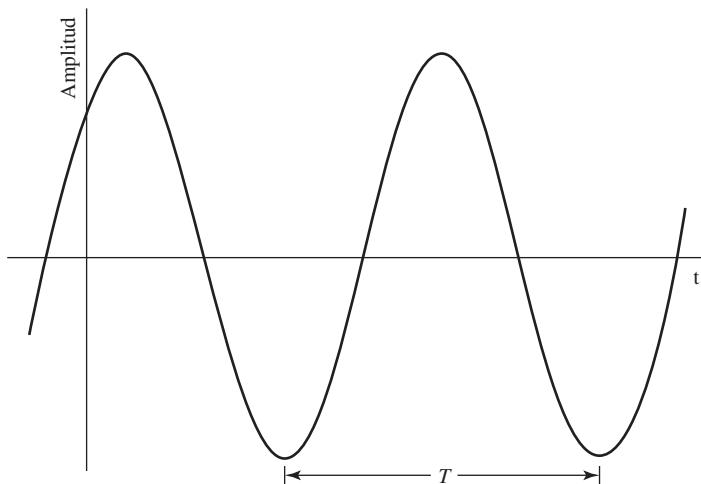


FIGURA 12.2. Variación en el tiempo de la amplitud en el campo eléctrico de una onda electromagnética con polarización plana. El tiempo comprendido entre dos picos consecutivos de amplitud o dos mínimos consecutivos de amplitud se denomina período de la onda.

la frecuencia son inversamente proporcionales entre sí, siendo la constante de proporcionalidad la velocidad de la luz  $c$ :

$$c = \lambda f \quad (12.1)$$

La frecuencia para cada color espectral es una constante para todos los materiales, pero la velocidad de la luz y la longitud de onda dependen de cada material. En el vacío, la velocidad de la luz es muy aproximadamente  $c = 3 \times 10^{10}$  cm/sec. Las longitudes de onda de la luz son muy pequeñas, por lo que las unidades de luz utilizadas para designar los colores espectrales suelen proporcionarse en *angstroms* ( $1 \text{ \AA} = 10^{-8}$  cm) o en nanómetros ( $1 \text{ nm} = 10^{-7}$  cm). Un término equivalente para el nanómetro es el de milimicra. La luz en el extremo de baja frecuencia del espectro (roja) tiene una longitud de onda de aproximadamente 780 nanómetros (nm), mientras que la longitud de onda del otro extremo del espectro (violeta) es de unos 380 nm. Puesto que las unidades de longitud de onda son algo más cómodas de utilizar que las de frecuencia, los colores espectrales se suelen escenificar en términos de su longitud de onda en el vacío.

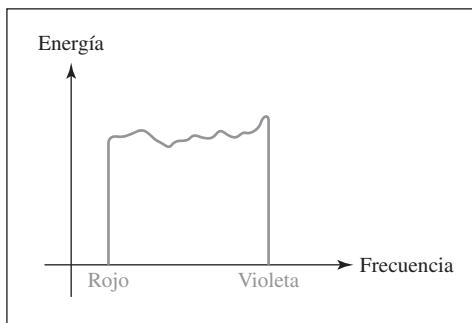
Una fuente luminosa como el Sol o como una bombilla doméstica emite todas las frecuencias dentro del rango visible para producir luz blanca. Cuando la luz blanca incide sobre un objeto opaco, algunas de las frecuencias son reflejadas y otras son absorvidas. La combinación de frecuencias presentes en la luz reflejada determina lo que nosotros percibimos como color del objeto. Si las bajas frecuencias son las predominantes en la luz reflejada, describiremos el objeto como rojo. En este caso, decimos que la luz percibida tiene una

**frecuencia dominante** (o **longitud de onda dominante**) en el extremo rojo del espectro. La frecuencia dominante se denomina también **tono**, o simplemente **color**, de la luz.

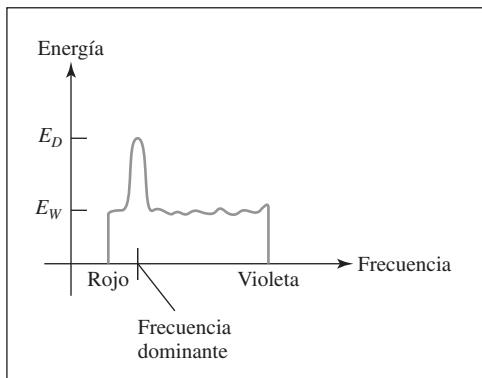
## Características psicológicas del color

Necesitamos otras propiedades, además de la frecuencia, para caracterizar nuestra percepción de la luz. Cuando observamos una fuente luminosa, nuestros ojos responden al color (o frecuencia dominante) y a otras dos sensaciones básicas. Una de estas es lo que denominamos **brillo**, que se corresponde con la energía lumínosa total y puede cuantificarse como la luminancia de la luz (Sección 10.3). La tercera característica percibida se denomina **pureza** o **saturación** de la luz. La pureza describe hasta qué punto una determinada radiación lumínosa nos parece un colorpectral puro, como por ejemplo el rojo. Los colores pálidos y los colores pastel tienen una baja pureza (baja saturación) con lo que parecen casi blancos. Otro término, la **cromaticidad**, se utiliza para referirse conjuntamente a las dos propiedades que describen las características de un color: pureza y frecuencia dominante (tono).

La radiación emitida por una fuente luminosa blanca tiene una distribución de energía que puede representarse en el rango de frecuencias visibles como se muestra en la Figura 12.3. Cada componente de frecuencia dentro del rango que va del rojo al violeta contribuye más o menos en la misma cantidad a la energía total, y el poder de la fuente se describe como blanco. Cuando hay presente una frecuencia dominante, la distribución de energía de la fuente toma una forma como la de la Figura 12.4. Este haz luminoso los describiríamos como de color rojo (la frecuencia dominante), con un valor relativamente alto de pureza. La densidad de energía de la componente dominante de la luz está etiquetada como  $E_D$  en esta figura, y las contribuciones de las otras frecuencias producen una luz blanca con densidad de energía  $E_W$ . Podemos calcular el brillo de la fuente como el área comprendida bajo la curva, que nos da la densidad total de energía emitida. La pureza (saturación) depende de la diferencia entre  $E_D$  y  $E_W$ . Cuanto mayor sea la energía  $E_D$  de la frecuencia dominante, comparada con la componente de luz blanca  $E_W$ , mayor será la pureza de la luz. Tendremos una pureza del 100 por cien cuando  $E_W = 0$  y una pureza del 0 por cien cuando  $E_W = E_D$ .



**FIGURA 12.3.** Distribución de energía de una fuente de luz blanca.



**FIGURA 12.4.** Distribución de energía para una fuente lumínica con una frecuencia dominante cerca del extremo rojo del rango de frecuencias.

## 12.2 MODELOS DE COLOR

Cualquier método utilizado para explicar las propiedades o el comportamiento del color dentro de un contexto concreto se denomina **modelo de color**. No hay un único modelo que pueda explicar todos los aspectos del color, por lo que se utilizan diferentes modelos como ayuda para describir las diferentes características del color.

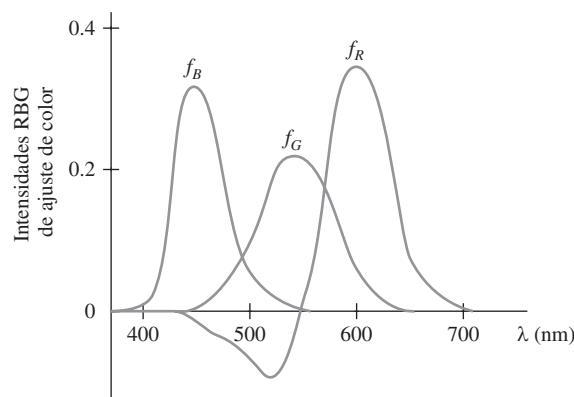
### Colores primarios

Cuando combinamos la luz de dos o más fuentes con diferentes frecuencias dominantes, podemos variar la cantidad (intensidad) de luz de cada fuente para generar un rango de colores adicionales. Esto representa un posible método para formar un modelo de color. Los tonos que elegimos para las fuentes se denominan **colores primarios** y la **gama de colores** del modelo será el conjunto de todos los colores que podemos producir a partir de los colores primarios. Dos primarios que se combinan para generar el color blanco se denominan **colores complementarios**. Como ejemplos de parejas de colores complementarios podemos citar el rojo y el cian, el verde y el magenta y el azul y el amarillo.

No existe ningún conjunto finito de colores primarios reales que puedan combinarse para producir todos los posibles colores visibles. Sin embargo, tres primarios resultan suficientes para la mayoría de los propósitos y los colores que no están presentes en la gama de colores correspondientes a un conjunto especificado de primarios pueden, de todos modos, describirse utilizando extensiones a los métodos. Dado un conjunto de tres colores primarios, podemos caracterizar cualquier cuarto color utilizando procesos de mezcla de colores. Así, una mezcla de uno o dos de los primarios con el cuarto color puede utilizarse para que se ajuste a alguna combinación de los restantes primarios. En este sentido ampliado, podemos considerar que un conjunto de tres colores primarios puede describir todos los colores. La Figura 12.5 muestra un conjunto de *funciones ajuste de color* para tres primarios y la cantidad de cada uno que hace falta para producir cualquier color espectral. Las cuerdas dibujadas en la Figura 12.5 se han obtenido promediando las opiniones de un gran número de observadores. Los colores en la vecindad de 500 nm sólo pueden generarse «restando» una cierta cantidad de luz roja de una combinación de luz azul y luz verde. Esto significa que un color situado en torno a 500 nm sólo puede describirse combinando dicho color con una cierta cantidad de luz roja con el fin de producir la combinación azul-verde especificada en el diagrama. Así, un monitor color RGB no puede mostrar colores en la vecindad de 500 nm.

### Conceptos intuitivos del color

Un artista crea una pintura en color mezclando pigmentos coloreados con pigmentos blancos y negros con el fin de formar las diversas sombras, tintas y tonos de la escena. Comenzando con el pigmento de un «color



**FIGURA 12.5.** Tres funciones de ajuste de color para mostrar frecuencias espectrales dentro del rango aproximado que va de 400 nm a 700 nm.

puro» («tono puro»), el artista añade pigmento negro para generar distintas **sombra**s de dicho color. Cuanto más pigmento negro añada, más oscura será la sombra del color. De forma similar, las diferentes **tintas** del color se obtienen añadiendo pigmento blanco al color original, haciéndolo más claro cuanta mayor cantidad de blanco se añada. Los **tonos** del color se obtienen añadiendo pigmento tanto blanco como negro.

Para muchas personas, estos conceptos de color son más intuitivos que describir un color mediante un conjunto de tres números que nos den las proporciones relativas de los colores primarios. Generalmente, es mucho más fácil pensar en crear un color rojo pastel añadiendo blanco a un rojo puro y generar un color azul oscuro añadiendo negro a un azul puro. Por esto, los paquetes gráficos que proporcionan paletas de color a los usuarios suelen utilizar dos o más modelos de color. Uno de los modelos proporciona una interfaz de color intuitiva para el usuario y el otro describe los componentes de color para los dispositivos de salida.

## 12.3 PRIMARIOS ESTÁNDAR Y DIAGRAMA CROMÁTICO

---

Puesto que no puede combinarse ningún conjunto finito de puntos luminosos para mostrar todos los colores posibles, en 1931 la CIE (Commission Internationale de l'Éclairage, Comisión Internacional de Iluminación) definió tres primarios estándar. Estos tres primarios son colores imaginarios, que se definen matemáticamente con funciones positivas de ajuste de color (Figura 12.6) que especifican la cantidad de cada primario necesaria para describir cualquier color espectral. Esto proporciona una definición internacional estándar para todos los colores, y los primarios del CIE eliminan el ajuste de color con valores negativos y otros problemas asociados con la selección de un conjunto de primarios reales.

### El modelo de color XYZ

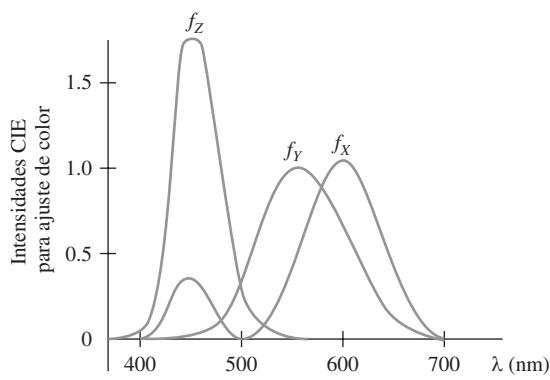
El conjunto de primarios CIE se suele denominar modelo de color XYZ, donde los parámetros  $X$ ,  $Y$  y  $Z$  representan la cantidad de cada primario CIE necesaria para producir el color seleccionado. Así, un color se describe con el modelo XYZ de la misma forma en que describimos un color utilizando el modelo RGB.

En el espacio de color tridimensional XYZ, representamos cualquier color  $C(\lambda)$  como:

$$C(\lambda) = (X, Y, Z) \quad (12.2)$$

donde  $X$ ,  $Y$  y  $Z$  se calculan a partir de las funciones de ajuste de color (Figura 12.6):

$$\begin{aligned} X &= k \int_{\text{visible } \lambda} f_X(\lambda) I(\lambda) d\lambda \\ Y &= k \int_{\text{visible } \lambda} f_Y(\lambda) I(\lambda) d\lambda \\ Z &= k \int_{\text{visible } \lambda} f_Z(\lambda) I(\lambda) d\lambda \end{aligned} \quad (12.3)$$



**FIGURA 12.6.** Las tres funciones de ajuste de color para los primarios de la CIE.

El parámetro  $k$  en estos cálculos tiene el valor de 683 lumens/vatio, donde el lumen es una unidad de medida para la radiación luminosa por unidad de ángulo sólido desde una fuente luminosa puntual «estándar» (que antiguamente se denominaba *candela*). La función  $I(\lambda)$  representa la radiancia espectral, que es la intensidad luminosa seleccionada en una dirección concreta, y la función de ajuste de color  $f_Y$  se elige de tal de manera que el parámetro  $Y$  es la luminancia (Ecuación 10.26) de dicho color. Los valores de luminancia se ajustan normalmente para que ocupen el rango de 0 a 100.0, donde 100.0 representa la luminancia de la luz blanca.

Cualquier color puede representarse en el espacio de color XYZ como una combinación aditiva de los primarios utilizando los vectores unitarios  $\mathbf{X}$ ,  $\mathbf{Y}$ ,  $\mathbf{Z}$ . Así, podemos escribir la Ecuación 12.2 de la forma:

$$C(\lambda) = X \mathbf{X} + Y \mathbf{Y} + Z \mathbf{Z} \quad (12.4)$$

### Valores XYZ normalizados

Al hablar de las propiedades del color, resulta conveniente normalizar los valores de la Ecuación 12.3 con respecto a la suma  $X + Y + Z$ , que representa la energía luminosa total. Así, los valores normalizados se calculan de la forma siguiente:

$$x = \frac{X}{X+Y+Z}, \quad y = \frac{Y}{X+Y+Z}, \quad z = \frac{Z}{X+Y+Z} \quad (12.5)$$

Puesto que  $x + y + z = 1$ , cualquier color puede representarse simplemente con los valores  $x$  e  $y$ . Asimismo, hemos efectuado la normalización con respecto a la energía total, por lo que los parámetros  $x$  e  $y$  dependen sólo del tono y de la pureza y se denominan **valores de cromaticidad**. Sin embargo, los valores  $x$  e  $y$  no nos permiten por sí solos describir completamente todas las propiedades del color, y no podemos obtener los valores  $X$ ,  $Y$  y  $Z$ . Por tanto, una descripción completa de un color se suele proporcionar mediante los tres valores  $x$ ,  $y$  y la luminancia  $Y$ . Los valores restantes del modelo de la CIE se calculan entonces mediante las fórmulas

$$X = \frac{x}{y} Y, \quad Z = \frac{z}{y} Y \quad (12.6)$$

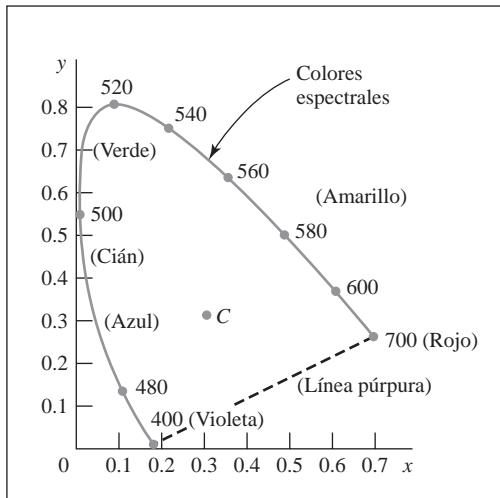
donde  $z = 1 - x - y$ . Utilizando las coordenadas de cromaticidad  $(x, y)$ , podemos representar todos los colores en un diagrama bidimensional.

### Diagrama cromático de la CIE

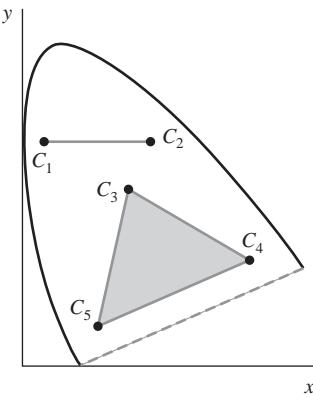
Cuando dibujamos los valores normalizados  $x$  e  $y$  para los colores del espectro visible, obtenemos la curva con forma aproximada de parábola mostrada en la Figura 12.7. Esta curva se denomina **diagrama cromático CIE**. Los puntos de la curva son los colores espectrales (colores puros). La línea que une los puntos espirales correspondientes al rojo y al violeta, denominada *línea púrpura*, no forma parte del espectro. Los puntos interiores representan todas las posibles combinaciones de colores visibles. El punto  $C$  del diagrama corresponde a la posición de la luz blanca. En realidad, este punto se dibuja para una fuente luminosa blanca conocida con el nombre **iluminante C**, que se utiliza como aproximación estándar para la luz diurna promedio.

Los valores de luminancia no están disponibles en el diagrama cromático, debido a la normalización. Distintos colores con diferente luminancia pero con la misma cromaticidad se representarán mediante el mismo punto. El diagrama cromático resulta útil para:

- Comparar las barras de colores correspondientes a diferentes conjuntos de primarios.
- Identificar colores complementarios.
- Determinar la pureza y la longitud de onda dominante de un color especificado.



**FIGURA 12.7.** Diagrama cromático CIE para los colores espectrales comprendidos entre 400 nm y 700 nm.



**FIGURA 12.8.** Gamas de colores definidos en el diagrama cromático para un sistema de primarios de dos colores y de tres colores.

## Gamas de colores

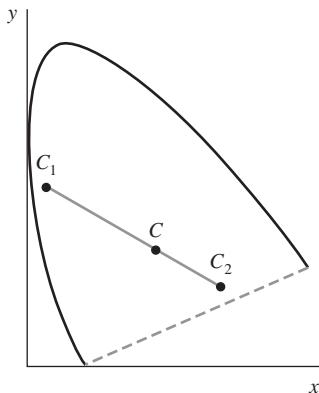
Podemos identificar las gamas de colores en el diagrama cromático como segmentos de línea recta o regiones poligonales. Todos los colores situados en la línea recta que une las posiciones  $C_1$  y  $C_2$  en la Figura 12.8 pueden obtenerse mezclando las cantidades apropiadas de dichos colores  $C_1$  y  $C_2$ . Si se utiliza una mayor proporción de  $C_1$ , el color resultante estará más próximo a  $C_1$  que a  $C_2$ . La gama de colores para tres puntos, como por ejemplo  $C_3$ ,  $C_4$  y  $C_5$  en la Figura 12.8, es un triángulo con sus vértices situados en las posiciones de dichos tres colores. Estos tres primarios pueden generar únicamente los colores situados dentro del triángulo o en sus aristas de contorno. Así, el diagrama cromático nos ayuda a comprender por qué ningún conjunto de tres primarios puede combinarse de forma aditiva para generar todos los colores, ya que ningún triángulo del diagrama puede abarcar todos los colores del mismo. Podemos comparar cómodamente mediante el diagrama cromático las gamas de colores de los monitores de vídeo y de los dispositivos de obtención de copias impresas.

## Colores complementarios

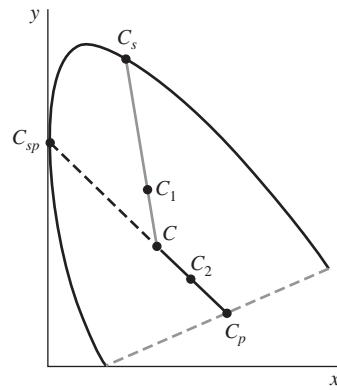
Puesto que la gama de colores para dos puntos es una línea recta, los colores complementarios deben estar representados en el diagrama de cromaticidad como dos puntos situados en lados opuestos de  $C$  y colineales con  $C$ , como en la Figura 12.9. Las distancias de los dos colores  $C_1$  y  $C_2$  a  $C$  determinan la cantidad de cada color necesaria para producir luz blanca.

## Longitud de onda dominante

Para determinar la longitud de onda dominante de un color, dibujamos una línea recta desde  $C$  a través de dicho punto de color, hasta alcanzar un colorpectral en la curva cromática. El colorpectral  $C_s$  en la Figura 2.10 es la longitud de onda dominante para el color  $C_1$  en este diagrama. Así, el color  $C_1$  puede representarse como una combinación de la luz blanca  $C$  y el colorpectral  $C_s$ . Este método para determinar la longitud de onda dominante no sirve para los puntos de color que se encuentren entre  $C$  y la línea púrpura. Si dibujamos una línea desde  $C$  a través del punto  $C_2$  en la Figura 12.10, llegamos al punto  $C_p$  en la línea púrpura, que no se encuentra en el espectro visible. En este caso, tomamos el complementario de  $C_p$  en la curva espectral,



**FIGURA 12.9.** Representación de colores complementarios en el diagrama cromático.



**FIGURA 12.10.** Determinación de la longitud de onda dominante y de la pureza utilizando el diagrama cromático.

que será el punto  $C_{sp}$ , como longitud de onda dominante. Los colores como  $C_2$  en este diagrama tienen distinciones espetrales como longitudes de onda dominantes substractivas. Podemos describir dichos colores restando de la luz blanca la longitud de onda dominante espectral.

### Pureza

Para un punto de color tal como  $C_1$  en la Figura 12.10, determinamos la pureza como la distancia relativa de  $C_1$  a  $C$  según la línea recta que une  $C$  con  $C_s$ . Si  $d_{c1}$  denota la distancia desde  $C$  a  $C_1$  y  $d_{cs}$  es la distancia desde  $C$  a  $C_s$ , podemos representar la pureza como el cociente  $d_{c1}/d_{cs}$ . El color  $C_1$  en esta figura tiene una pureza aproximadamente del 25 por ciento, ya que está situado aproximadamente a un cuarto de la distancia total de  $C$  a  $C_s$ . En la posición  $C_s$ , el punto de color sería 100 por cien puro.

## 12.4 EL MODELO DE COLOR RGB

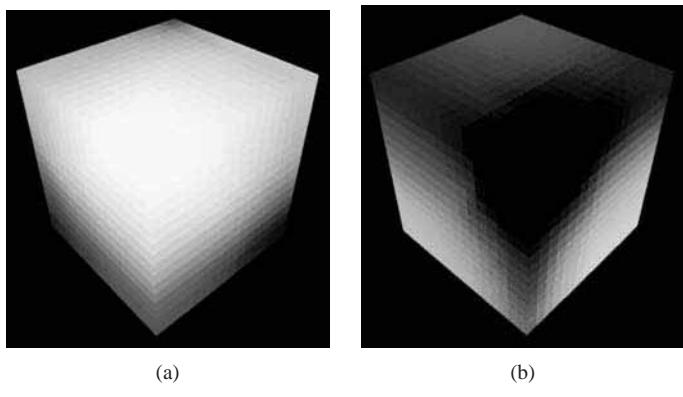
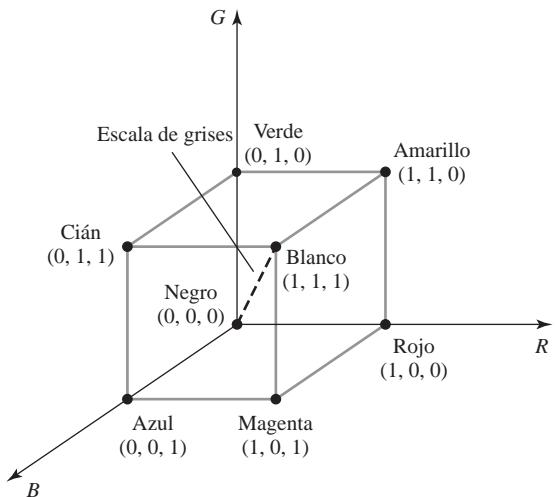
De acuerdo con la *teoría de los tres estímulos* de la visión, nuestros ojos perciben el color mediante la estimación de tres pigmentos visuales en los conos de la retina. Uno de los pigmentos es más sensible a la luz con una longitud de onda de unos 630 nm (roja), otro tiene su pico de sensibilidad para unos 530 nm (verde) y el tercer pigmento es especialmente receptivo a la luz con una longitud de onda de unos 450 nm (azul). Comparando las intensidades de una fuente luminosa, podemos percibir el color de la luz. Esta teoría de la visión es la base para la visualización de los colores en un monitor de video utilizando los tres primarios rojo, verde y azul, que es lo que se denomina modelo de color RGB.

Podemos representar este modelo utilizando el cubo unitario definido sobre sendos ejes  $R$ ,  $G$  y  $B$ , como se muestra en la Figura 12.11. El origen representa el color negro y el vértice diagonalmente opuesto, con coordenadas  $(1, 1, 1)$ , es el blanco. Los vértices del cubo situados sobre los ejes representan los colores primarios y los restantes vértices son los puntos de color complementario para cada uno de los colores primarios.

Al igual que con el sistema de color XYZ, el esquema de color RGB es un modelo aditivo. Cada punto de color dentro del cubo unitario puede representarse como una suma vectorial ponderada de los colores primarios, utilizando los vectores unitario  $\mathbf{R}$ ,  $\mathbf{G}$  y  $\mathbf{B}$ :

$$C(\lambda) = (R, G, B) = R \mathbf{R} + G \mathbf{G} + B \mathbf{B} \quad (12.7)$$

donde los parámetros  $R$ ,  $G$  y  $B$  toman valores en el rango que va de 0 a 1.0. Por ejemplo, el vértice magenta se obtiene sumando los valores máximos de rojo y azul para producir la tripla (1, 0, 1), mientras que el blan-



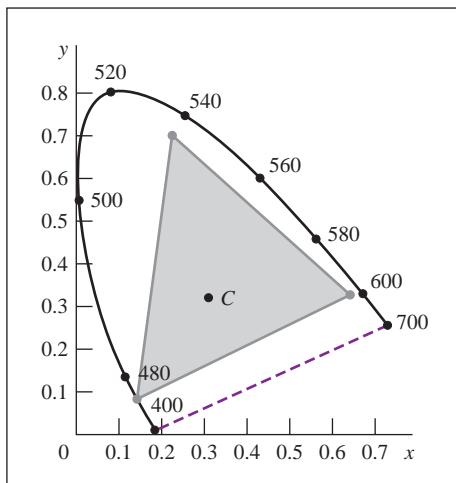
**FIGURA 12.11.** El modelo de color RGB. Cualquier color dentro del cubo unitario puede describirse mediante la combinación aditiva de los tres colores primarios.

co en  $(1, 1, 1)$  es la suma de los valores máximos de rojo, verde y azul. Las sombras de gris están representadas a lo largo de la diagonal principal del cubo, desde el origen (negro) hasta el vértice blanco. Los puntos situados en esta diagonal tienen una contribución igual de cada uno de los colores primarios, y una sombra de gris a medio camino entre el negro y el blanco se representa como  $(0.5, 0.5, 0.5)$ . Las graduaciones de color en los planos frontales y superior del cubo RGB se ilustran en la Figura 12.12.

Las coordenadas cromáticas para los fósforos RGB del estándar NTSC (National Television System Committee) se indican en la Tabla 12.1. También se indican las coordenadas cromáticas RGB dentro del modelo de color CIE y los valores aproximados utilizados para los fósforos en los monitores en color. La Figura 12.13 muestra la gama aproximada de colores para los primarios RGB del estándar NTSC.

**TABLA 12.1.** COORDENADAS CROMÁTICAS RGB ( $x, y$ ).

	Estándar NTSC	Modelo CIE	Valores aproximados monitor color
R	(0.670, 0.330)	(0.735, 0.265)	(0.628, 0.346)
G	(0.210, 0.710)	(0.274, 0.717)	(0.268, 0.588)
B	(0.140, 0.080)	(0.167, 0.009)	(0.150, 0.070)



**FIGURA 12.13.** La gama de colores RGB para las coordenadas cromáticas NTSC. El iluminante C se encuentra en la posición (0.310, 0.316), con un valor de luminancia de  $Y = 100.0$ .

## 12.5 EL MODELO DE COLOR YIQ Y LOS MODELOS RELACIONADOS

Aunque un monitor gráfico RGB requiere señales separadas para las componentes roja, verde y azul de las imágenes, los monitores de televisión utilizan una señal compuesta. La codificación de color NTSC para formar la señal de vídeo compuesta se denomina modelo YIQ.

### Los parámetros YIQ

En el modelo de color YIQ, el parámetro  $Y$  es igual que la componente  $Y$  del espacio de color  $XYZ$  de CIE. La información de luminancia (brillo) está contenida en el parámetro  $Y$ , mientras que la información cromática (tono y pureza) está incorporada en los parámetros  $I$  y  $Q$ . Para el parámetro  $Y$  se elige una combinación de rojo, verde y azul para obtener la curva de luminosidad estándar. Puesto que  $Y$  contiene la información de luminancia, los monitores de televisión en blanco y negro sólo utilizan la señal  $Y$ . El parámetro  $I$  contiene la información de color naranja-cián que proporciona el tono de sombreado de la piel, mientras que el parámetro  $Q$  comunica la información de color verde-magenta.

La señal de color compuesta NTSC está diseñada para proporcionar la información de tal manera que pueda ser recibida por los monitores de televisión en blanco y negro, que obtienen la información de escala de grises para una imagen dentro de un ancho de banda de 6 MHz. Así, la información YIQ está también codificada dentro de un ancho de banda de 6 MHz, pero los valores de luminancia y cromáticos están codificados en señales analógicas separadas. De esta forma, la señal de luminancia no tiene porqué cambiar con respecto a la de los monitores en blanco y negro, añadiéndose simplemente la información de color dentro del mismo ancho de banda. La información de luminancia, el valor  $Y$ , se comunica como una modulación de amplitud sobre una señal portadora con un ancho de banda de 4.2 MHz. La información cromática, los valores  $I$  y  $Q$ , está combinada sobre una segunda señal portadora que tiene un ancho de banda de unos 1.8 MHz. Los nombres de los parámetros  $I$  y  $Q$  hacen referencia a los métodos de modulación utilizados para codificar la información de color sobre esta portadora. Una codificación de modulación de amplitud (la señal «en fase») transmite el valor  $I$ , utilizando unos 1.3 MHz del ancho de banda, mientras que una codificación por modulación de fase (la señal «en cuadratura»), que utilizan unos 0.5 MHz, transporta el valor  $Q$ .

Los valores de luminancia están codificados con mayor precisión en la señal NTSC (ancho de banda de 4.2 MHz) que los valores cromáticos (ancho de banda 1.8 MHz), porque los humanos podemos detectar más fácilmente los pequeños cambios de brillo que los pequeños cambios de color. Sin embargo, esta menor precisión de la codificación cromática hace que las imágenes NTSC presenten una cierta degradación de la calidad el color.

Podemos calcular el valor de luminancia para un color RGB utilizando la Ecuación 10.27, y un método para producir los valores cromáticos consiste en restar la luminancia de las componentes roja y azul del color. Así,

$$\begin{aligned} Y &= 0.299 R + 0.587 G + 0.114 B \\ I &= R - Y \\ Q &= B - Y \end{aligned} \quad (12.8)$$

### Transformaciones entre los espacios de color RGB e YIQ

Un color RGB puede convertirse a un conjunto de valores YIQ utilizando un codificador NTSC que implemente los cálculos de la Ecuación 12.8 y que module las señales portadoras. La conversión del espacio RGB al espacio YIQ se lleva a cabo utilizando la siguiente matriz de transformación:

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.701 & -0.587 & -0.114 \\ -0.299 & -0.587 & 0.886 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix} \quad (12.9)$$

A la inversa, una señal de vídeo NTSC puede convertirse a valores de color RGB utilizando un decodificador NTSC, que primero separa la señal de vídeo en las componentes YIQ y luego convierte los valores YIQ a valores RGB. La conversión desde el espacio YIQ al espacio RGB se lleva a cabo mediante la transformación inversa 12.9:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1.000 & 1.000 & 0.000 \\ 1.000 & -0.509 & -0.194 \\ 1.000 & 0.000 & 1.000 \end{bmatrix} \cdot \begin{bmatrix} Y \\ I \\ Q \end{bmatrix} \quad (12.10)$$

### Los sistemas YUV e $YC_rC_b$

Debido al menor ancho de banda asignada a la información cromática en la señal de vídeo analógico compuesta NTSC, la calidad del color de una imagen NTSC no es la que debiera. Por tanto, se han desarrollado variaciones de la codificación YIQ para mejorar la calidad del color en las transmisiones de vídeo. Una de tales codificaciones es el conjunto YUV de parámetros de color, que proporciona la información de color compuesto para transmisiones de vídeo en sistemas tales como PAL (Phase Alternation Line) Broadcasting, que se utiliza en la mayor parte de Europa, así como en África, Australia y Eurasia. Otra variación de YIQ es la codificación digital denominada  $YC_rC_b$ . Esta representación del color se utiliza para manipulación de vídeo digital, y está incorporada en diversos formatos de archivo gráfico, como por ejemplo el sistema JPEG (Sección 15.4).

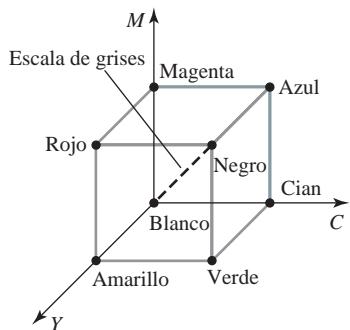
## 12.6 LOS MODELOS DE COLOR CMY Y CMYK

---

Un monitor de vídeo muestra los patrones de color combinando la luz emitida por los fósforos de pantalla, lo cual es un proceso aditivo. Sin embargo, los dispositivos de obtención de copias impresas, como las impresoras y trazadoras gráficas, generan una imagen en color recubriendo el papel con pigmentos coloreados. Nosotros vemos los patrones de color del papel mediante la luz reflejada, lo cual es un proceso substractivo.

### Los parámetros CMY

Podemos formar un modelo de color substractivo utilizando los tres colores primarios cian, magenta y amarillo. Como hemos indicado, el cian puede describirse como una combinación de verde y azul; por tanto, cuan-



**FIGURA 12.14.** El modelo de color CMY. Las posiciones dentro del cubo se describen restando del blanco las cantidades especificadas de los colores primarios.

do se refleja luz blanca en una tinta de color cian, la luz reflejada sólo contiene las componentes verde y azul, y la componente roja es absorbida, o restada, por la tinta. De forma similar, la tinta magenta resta la componente verde de la luz incidente y la tinta amarilla resta la componente azul. En la Figura 12.14 se ilustra un cubo unitario que sirve para representar el modelo CMY.

En el modelo CMY, la posición espacial (1, 1, 1) representa el negro, porque se restan todos los componentes de la luz incidente. El origen representa la luz blanca. Si se utilizan cantidades iguales de cada uno de los colores primarios, se obtienen las sombras de gris, situadas a lo largo de la diagonal principal del cubo. Una combinación de cian y magenta produce luz azul, porque las componentes roja y verde de la luz incidente se absorven. De forma similar, una combinación de tinta cian y amarilla produce luz verde, mientras que una combinación de tinta magenta y amarilla nos da la luz roja.

El proceso de impresión CMY utiliza a menudo una colección de cuatro puntos de tinta, que están dispuestos en un determinado patrón, de modo similar a cómo se utilizan en un monitor RGB los tres puntos de fósforo. Así, en la práctica, el modelo de color CMY se denomina modelo CMYK, donde  $K$  es el parámetro de color negro. Se utiliza un punto de tinta para cada uno de los colores primarios (cian, magenta y amarillo) y otro punto de tinta es negro. Se incluye un punto negro porque la luz reflejada a partir de las tintas cian, magenta y amarilla sólo produce normalmente sombras de gris. Algunos trazadores gráficos producen diferentes combinaciones de color indexando las tintas de unos colores primarios sobre otros y permitiéndolas mezclarse antes de que se sequen. Para impresión en blanco y negro o en escala de grises, sólo se utiliza la tinta negra.

### Transformaciones entre los espacios de color CMY y RGB

Podemos expresar la conversión de una representación RGB a una representación CMY utilizando la siguiente matriz de transformación:

$$\begin{bmatrix} C \\ M \\ Y \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} R \\ G \\ B \end{bmatrix} \quad (12.11)$$

donde el punto blanco del espacio RGB está representado mediante el vector columna unitario. Asimismo, podemos convertir de una representación en color CMY a una representación en color RGB utilizando la matriz de transformación:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} C \\ M \\ Y \end{bmatrix} \quad (12.12)$$

En esta transformación, el vector columna unitario representa el punto negro del espacio de color CMY.

Para la conversión de RGB al espacio de color CMYK, primero hacemos  $K = \max(R, G, B)$  y luego restamos  $K$  de cada uno de los valores  $C, M$  e  $Y$  en la Ecuación 12.11. De forma similar, para la transformación de CMYK a RGB, primero hacemos  $K = \min(R, G, B)$  y luego restamos  $K$  de cada uno de los valores  $R, G$  y  $B$  de la Ecuación 12.12. En la práctica, estas ecuaciones de transformación se suelen modificar con el fin de mejorar la calidad de impresión de cada sistema concreto.

## 12.7 EL MODELO DE COLOR HSV

Las interfaces para selección de colores por parte de los usuarios emplean a menudo un modelo de color basado en conceptos intuitivos, en lugar de en un conjunto de colores primarios. Podemos especificar un color en un modelo intuitivo seleccionando un color espectral y las cantidades de blanco y de negro que hay que añadir a ese color para obtener diferentes sombras, tintas y tonos (Sección 12.2).

### Los parámetros HSV

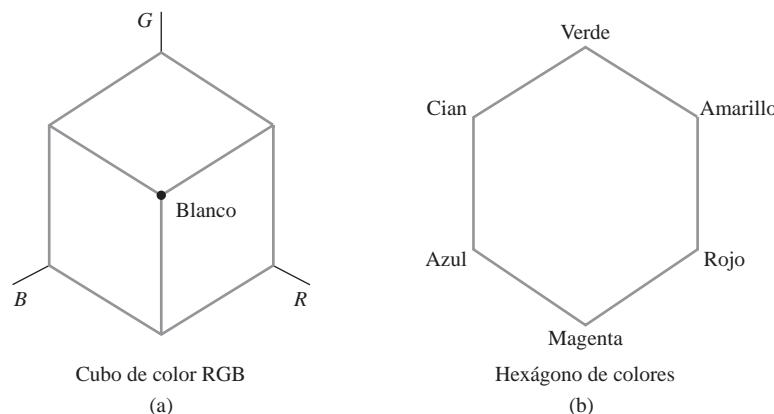
Los parámetros de color en este modelo se denominan *tono (H)*, *saturación (S)* y *valor (V)*. Podemos definir este espacio de color tridimensional relacionando los parámetros HSV con las direcciones del cubo RGB. Si pensamos en la visualización del cubo según la diagonal que va desde el vértice blanco hasta el origen (negro), podremos ver el contorno del cubo con forma hexagonal que se muestra en la Figura 12.15. El contorno del hexágono representa los distintos tonos y se utiliza como parte superior del cono hexagonal HSV (Figura 12.16). En el espacio HSV, la saturación  $S$  se mide según un eje horizontal y el parámetro de valor  $V$  se mide según un eje vertical que pasa por el centro del cono hexagonal.

El tono está representado como un ángulo con respecto al eje vertical, yendo de  $0^\circ$  para el rojo a  $360^\circ$ . Los vértices del hexágono están separados por intervalos de  $60^\circ$ . El amarillo se encuentra en  $60^\circ$ , el verde en  $120^\circ$  y el cian (opuesto al punto rojo) se encuentra en  $H = 180^\circ$ . Los colores complementarios están separados por  $180^\circ$ .

El parámetro de saturación  $S$  se utiliza para especificar la pureza de un color. Un color puro (color espectral) tiene el valor  $S = 1.0$  y los valores de  $S$  decrecientes tienden hacia la línea de escala de grises ( $S = 0$ ), situada en el centro del cono hexagonal.

El valor  $V$  varía entre 0 en el vértice del cono hexagonal y 1.0 en el plano superior. El vértice del cono hexagonal es el punto negro. En el plano superior, los colores tienen su máxima intensidad. Cuando  $V = 1.0$  y  $S = 1.0$ , tendremos los tonos puros. Los valores de los parámetros para el punto blanco son  $V = 1.0$  y  $S = 0$ .

Para la mayoría de los usuarios, este es el modelo más cómodo para la selección de colores. Comenzando con una selección de un tono puro, que especifica el ángulo de tono  $H$  y hace  $V = S = 1.0$ , describimos el



**FIGURA 12.15.** Cuando vemos (a) el cubo de color RGB según la diagonal que va del blanco al negro, el contorno del cubo de color tiene forma hexagonal (b).

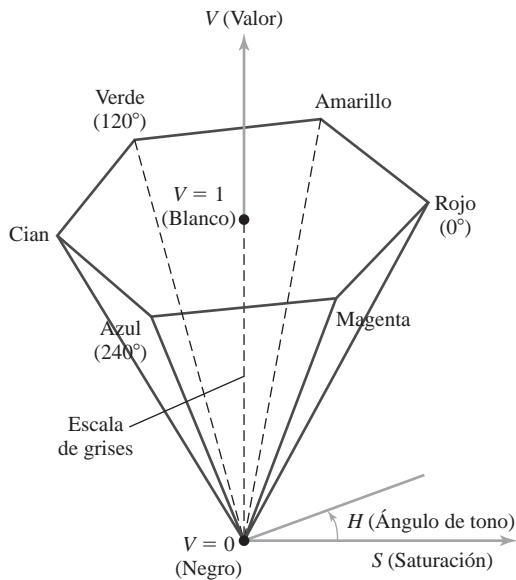


FIGURA 12.16. El cono hexagonal HSV.

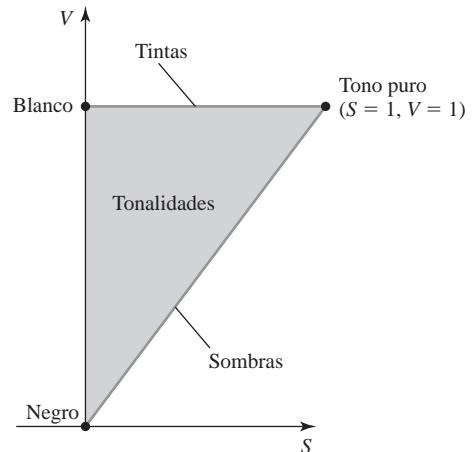


FIGURA 12.17. Sección transversal del cono hexagonal HSV, que muestra las regiones para las sombras, tintas y tonalidades.

color que queremos añadiendo blanco o negro al tono puro. La adición de negro hace que disminuya el valor de  $V$  mientras que  $S$  se mantiene constante. Por ejemplo, para obtener un azul oscuro, le podríamos asignar a  $V$  el valor 0.4 con  $S = 1.0$  y  $H = 240^\circ$ . De modo similar, cuando se añade blanco al tono seleccionado, el parámetro  $S$  decrece mientras que  $V$  se mantiene constante. Un azul claro podría especificarse con  $S = 0.3$ ,  $V = 1.0$  y  $H = 240^\circ$ . Añadiendo algo de negro y algo de blanco, hacemos que disminuyan tanto  $V$  como  $S$ . Las interfaces que utilizan este modelo suelen presentar las opciones de selección de los parámetros HSV mediante una paleta de color que contiene deslizadores y una rueda de color.

### Selección de sombras, tintas y tonalidades

Las regiones de color para la selección de sombras, tintas y tonos están representadas en el plano de sección transversal del cono hexagonal HSV que se muestra en la Figura 12.17. Añadiendo negro a un color espectral, el valor de  $V$  decrece a lo largo del lado del cono hexagonal, desplazándonos hacia el punto negro. Así, las distintas sombras estarán representadas por los valores  $S=1.0$  y  $0.0 \leq V \leq 1.0$ . La adición de blanco a los colores espectrales produce las distintas tintas situadas en el plano superior del cono hexagonal, donde los valores de los parámetros son  $V = 1.0$  y  $0 \leq S \leq 1.0$ . Las diversas tonalidades se obtienen añadiendo tanto blanco como negro a los colores espectrales, lo que genera los puntos de color situados dentro del área triangular de sección transversal del cono hexagonal.

El ojo humano puede distinguir aproximadamente unos 128 tonos distintos y unas 130 tintas (niveles de saturación) distintas. Para cada uno de estos, podemos detectar diversas sombras (valores), dependiendo del tono seleccionado. Con los colores amarillos, podemos distinguir unas 23 sombras, mientras que en el extremo azul del espectro sólo podemos distinguir 16. Esto significa que el número de colores distintos que podemos distinguir es de aproximadamente  $128 \times 130 \times 23 = 382,720$ . Para la mayoría de las aplicaciones gráficas, 128 tonos, 8 niveles de saturación y 16 valores suelen ser suficientes. Con este rango de parámetros en el modelo de color HSV, habrá disponible 16,384 colores para el usuario. Estos valores de color pueden almacenarse con 14 bits por píxel, o bien pueden utilizarse tablas indexadas de colores y un número de bits por píxel menor.

## Transformaciones entre los espacios de color HSV y RGB

Para determinar las operaciones requeridas para las transformaciones entre los espacios HSV y RGB, vamos a ver primero cómo puede construirse el cono hexagonal HSV a partir del cubo RGB. La diagonal del cubo RGB que va de negro (el origen) a blanco se corresponden con el eje  $V$  del cono hexagonal. Asimismo, cada subcubo del cubo RGB se corresponde con un área hexagonal de sección transversal del cono. En cada sección transversal, todos los lados del hexágono y todas las líneas radiales que van del eje  $V$  hasta cualquier vértice tienen el valor  $V$ . Así, para cualquier conjunto de valores RGB,  $V$  será igual al valor de la componente RGB máxima. El punto HSV correspondiente a este conjunto de valores RGB estará en la sección transversal hexagonal correspondiente al valor  $V$ . El parámetro  $S$  puede entonces determinarse como la distancia relativa de este punto con respecto al eje  $V$ . El parámetro  $H$  se determinará calculando la posición relativa del punto dentro de cada sextante del hexágono. En el siguiente procedimiento se proporciona un algoritmo para mapear cualquier conjunto de valores RGB sobre los correspondientes valores HSV:

Podemos obtener la transformación desde el espacio HSV al espacio RGB realizando las operaciones inversas a las que se muestran en el procedimiento anterior. Estas operaciones inversas se llevan a cabo para cada sextante del cono hexagonal y las ecuaciones de transformación resultantes están resumidas en el siguiente algoritmo:

```

class rgbSpace {public: float r, g, b;};
class hsvSpace {public: float h, s, v;};

const float noHue = -1.0;
inline float min(float a, float b) {return (a < b)? a : b;}
inline float max(float a, float b) {return (a > b)? a : b;}

void rgbTOhsv (rgbSpace& rgb, hsvSpace& hsv)
{
    /* Los valores RGB y HSV están en el rango 0 a 1.0 */
    float minRGB = min (r, min (g, b)), maxRGB = max (r, max (g, b));
    float deltaRGB = maxRGB - minRGB;

    v = maxRGB;
    if (maxRGB != 0.0)
        s = deltaRGB / maxRGB;
    else
        s = 0.0;
    if (s <= 0.0)
        h = noHue;
    else {
        if (r == maxRGB)
            h = (g - b) / deltaRGB;
        else
            if (g == maxRGB)
                h = 2.0 + (b - r) / deltaRGB;
            else
                if (b == maxRGB)
                    h = 4.0 + (r - g) / deltaRGB;
        h *= 60.0;
        if (h < 0.0)

```

```

    h += 360.0;
    h /= 360.0;
}
}

```

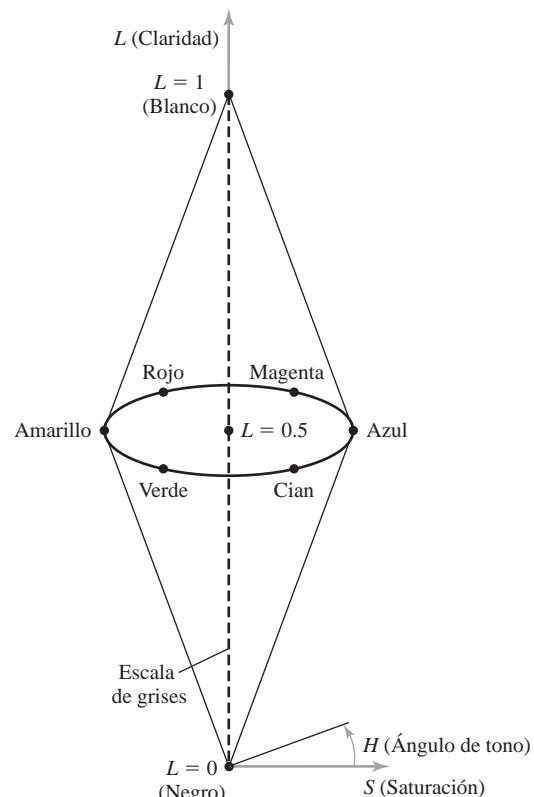
## 12.8 EL MODELO DE COLOR HLS

Otro modelo basado en parámetros intuitivos de color es el sistema HLS utilizado por Tektronix Corporation. Este espacio de color tiene la representación de doble cono que se muestra en la Figura 12.18. Los tres parámetros de este modelo de color se denominan tono ( $H$ ), claridad ( $L$ ) y saturación ( $S$ ).

El tono tiene el mismo significado que en el modelo HSV. Especificando un ángulo con respecto al eje vertical que indica un tono (colorpectral). En este modelo,  $H = 0^\circ$  se corresponde con el azul. Los colores restantes se especifican alrededor del perímetro de cono en el mismo orden que en el modelo HSV. El magenta estará a  $60^\circ$ , el rojo a  $120^\circ$  y el cian a  $H = 180^\circ$ . De nuevo, los colores complementarios estarán separados  $180^\circ$  en este doble cono.

El eje vertical en este modelo se denomina claridad,  $L$ . Para  $L = 0$ , tendremos el negro, mientras que el blanco se encontrará en  $L = 1.0$ . Los valores de escala de grises se encuentran a lo largo del eje  $L$  y los colores puros están en el plano  $L = 0.5$ .

El parámetro de saturación  $S$  especifica de nuevo la pureza de un color. Este parámetro varía entre 0 y 1.0 y los colores puros son aquellos para los que  $S = 1.0$  y  $L = 0.5$ . A medida que  $S$  disminuye, se añade más blanco a un color. La línea de escala de grises se encuentra en  $S = 0$ .



**FIGURA 12.18.** El doble cono HLS.

Para especificar un color, comenzamos seleccionando el ángulo de tono  $H$ . Entonces, podemos obtener una sombra, tinta o tonalidad correspondientes a dicho tono ajustando los parámetros  $L$  y  $S$ . Se obtiene un color más claro incrementando  $L$ , mientras que si se disminuye  $L$  el color que se obtiene es más oscuro. Cuando se reduce  $S$ , el punto de color espacial se mueve hacia la línea de escala de grises.

## 12.9 SELECCIÓN Y APLICACIONES DEL COLOR

---

Los paquetes gráficos pueden proporcionar capacidades de color que nos ayuden a seleccionar los colores necesarios. Por ejemplo, una interfaz puede contener deslizadores y ruedas de color en lugar de exigirnos que todas las especificaciones de color se proporcionen como valores numéricos que definen las componentes RGB. Además, pueden proporcionarse facilidades para seleccionar combinaciones de color armoniosas, así como directrices para la selección básica de colores.

Un método para obtener un conjunto de colores coordinados consiste en generar las combinaciones de color a partir de un pequeño subespacio del modelo de color. Si los colores se seleccionan a intervalos regulares situados a lo largo de cualquier línea recta dentro del cubo RGB o CMY, por ejemplo, cabe esperar que obtengamos un conjunto de colores bien adaptados. Los tonos aleatoriamente seleccionados producen normalmente combinaciones de color duras y muy chocantes. Otra consideración que hay que tener en cuenta en las imágenes de color es el hecho de que percibimos los colores con profundidades diferentes. Esto sucede porque los ojos enfocan los colores de acuerdo con su frecuencia. Por ejemplo, los azules tienden a parecer más lejanos de lo que están en realidad. Si se muestra un patrón azul al lado de uno rojo los ojos tenderán a fatigarse, ya que necesitaremos continuamente ajustar el enfoque cuando desplazemos nuestra atención de un área a otra. Este problema puede reducirse separando estos colores o utilizando colores extraídos de una mitad o menos del hexágono de colores del modelo HSV. Con esta técnica, la imagen contendrá azules y verdes o rojos y amarillos, por ejemplo.

Como regla general, la utilización de un número menor de colores produce imágenes con mejor aspecto que si se utilizara un gran número de colores. Asimismo, las tintas y las sombras tienden a mezclarse mejor que los tonos puros. Para un fondo, el gris o el complementario de uno de los colores de primer plano suelen ser las mejores elecciones.

## 12.10 RESUMEN

---

La luz puede describirse como una radiación electromagnética con una cierta distribución de energía y que se propaga a través del espacio, y los componentes de color de la luz se corresponden con las frecuencias situadas dentro de una estrecha banda del espectro electromagnético. Sin embargo, la luz exhibe otras propiedades, y podemos caracterizar los diferentes aspectos de la luz utilizando diversos parámetros. Con las teorías de la luz basadas en la dualidad onda-corpúsculo, podemos explicar las características físicas de la radiación visible, mientras que para cuantificar nuestras percepciones sobre una fuente luminosa utilizamos términos tales como la frecuencia dominante (tono), la luminancia (brillo) y la pureza (saturación). El tono y la pureza se suelen denominar, conjuntamente, propiedades cromáticas de un color.

También utilizamos los modelos de color para explicar los efectos de la combinación de fuentes luminosas. Un método para definir un modelo de color consiste en especificar un conjunto de dos o más colores primarios que se combinan para generar otros colores. Sin embargo, no hay ningún conjunto finito de colores primarios capaz de producir todos los colores o de describir todas las características de color. El conjunto de colores que puede generarse a partir del conjunto de primarios se denomina gama de colores. Dos colores que puedan combinarse para producir luz blanca se denominan colores complementarios.

En 1931, la CIE (Comisión Internacional de Iluminación) adoptó como estándar un conjunto de tres funciones hipotéticas de ajuste de color. Este conjunto de colores se denomina modelo XYZ, donde  $X$ ,  $Y$  y  $Z$

representan las cantidades de cada color necesarias para generar cualquier color del espectro electromagnético. Las funciones de ajuste de color están estructuradas de modo que todas las funciones sean positivas y que el valor de  $Y$  correspondiente a cada color represente la luminancia. Los valores  $X$  e  $Y$  normalizados, denominados  $x$  e  $y$ , se utilizan para situar las posiciones de todos los colores espectrales en el diagrama cromático CIE. Podemos utilizar el diagrama cromático para comparar gamas de colores correspondientes a diferentes modelos de color, para identificar colores complementarios y para determinar la frecuencia dominante y pureza de un color especificado.

Otros modelos de color basados en un conjunto de tres primarios son los modelos RGB, YIQ y CMY. Utilizamos el modelo RGB para describir los colores que se muestran en los monitores de vídeo. El modelo YIQ se utiliza para describir la señal de vídeo compuesta utilizada en las emisiones de televisión. Por su parte, el modelo CMY se emplea para describir el color en los dispositivos de obtención de copias impresas.

Las interfaces de usuario proporcionan a menudo modelos de color intuitivos, como los modelos HSV y HLS, para las selecciones de valores de color. Con estos modelos, especificamos un color como una mezcla de un tono seleccionado y ciertas cantidades de blanco y de negro. La adición de negro produce las distintas sombras de color, la adición de blanco produce las tintas y la adicción tanto de negro como de blanco produce las tonalidades.

La selección del color es un factor importante en el diseño de imágenes efectivas. Para evitar las combinaciones de color chilonas, podemos seleccionar colores adyacentes en una imagen que no difiera grandemente en cuanto a su frecuencia dominante. Asimismo, podemos seleccionar las combinaciones de color extrayéndolas de un pequeño subespacio de un cierto modelo de color. Como regla general, un pequeño número de combinaciones de color formadas por tintas y sombras, en lugar de por tonos puros, da como resultado imágenes en color más armoniosas.

## REFERENCIAS

---

Puede encontrar un análisis detallado de la ciencia del color en Wyszecki y Stiles (1982). Los modelos de color y las técnicas de visualización de colores se tratan en Smith (1978), Heckbert (1982), Durrett (1987), Schwartz, Cowan y Beatty (1987), Hall (1989) y Travis (1991). Puede encontrar algoritmos para diversas aplicaciones del color en Glassner (1990), Arvo (1991), Kirk (1992), Heckbert (1994) y Paeth (1995). Para obtener información adicional sobre el sistema visual humano y nuestra percepción de la luz y el color, consulte Glassner (1995).

## EJERCICIOS

---

- 12.1 Calcule las expresiones para convertir los parámetros de color RGB en valores HSV.
- 12.2 Calcule las expresiones para convertir valores de color HSV en valores RGB.
- 12.3 Diseñe un procedimiento interactivo que permita seleccionar parámetros de color HSV a partir de un menú visualizado; entonces, convierta los valores HSV en valores RGB para poder almacenarlos en un búfer de imagen.
- 12.4 Escriba un programa para seleccionar colores utilizando un conjunto de tres deslizadores con los que seleccionar los valores de los parámetros de color HSV.
- 12.5 Modifique el programa del ejercicio anterior para mostrar los valores numéricos de las componentes RGB de cada color seleccionado.
- 12.6 Modifique el programa del ejercicio anterior para mostrar las componentes de color RGB y el color combinado en pequeñas ventanas de visualización.
- 12.7 Calcule las expresiones para convertir los valores de color RGB en parámetros de color HLS.
- 12.8 Calcule las expresiones para convertir valores de color HLS en valores RGB.

- 12.9 Escriba un programa que produzca un conjunto de colores linealmente interpolados a partir de dos posiciones especificadas en el espacio RGB.
- 12.10 Escriba una rutina interactiva para seleccionar valores de color dentro de un subespacio especificado del espacio RGB.
- 12.11 Escriba un programa que produzca un conjunto de colores linealmente interpolados a partir de dos posiciones especificadas dentro del espacio HSV.
- 12.12 Escriba un programa que genere un conjunto de colores linealmente interpolados entre de dos posiciones especificadas del espacio HLS.
- 12.13 Escriba un programa para mostrar dos rectángulos de color RGB adyacentes. Rellene un rectángulo con un conjunto de puntos de color RGB aleatoriamente seleccionados y rellene el otro rectángulo con un conjunto de puntos de color seleccionados de entre un pequeño subespacio RGB. Experimente con diferentes selecciones aleatorias y diferentes subespacios para comparar los dos patrones de color.
- 12.14 Muestre los dos rectángulos de color del ejercicio anterior utilizando selecciones de color del espacio HSV o del espacio HLS.

# Animación por computadora



Una imagen cuyas características faciales han sido transformadas mediante las técnicas de morfismo.  
(Cortesía de Vertigo Technology, Inc.)

- |  |  |
|--|--|
| <ul style="list-style-type: none"><li>13.1 Métodos de barrido para las animaciones por computadora</li><li>13.2 Diseño de secuencias de animación</li><li>13.3 Técnicas tradicionales de animación</li><li>13.4 Funciones generales de animación por computadora</li><li>13.5 Lenguajes de animación por computadora</li></ul> | <ul style="list-style-type: none"><li>13.6 Sistemas de fotogramas clave</li><li>13.7 Especificaciones de movimientos</li><li>13.8 Animación de figuras articuladas</li><li>13.9 Movimientos periódicos</li><li>13.10 Procedimientos de animación en OpenGL</li><li>13.11 Resumen</li></ul> |
|--|--|

Hoy en día, se utilizan los métodos infográficos de forma común para generar animaciones para muy diversas aplicaciones, incluyendo el entretenimiento (películas y dibujos animados), la industria publicitaria, los estudios científicos y de ingeniería, la formación y la educación. Aunque todos tendemos a pensar en la animación como si implicara el movimiento de objetos, el término **animación por computadora** se refiere, en general, a cualquier secuencia temporal donde se aprecien cambios visuales en una imagen. Además de cambiar las posiciones de los objetos mediante traslaciones o rotaciones, una animación generada por computadora puede mostrar variaciones temporales que afecten al tamaño de los objetos, a su color, a su transparencia o a las texturas superficiales. Las animaciones utilizadas en la industria publicitaria recurren frecuentemente a las transiciones entre una forma de objeto y otra: por ejemplo, transformar una lata de aceite para motores en un motor de automóvil. También podemos generar animaciones por computadora variando parámetros de la cámara, como la posición, la orientación o la distancia focal. Y las variaciones en los efectos de iluminación u otros parámetros y procedimientos asociados con la iluminación y la representación de escenas pueden usarse también para producir animaciones por computadora.

Una consideración importante en las animaciones generadas por computadora es la cuestión del realismo. Muchas aplicaciones requieren imágenes suficientemente realistas. Por ejemplo, una representación precisa de la forma de una tormenta o de otro fenómeno natural escrito con un modelo dinámico tiene una gran importancia para evaluar la fiabilidad del modelo. De forma similar, los simuladores para el entrenamiento de los pilotos de aeronaves y de los operadores de equipos pesados debe producir representaciones razonablemente precisas del entorno. Las aplicaciones de entretenimiento y publicitarias, por el contrario, suelen estar más interesadas en los efectos visuales. De este modo, puede que las escenas se muestren con formas exageradas y movimientos y transformaciones no realistas. Sin embargo, hay muchas aplicaciones de entretenimiento y publicitarias que sí que requieren una representación precisa en las escenas generadas por computadora. En algunos estudios científicos y de ingeniería, el realismo no constituye un objetivo; por ejemplo, determinadas magnitudes físicas suelen mostrarse con pseudo-colores o con formas abstractas que cambian con el tiempo con el fin de ayudar al investigador a comprender la naturaleza del proceso físico.

Dos métodos básicos para la construcción de una secuencia animada son la **animación en tiempo real** y la **animación imagen a imagen**. En una animación por computadora en tiempo real, cada etapa de la secuencia se visualiza a medida que se la genera. Por ello, la animación debe generarse a una frecuencia que sea compatible con las restricciones de la frecuencia de refresco. Para una animación imagen a imagen, se genera de forma separada cada imagen de la secuencia y se la almacena. Posteriormente, las imágenes pueden grabarse sobre una película o mostrarse de forma consecutiva en un monitor de vídeo, en el modo de «reproducción en tiempo real». Las secuencias animadas simples se suelen producir en tiempo real, mientras que las animaciones más complejas se construyen más lentamente, imagen a imagen. Pero algunas aplicaciones requieren animación en tiempo real, independientemente de la complejidad de la animación. Una animación para un

simulador de vuelo, se tiene que generar en tiempo real, porque las imágenes de vídeo deben construirse respondiendo de manera inmediata a los cambios de las configuraciones de control. En tales casos, se suelen desarrollar sistemas hardware y software especializados con el fin de poder generar rápidamente las complejas secuencias de animación.

## 13.1 MÉTODOS DE BARRIDO PARA LAS ANIMACIONES POR COMPUTADORA

---

La mayoría de las veces, podemos crear secuencias simples de animación en nuestros programas utilizando métodos de tiempo real, pero en general, podemos producir una secuencia animada en un sistema de visualización por barrido generando una imagen cada vez y guardando dicha imagen completa en un archivo para su visualización posterior. La animación puede verse entonces recorriendo la secuencia completa de imágenes, o bien pueden transferirse esas imágenes a una película. Sin embargo, si queremos generar una imagen en tiempo real, debemos producir las imágenes de la secuencia con la suficiente rapidez como para que se perciba un movimiento continuo. Para una escena compleja, la generación de cada imagen de la animación puede ocupar la mayor parte del ciclo de refresco. En tal caso, los objetos que se generen primero se visualizarán durante la mayor parte de ese ciclo de refresco, pero los objetos generados hacia el final del mismo desaparecerán muy poco después de mostrarlos. Asimismo, en las animaciones muy complejas, el tiempo de generación de la imagen podría ser superior al tiempo necesario para refrescar la pantalla, lo que puede hacer que se perciban movimientos erráticos y que se muestren imágenes fracturadas. Puesto que las imágenes de la pantalla se generan a partir de los valores de píxel sucesivamente modificados que hay almacenados en el búfer de refresco, podemos aprovechar algunas de las características del proceso de refresco de pantalla en los sistemas de barrido con el fin de generar rápidamente las secuencias de animación.

### Doble búfer

Un método para producir una animación en tiempo real con un sistema de barrido consiste en emplear dos búferes de refresco. Inicialmente, creamos una imagen para la animación en uno de los búferes. Después, mientras se refresca la pantalla a partir del contenido de dicho búfer, construimos la imagen siguiente de la secuencia en el otro búfer. Cuando dicha imagen se complete, cambiamos los roles de los dos búferes para que las rutinas de refresco utilicen el segundo búfer mientras se crea la siguiente imagen de la secuencia en el siguiente búfer. Este proceso alternativo de comutación de búferes continua mientras dure la secuencia. Las bibliotecas gráficas que permiten tales operaciones suelen disponer de una función para activar la rutina de doble búfer y otra función para intercambiar los dos búferes.

Cuando se realiza una llamada para comutar los dos búferes de refresco, el intercambio puede realizarse en diversos instantes. La implementación más sencilla consiste en comutar los búferes al final del ciclo de refresco actual, durante el retorno vertical del haz de electrones. Si un programa puede completar la construcción de una imagen dentro del tiempo que dura un ciclo de refresco, como por ejemplo  $\frac{1}{60}$  de segundo, la secuencia animada se mostrará de forma sincronizada con la tasa de refresco de pantalla.

Pero si el tiempo necesario para construir una imagen es mayor que el tiempo de refresco, la imagen actual se mostrará durante dos o más ciclos de refresco mientras se genera la siguiente imagen de la secuencia de animación. Por ejemplo, si la tasa de refresco de pantalla es de 60 imágenes por segundo y se tarda  $\frac{1}{50}$  de segundo en construir cada nueva imagen de la secuencia, las imágenes se mostrarán en pantalla dos veces y la velocidad de animación será únicamente de 30 imágenes por segundo. De forma similar, si el tiempo de construcción de una nueva imagen es de  $\frac{1}{25}$  de segundo, la velocidad de imagen de la animación se reduce a 20 imágenes por segundo, ya que cada una de las imágenes tendrá que ser mostrada tres veces.

Con la técnica de doble búfer pueden aparecer velocidades de animación irregulares cuando el tiempo de generación de cada imagen está muy próximo a un múltiplo entero del tiempo de refresco de pantalla. Como ejemplo de esto, si la velocidad de refresco de pantalla es de 60 imágenes por segundo, podría producirse una velocidad de animación errática si el tiempo de construcción de la imagen estuviera muy próximo a  $\frac{1}{60}$  de

segundo,  $\frac{2}{60}$  de segundo o  $\frac{3}{60}$  de segundo, etc. Debido a las pequeñas variaciones en el tiempo de ejecución de las rutinas que generan las primitivas y sus atributos, algunas imágenes podrían requerir algo más de tiempo para generarse y otras podrían requerir un tiempo más corto. Eso podría hacer que la velocidad de animación cambiara de forma abrupta y errática. Una forma de compensar este efecto consiste en añadir un pequeño retardo temporal al programa, mientras que otra posibilidad es alterar los movimientos o la descripción de la escena con el fin de acortar el tiempo de construcción de las imágenes.

### Generación de animaciones mediante operaciones de barrido

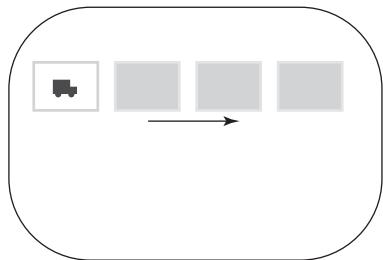
También podemos generar animaciones en tiempo real en los sistemas de barrido, para algunas aplicaciones limitadas, utilizando la transferencia en bloque de matrices rectangulares de píxeles. Esta técnica de animación se utiliza a menudo en los programas de juegos. Como hemos visto en la Sección 5.6, un método simple para mover un objeto de una ubicación a otra en el plano *xy* consiste en transferir el grupo de píxeles que definen la forma del objeto hasta una nueva ubicación. Las rotaciones bidimensionales en múltiplos de 90° también son simples de realizar, e incluso podemos rotar bloques rectangulares de píxeles otros ángulos distintos, siempre que utilicemos procedimientos de *antialiasing*. Para una rotación que no sea un múltiplo de 90°, necesitamos estimar el porcentaje de recubrimiento de área correspondiente a los píxeles que se solapan con el bloque rotado. Pueden ejecutarse secuencias de operaciones de barrido para obtener una animación en tiempo real de objetos bidimensionales o tridimensionales, siempre que restrinjamos la animación a movimientos dentro del plano de proyección. Entonces, no será necesario invocar algoritmos de proyección ni de detección de superficies visibles.

También podemos animar los objetos según trayectorias de movimiento bidimensionales utilizando **tablas de transformación de colores**. Con este método, predefinimos el objeto en posiciones sucesivas a lo largo de la trayectoria de movimiento y asignamos a los sucesivos bloques de píxeles una serie de entradas en la tabla de color. Los píxeles correspondientes a la primera posición del objeto se configuran con un color de primer plano y los píxeles de las demás posiciones del objeto se configuran con el color de fondo. La animación se consigue entonces cambiando los valores de la tabla de colores, de modo que el color del objeto en las posiciones sucesivas a lo largo del trayecto de animación se vaya transformando en el color de primer plano, a medida que la posición precedente se configura con el color de fondo (Figura 13.1).

## 13.2 DISEÑO DE SECUENCIAS DE ANIMACIÓN

La construcción de una secuencia de animación puede ser una tarea complicada, particularmente cuando requiere un guión y múltiples objetos, cada uno de los cuales puede moverse de diferente forma. Un enfoque básico consiste en diseñar tales secuencias de animación mediante las siguientes etapas de desarrollo:

- Realización del guión.
- Definición de los objetos.
- Especificación de los fotogramas clave.
- Generación de los fotogramas intermedios.



**FIGURA 13.1.** Animación en tiempo real en un sistema de barrido mediante una tabla de colores.

El **guión** es un resumen de la acción en el que se define la secuencia de movimiento como el conjunto de sucesos básicos que deben tener lugar. Dependiendo del tipo de animación que haya que producir, el guión puede estar compuesto por un conjunto de burdos dibujos y una breve descripción de los movimientos, o puede ser simplemente una lista de las ideas básicas que describen la acción. Originalmente, ese conjunto de burdos dibujos que describen el guión se solía fijar en un panel de gran tamaño que se utilizaba para presentar una vista global del proyecto de animación. De aquí proviene el nombre inglés «*storyboard*».

Para cada participante en la acción se proporciona una **definición del objeto**. Los objetos pueden definirse en términos de las formas básicas, como por ejemplo polígonos o *splines* superficiales. Además, suele proporcionarse una descripción de los movimientos que tengan que realizar cada personaje u objeto descrito en el guión.

Un **fotograma clave** es un dibujo detallado de la escena en un cierto momento de la secuencia de animación. Dentro de cada fotograma clave, cada objeto (o personaje) se posiciona de acuerdo con el tiempo correspondiente a dicho fotograma. Algunos fotogramas clave se eligen en las posiciones extremas de la acción, mientras que otros se espacian para que el intervalo de tiempo entre un fotograma clave y el siguiente no sea excesivo. Para los movimientos intrincados se especifican más fotogramas clave que para los movimientos simples o lentos. El desarrollo de los fotogramas clave suele, por regla general, ser responsabilidad de los animadores expertos, siendo normal que se asigne un animador distinto para cada personaje de la animación.

Los **fotogramas intermedios** (*in-betweens*) son los comprendidos entre los sucesivos fotogramas clave. El número total de imágenes o fotogramas, y por tanto el número total de fotogramas intermedios, necesarios para una animación vendrá determinado por el medio de visualización que se utilice. Las películas requieren 24 imágenes por segundo, mientras que los terminales gráficos se refrescan con una tasa de 60 o más imágenes por segundo. Normalmente, los intervalos temporales de la secuencia se configuran de tal modo que haya entre tres y cinco fotogramas intermedios entre cada par de fotogramas clave sucesivos. Dependiendo de la velocidad especificada para la secuencia, será necesario definir más o menos fotogramas clave. Como ejemplo, una secuencia de película de un minuto de duración contiene un total de 1440 fotogramas; si se requieren cinco fotogramas intermedios entre cada par de fotogramas clave, entonces será necesario desarrollar 288 fotogramas clave.



**FIGURA 13.2.** Una imagen del galardonado corto animado *Luxo Jr.* Esta película fue diseñada utilizando un sistema de animación basado en fotogramas clave y técnicas de dibujos animados con el fin de que las lámparas se muevan con si estuvieran vivas. Las imágenes finales fueron obtenidas con múltiples fuentes luminosas y con técnicas de texturado procedimental. (Cortesía de Pixar. © 1986 Pixar.)



**FIGURA 13.3.** Un fotograma del corto *Tin Toy*, la primera película de animación por computadora que ganó un Oscar. Diseñada mediante un sistema de animación basada en fotogramas clave, la película también requirió un detallado modelado de las expresiones faciales. Las imágenes finales fueron obtenidas utilizando sombreado procedimental, técnicas de auto-sombreado, desenfoque de movimiento y mapeado de texturas. (Cortesía de Pixar. © 1988 Pixar.)

Puede que sea necesario llevar a cabo diversas otras tareas, dependiendo de la aplicación. Estas tareas adicionales incluyen la verificación del movimiento, la edición y la producción y sincronización de una banda sonora. Muchas de las funciones necesarias para producir animaciones generales se llevan ahora a cabo con ayuda de computadoras. Las Figuras 13.2 y 13.3 muestran ejemplos de imágenes generadas por computadora para secuencias de animación.

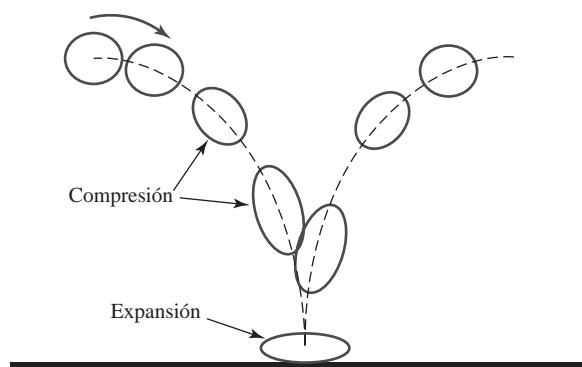
### 13.3 TÉCNICAS TRADICIONALES DE ANIMACIÓN

Los profesionales que trabajan en la producción de películas animadas utilizan diversos métodos para mostrar y enfatizar secuencias de movimientos. Estos métodos incluyen la deformación de los objetos, el espaciado entre fotogramas de la secuencia, la anticipación y seguimiento del movimiento y el enfoque de la acción.

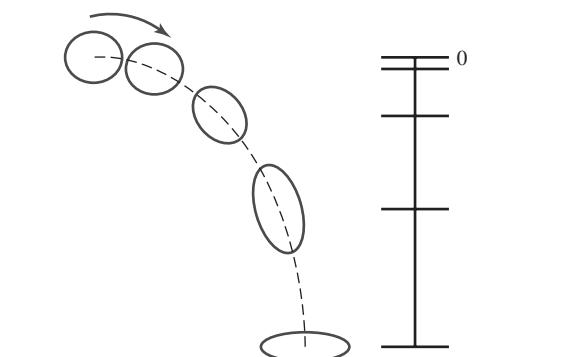
Una de las técnicas más importantes para simular efectos de aceleración, particularmente para los objetos no rígidos, es la técnica de **compresión y expansión**. La Figura 13.4 muestra cómo se utiliza esta técnica para enfatizar la aceleración y deceleración de una bola que rebota contra el suelo. A medida que la bola acelera, comienza a expandirse. Cuando la bola impacta en el suelo y se detiene, primero se comprime y luego se vuelve a expandir a medida que acelera y rebota hacia arriba.

Otra técnica utilizada por los profesionales de las películas animadas es la **temporización**, que hace referencia al espaciado entre fotogramas de la secuencia. Un objeto que se mueva más lentamente se representará mediante fotogramas con un espaciado menor, mientras que un objeto que se mueva rápidamente se mostrará distribuyendo menos fotogramas a lo largo de todo el trayecto de movimiento. Este efecto se ilustra en la Figura 13.5, donde podemos ver que los cambios de posición entre un fotograma y otro se incrementan a medida que se acelera la bola en su caída.

Los movimientos de los objetos también pueden enfatizarse creando acciones preliminares que indiquen una **anticipación** de un movimiento inminente. Por ejemplo, un personaje de dibujos animados puede inclinar-



**FIGURA 13.4.** Ejemplo de rebote de una pelota donde se ilustra la técnica de «compresión y expansión» para enfatizar la aceleración de los objetos.



**FIGURA 13.5.** Los cambios de posición entre fotogramas para el rebote de una bola se incrementan a medida que lo hace la velocidad de una bola.

se hacia adelante y girar el cuerpo antes de comenzar a correr. O bien, otro personaje puede hacer un «molinillo» con los brazos antes de arrojar una bola. De forma similar, las **acciones de seguimiento** pueden utilizarse para enfatizar un movimiento anterior. Después de arrojar una bola, un personaje puede continuar moviendo el brazo hasta volver a acercarlo al cuerpo. O bien, el sombrero de la cabeza de un personaje que se ha detenido abruptamente puede salir volando. Asimismo, las acciones pueden enfatizarse mediante las técnicas de **regulación del punto de atención**, que hacen referencia a cualquier método que permita centrarse en una parte importante de la escena, como por ejemplo aquella parte donde un personaje está ocultando algo.

## 13.4 FUNCIONES GENERALES DE ANIMACIÓN POR COMPUTADORA

---

Se han desarrollado muchos paquetes software para diseño de animaciones generales o para realizar tareas de animación especializadas. Las funciones de animación típicas incluyen la gestión del movimiento de los objetos, la generación de vistas de los objetos, la proyección de movimientos de la cámara y la generación de fotogramas intermedios. Algunos paquetes de animación, como por ejemplo Wavefront, proporcionan funciones especiales tanto para el diseño global de animación como para el procesamiento de objetos individuales. Otros son paquetes de propósito especial para características concretas de una animación, como por ejemplo los sistemas para generar fotogramas intermedios o los sistemas para animación de personajes.

En los paquetes de animación generales se suele proporcionar un conjunto de rutinas para la gestión de la base de datos de objetos. Las formas de los objetos y sus parámetros asociados se almacenan y actualizan en la base de datos. Otras funciones de manejo de los objetos incluyen las necesarias para generar los movimientos de los objetos y las empleadas para representar las superficies de los objetos. Los movimientos pueden generarse de acuerdo con restricciones especificadas utilizando transformaciones bidimensionales o tridimensionales. Entonces, pueden aplicarse funciones estándar para identificar las superficies visibles y aplicar los algoritmos de representación.

Otro conjunto típico de funciones simula los movimientos de la cámara. Los movimientos estándar de una cámara son el zoom, las panorámicas y los giros. Finalmente, dada la especificación de los fotogramas clave, pueden generarse automáticamente los fotogramas intermedios.

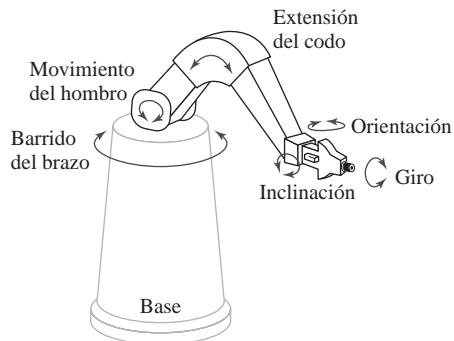
## 13.5 LENGUAJES DE ANIMACIÓN POR COMPUTADORA

---

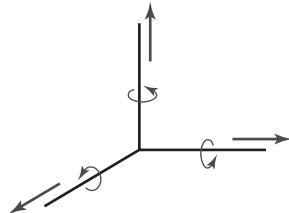
Podemos desarrollar rutinas para diseñar y controlar las secuencias de animación mediante un lenguaje de programación de propósito general, como C, C++, Lisp o Fortran, pero también se han desarrollado diversos lenguajes especializados de animación. Estos lenguajes incluyen normalmente un editor gráfico, un generador de fotogramas clave, un generador de fotogramas intermedios y una serie de rutinas gráficas estándar. El editor gráfico permite al animador diseñar y modificar formas de objetos, utilizando *splines* de superficie, métodos constructivos de geometría sólida u otros esquemas de representación.

Una tarea importante dentro de la especificación de la animación es la *descripción de la escena*. Esto incluye el posicionamiento de los objetos y las fuentes luminosas, la definición de los parámetros fotométricos (intensidades de las fuentes luminosas y propiedades de iluminación de las superficies) y la configuración de los parámetros de la cámara (posición, orientación y características del objetivo). Otra función estándar en este tipo de lenguajes es la *especificación de acciones*, que implica la disposición de las trayectorias de movimiento correspondientes a los objetos y a la cámara. Y también necesitamos las rutinas gráficas usuales: transformaciones de visualización y de perspectiva, transformaciones geométricas para generar movimientos de los objetos en función de las aceleraciones o especificaciones de proyectos cinemáticos, identificación de superficies visibles y operaciones de representación de las superficies.

Los **sistemas de fotogramas clave** fueron diseñados originalmente como un conjunto separado de las rutinas de animación para generar los fotogramas intermedios a partir de los fotogramas clave especificados por los usuarios. Ahora, estas rutinas suelen formar parte de paquetes de animación más generales. En el caso más



**FIGURA 13.6.** Grados de libertad para un robot estacionario de un único brazo.



**FIGURA 13.7.** Grados de libertad de traslación y rotación para la base del brazo robotizado.

simple, cada objeto de la escena se define como un conjunto de cuerpos rígidos conectados en las uniones y con un número limitado de grados de libertad. Por ejemplo, el robot de un único brazo de la Figura 13.6 tiene seis grados de libertad, que se denominan barrido del brazo, movimiento del hombro, extensión del codo, inclinación, orientación y giro. Podemos ampliar el número de grados de libertad de este brazo robotizado a nueve permitiendo una traslación tridimensional de la base (Figura 13.7). Si también permitimos rotaciones de la base, el brazo robotizado puede tener un total de doce grados de libertad. Por comparación, el cuerpo humano tiene más de 200 grados de libertad.

Los **sistemas parametrizados** permiten especificar las características del movimiento de los objetos como parte de la definición de esos mismos objetos. Los parámetros ajustables controlan características de los objetos tales como grados de libertad, las limitaciones del movimiento y los cambios permitidos en la forma.

Los **sistemas de script** permiten definir las especificaciones de los objetos y las secuencias de animación mediante un *script* introducido por el usuario. Mediante el *script*, puede construirse una biblioteca de objetos y movimientos diversos.

## 13.6 SISTEMAS DE FOTOGRAMAS CLAVE

Puede utilizarse un sistema de fotogramas clave para generar un conjunto de fotogramas intermedios a partir de la especificación de dos (o más) fotogramas clave. Los trayectos de movimiento pueden especificarse mediante una *descripción cinemática* como un conjunto de *splines* curvas, o bien pueden *fundamentarse físicamente* los movimientos especificando las fuerzas que actúan sobre los objetos que hay que animar.

Para las escenas complejas, podemos separar los fotogramas en componentes u objetos individuales denominados *cel*s (transparencias de celuloide). Este término fue acuñado en el mundo de las técnicas de dibujos animados, donde el fondo y cada uno de los personajes de una escena se dibujaba en una transparencia separada. Entonces, apilando las transparencias por orden, desde el fondo hasta el primer plano, se fotografiaba el conjunto para obtener el fotograma completo. Con este sistema, se utilizan los trayectos de animación especificados para obtener el siguiente *cel* de cada personaje, interpolando las posiciones a partir de los tiempos correspondientes a los fotogramas clave.

Con transformaciones complejas de los objetos, las formas de los objetos pueden cambiar a lo largo del tiempo. Como ejemplos podríamos citar las ropas, las características faciales, las ampliaciones de determinados detalles, las formas evolutivas y la explosión o desintegración de los objetos. Para las superficies descritas con mallas poligonales, estos cambios pueden dar como resultado modificaciones significativas en la forma de los polígonos, por lo que el número de vistas de un polígono podría ser distinto entre un fotograma y el siguiente. Estos cambios se incorporan en el desarrollo de los fotogramas intermedios añadiendo o eliminando aristas a los polígonos de acuerdo con los requisitos impuestos por los fotogramas clave correspondientes.

## Morfismo

La modificación de la forma de un objeto, para que éste adopte una forma distinta, se denomina **morfismo**, palabra que proviene de «metamorfosis». Un animador puede modelar un morfismo haciendo que las formas de los polígonos efectúen una transición a lo largo de los fotogramas intermedios comprendidos entre un fotograma clave y el siguiente.

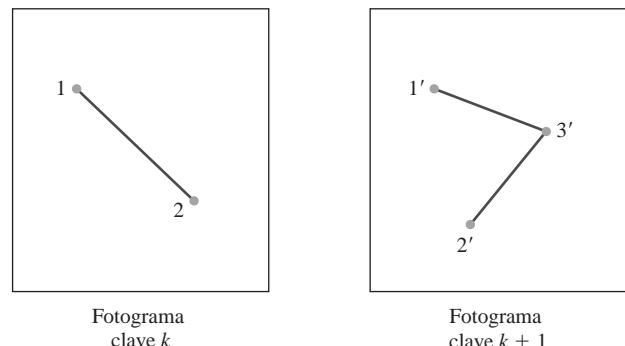
Dados dos fotogramas clave, cada uno de ellos con un número diferente de segmentos de línea que especifican la transformación de un objeto, podemos primero ajustar la especificación del objeto en uno de los fotogramas de modo que el número de aristas poligonales (o el número de vértices de los polígonos) sea el mismo para los dos fotogramas. Esta etapa de preprocesamiento se ilustra en la Figura 13.8. Un segmento de línea recta en el fotograma clave  $k$  se transforma en dos segmentos de línea en el fotograma clave  $k + 1$ . Puesto que el fotograma  $k + 1$  tiene un vértice adicional, añadimos un vértice entre los vértices 1 y 2 en el fotograma clave  $k$  para equilibrar el número de vértices (y aristas) en los dos fotogramas clave. Utilizando interpolación lineal para generar los fotogramas intermedios, efectuamos la transición del vértice añadido en el fotograma clave  $k$  hacia el vértice  $3'$  según el trayecto lineal mostrado en la Figura 13.9. En la Figura 13.10 se proporciona un ejemplo de un triángulo que se expande linealmente en un cuadrilátero. Las Figuras 13.11 y 13.12 muestran ejemplos de morfismo en anuncios de televisión.

Podemos enunciar reglas de procesamiento generales para ecualizar los fotogramas clave en términos del número de aristas o del número de vértices que haya que añadir a un fotograma clave. Vamos a considerar primero la ecualización del número de aristas, donde los parámetros  $L_k$  y  $L_{k+1}$  denotan el número de segmentos de línea en dos fotogramas consecutivos. El número máximo y mínimo de líneas que habrá que ecualizar será el siguiente:

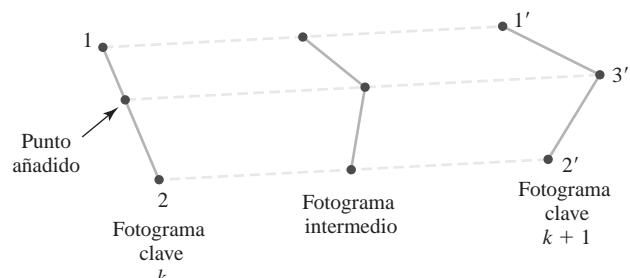
$$L_{\max} = \max(L_k, L_{k+1}), \quad L_{\min} = \min(L_k, L_{k+1}) \quad (13.1)$$

A continuación, calculamos los siguientes dos valores:

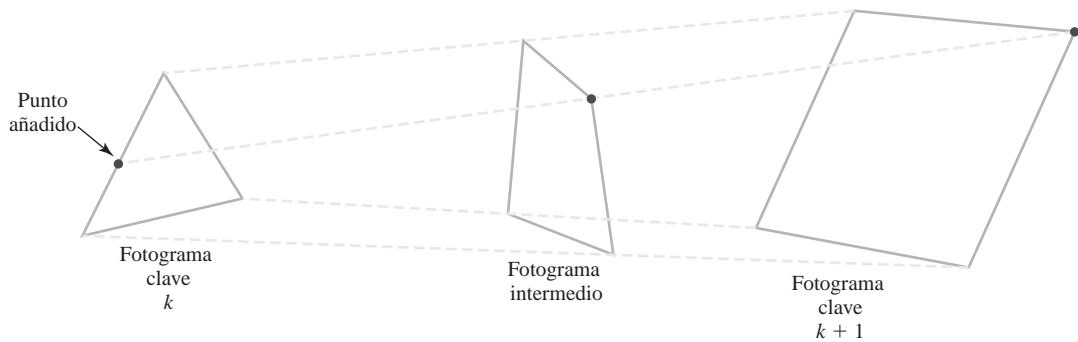
$$\begin{aligned} N_e &= L_{\max} \bmod L_{\min} \\ N_s &= \text{int}\left(\frac{L_{\max}}{L_{\min}}\right) \end{aligned} \quad (13.2)$$



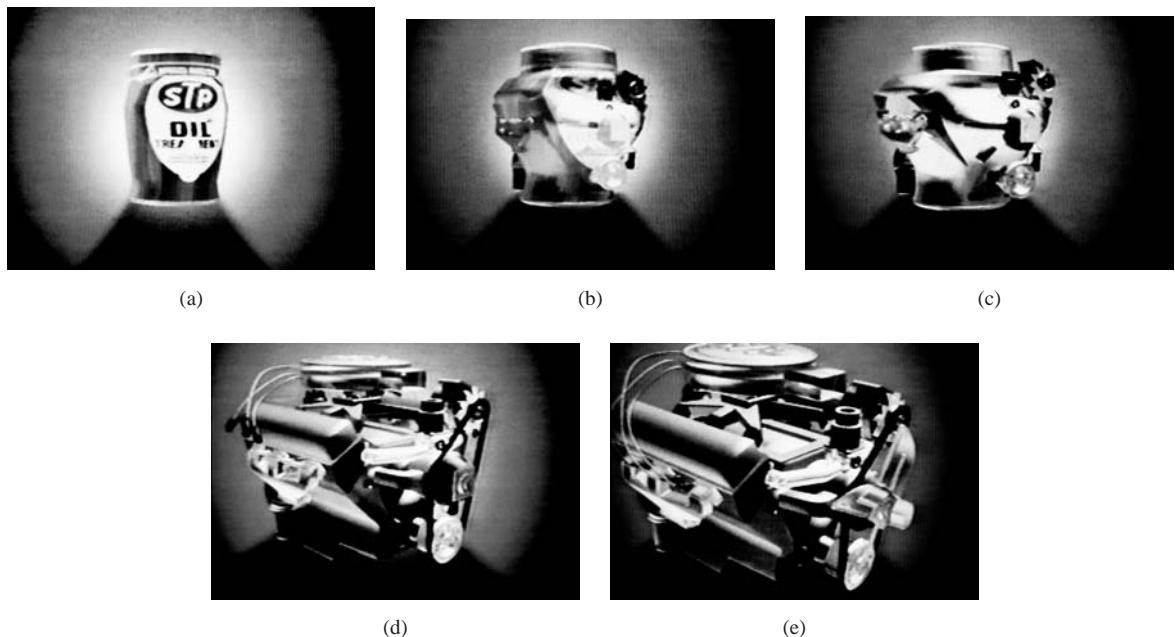
**FIGURA 13.8.** Una arista con dos vértices 1 y 2 en el fotograma clave  $k$  evoluciona para convertirse en dos aristas conectadas en fotograma clave  $k + 1$ .



**FIGURA 13.9.** Interpolación lineal para transformar un segmento de línea en el fotograma clave  $k$  en dos segmentos de línea conectados en el fotograma clave  $k + 1$ .



**FIGURA 13.10.** Interpolación lineal para transformar un triángulo en un cuadrilátero.



**FIGURA 13.11.** Transformación de una lata de aceite para automóviles STP en un motor de automóvil. (Cortesía de Silicon Graphics, Inc.)

Las etapas de preprocessamiento para la ecualización de aristas pueden llevarse a cabo entonces con los dos siguientes procedimientos:

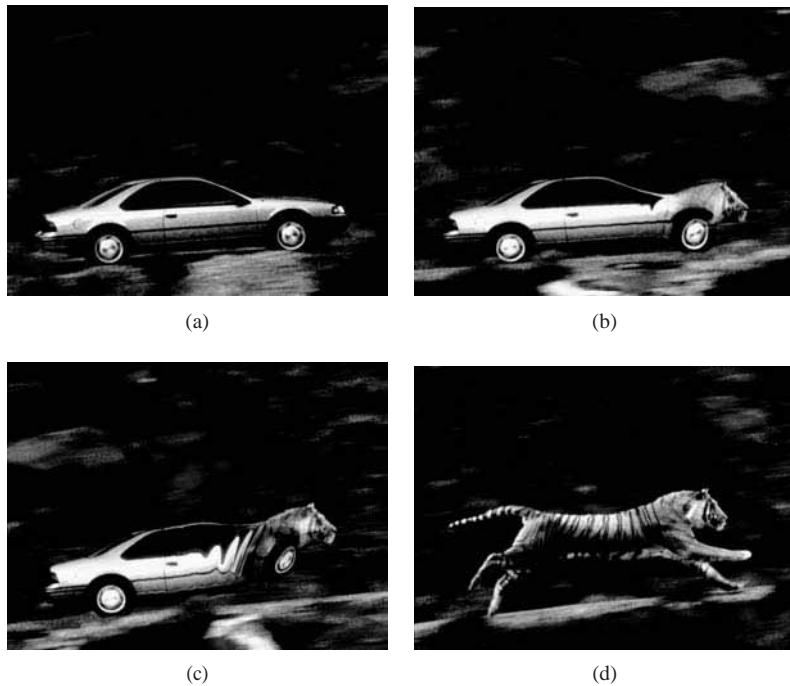
- (1) Dividir  $N_e$  aristas de  $fotogramaclave_{\min}$  en  $N_s + 1$  secciones.
- (2) Dividir las líneas restantes de  $fotogramaclave_{\min}$  en  $N_s$  secciones.

Como ejemplo, si  $L_k = 15$  y  $L_{k+1} = 11$ , dividiríamos cuatro líneas de  $fotogramaclave_{k+1}$  en dos secciones cada una. Las líneas restantes de  $fotogramaclave_{k+1}$  se dejan intactas.

Si ecualizamos el número de vértices, podemos utilizar los parámetros  $V_k$  y  $V_{k+1}$  para denotar el número de vértices en dos fotogramas clave consecutivos. En este caso, determinamos los números máximo y mínimo de vértices de la forma siguiente:

$$V_{\max} = \max(V_k, V_{k+1}), \quad V_{\min} = \min(V_k, V_{k+1}) \quad (13.3)$$

A continuación calculamos los siguientes dos valores:



**FIGURA 13.12.** Transformación de un automóvil en un tigre. (Cortesía de Exxon Company USA y Pacific Data Images.)

$$N_{ls} = (V_{\max} - 1) \bmod (V_{\min} - 1) \quad (13.4)$$

$$N_p = \text{int}\left(\frac{V_{\max} - 1}{V_{\min} - 1}\right)$$

Estos dos valores se utilizan entonces para llevar a cabo la ecualización de vértices mediante los procedimientos:

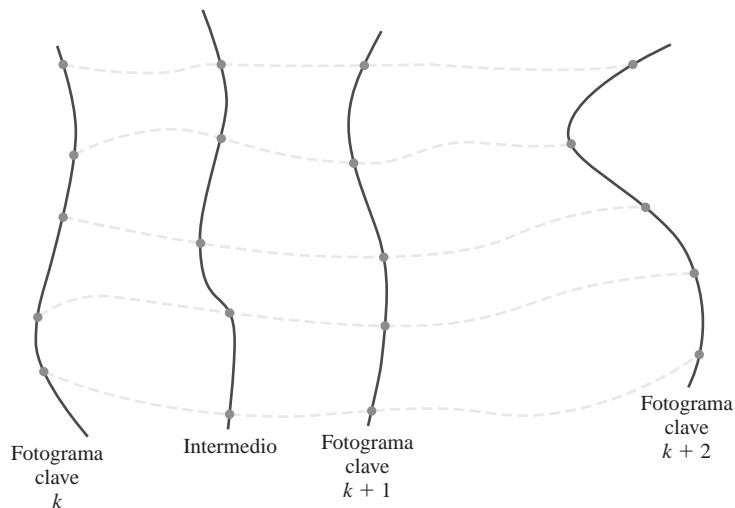
- (1) Añadir  $N_p$  puntos  $N_{ls}$  secciones de línea de  $\text{fotogramaclave}_{\min}$ .
- (2) Añadir  $N_p - 1$  puntos a las aristas restantes de  $\text{fotogramaclave}_{\min}$ .

Para el ejemplo de transformación de un triángulo en un cuadrilátero,  $V_k = 3$  y  $V_{k+1} = 4$ . Tanto  $N_{ls}$  como  $N_p$  son 1, por lo que añadiríamos un punto a una arista de  $\text{fotogramaclave}_k$ . No se añadiría ningún punto a las líneas restantes de  $\text{fotogramaclave}_k$ .

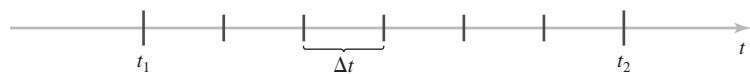
### Simulación de aceleraciones

A menudo se utilizan técnicas de ajuste de curvas para especificar los trayectos de animación entre fotogramas clave. Dadas las posiciones de los vértices en los fotogramas clave, podemos ajustar las posiciones mediante trayectos lineales o no lineales. La Figura 13.13 ilustra un ajuste no lineal de las posiciones en los fotogramas clave. Y para simular aceleraciones, podemos ajustar el espaciado temporal correspondiente a los fotogramas intermedios.

Si el movimiento debe tener lugar a velocidad constante (aceleración cero), utilizamos intervalos temporales iguales para los fotogramas intermedios. Por ejemplo, con  $n$  fotogramas intermedios y con sendos tiempos  $t_1$  y  $t_2$  para los fotogramas clave (Figura 13.14), el intervalo temporal entre los fotogramas clave se divide en  $n + 1$  subintervalos iguales, lo que nos da un espaciado de los fotogramas intermedios igual a:



**FIGURA 13.13.** Ajuste de las posiciones de los vértices en los fotogramas clave mediante *splines* no lineales.



**FIGURA 13.14.** Posiciones de los fotogramas intermedios para movimiento a velocidad constante.

$$\Delta t = \frac{t_2 - t_1}{n+1} \quad (13.5)$$

El tiempo correspondiente al fotograma intermedio  $j$ -ésimo será:

$$tB_j = t_1 + j\Delta t, \quad j = 1, 2, \dots, n \quad (13.6)$$

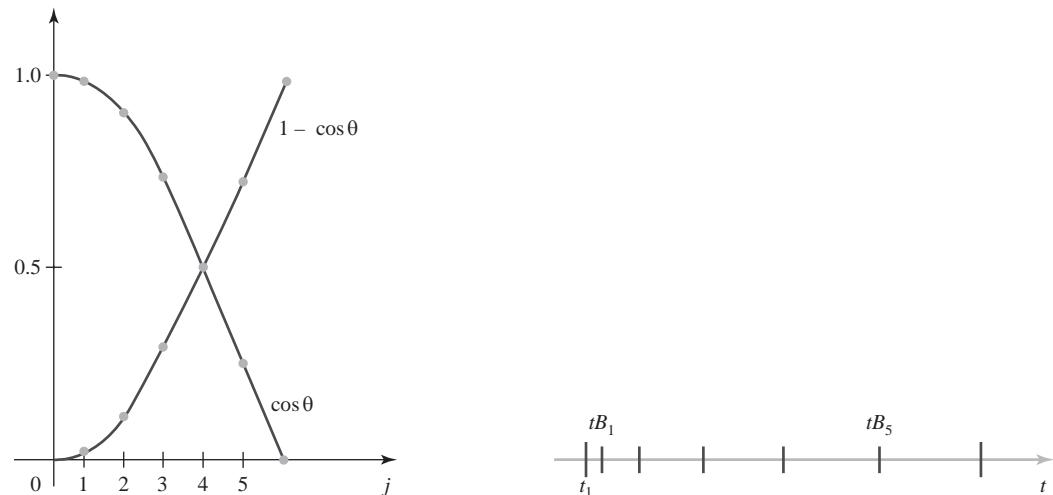
y este tiempo se utiliza para calcular las coordenadas de posición, los colores y otros parámetros físicos para dicho fotograma de la secuencia.

Usualmente, hacen falta cambios de velocidad (aceleración distinta de cero) en algún punto de las secuencias de animación o de dibujos animados, particularmente al principio y al final de un movimiento. Las partes de arranque y de parada de un trayecto de animación se suelen modelar con splines o funciones trigonométricas, pero también se han aplicado funciones temporales parabólicas y cúbicas para modelar las aceleraciones. Los paquetes de animación suelen proporcionar funciones trigonométricas para simular las aceleraciones. Para modelar una velocidad creciente (aceleración positiva), lo que puede hacerse es incrementar el espacio temporal entre fotogramas, de modo que se produzcan cambios más grandes en la posición a medida que aumenta la velocidad del objeto. Podemos obtener un tamaño creciente para el intervalo temporal mediante la función:

$$1 - \cos \theta, \quad 0 < \theta < \pi/2$$

Para  $n$  fotogramas intermedios, el tiempo correspondiente al fotograma intermedio  $j$ -ésimo se calcularía como:

$$tB_j = t_1 + j\Delta t \left[ 1 - \cos \frac{j\pi}{2(n+1)} \right], \quad j = 1, 2, \dots, n \quad (13.7)$$



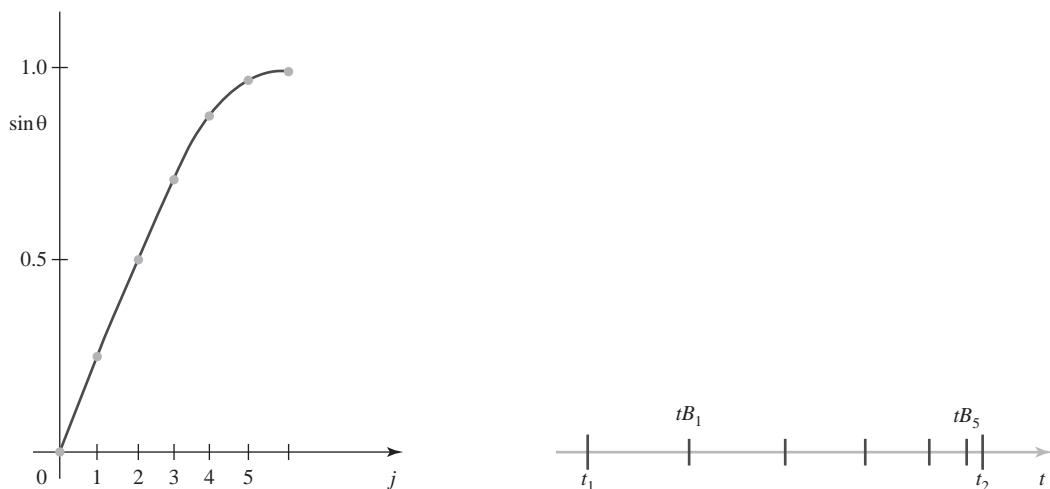
**FIGURA 13.15.** Una función de aceleración trigonométrica y el correspondiente espaciado de fotogramas intermedios para  $n = 5$  y  $\theta = j\pi/12$  en la Ecuación 13.7, lo que produce cambios crecientes en los valores de las coordenadas a medida que el objeto pasa de un intervalo temporal a otro.

donde  $\Delta t$  es la diferencia de tiempo entre los dos fotogramas clave. La Figura 13.15 muestra una gráfica de la función trigonométrica de aceleración y del espaciado de los fotogramas intermedios para  $n = 5$ .

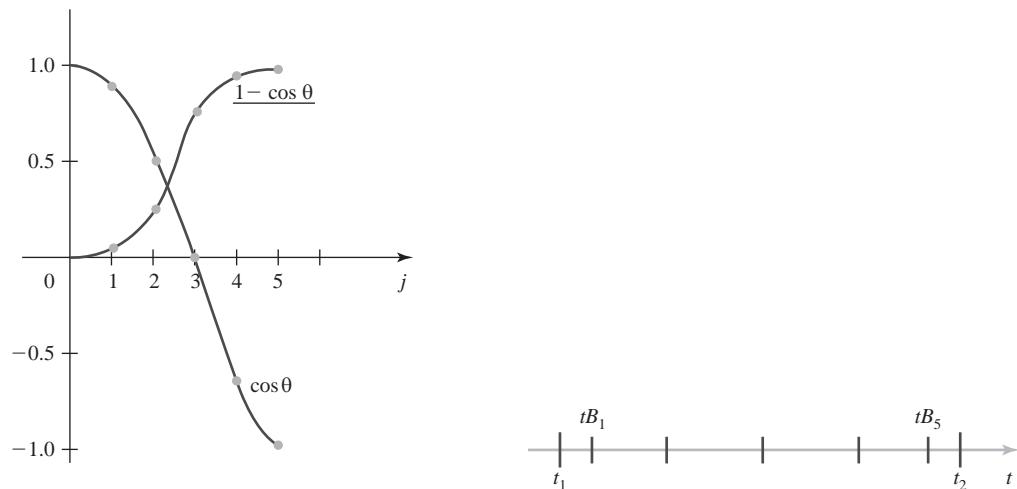
Podemos modelar una velocidad decreciente (deceleración) utilizando la función  $\sin \theta$ , con  $0 < \theta < \pi/2$ . El tiempo correspondiente a un fotograma intermedio se determina entonces mediante la fórmula

$$tB_j = t_1 + \Delta t \sin \frac{j\pi}{2(n+1)}, \quad j = 1, 2, \dots, n \quad (13.8)$$

En la Figura 13.16 se muestra una gráfica de esta función y el tamaño decreciente de los intervalos temporales, para cinco fotogramas intermedios.



**FIGURA 13.16.** Una función de deceleración trigonométrica y el correspondiente espaciado de los fotogramas intermedios para  $n = 5$  y  $\theta = j\pi/12$  en la Ecuación 13.8, lo que produce cambios decrecientes en las coordenadas a medida que el objeto pasa de un intervalo temporal a otro.



**FIGURA 13.17.** La función trigonométrica de aceleración-deceleración  $(1 - \cos \theta)/2$  y el correspondiente espaciado de los fotogramas intermedios para  $n = 5$  en la Ecuación 13.9.

A menudo los movimientos contienen tanto aceleraciones como frenados. Podemos modelar una combinación de velocidad creciente-decreciente incrementando primero el espaciado temporal de los fotogramas intermedios y luego reduciéndolo. Una función para poder conseguir estos cambios de carácter temporal es:

$$\frac{1}{2}(1 - \cos \theta), \quad 0 < \theta < \pi/2$$

El tiempo correspondiente al fotograma intermedio  $j$ -ésimo se calculará ahora como:

$$t_{B_j} = t_1 + \Delta t \left\{ \frac{1 - \cos[j\pi/(n+1)]}{2} \right\}, \quad j = 1, 2, \dots, n \quad (13.9)$$

donde  $\Delta t$  denota la diferencia temporal entre dos fotogramas clave. Los intervalos temporales para un objeto en movimiento se incrementarán primero y luego se reducirán, como se muestra en la Figura 13.17.

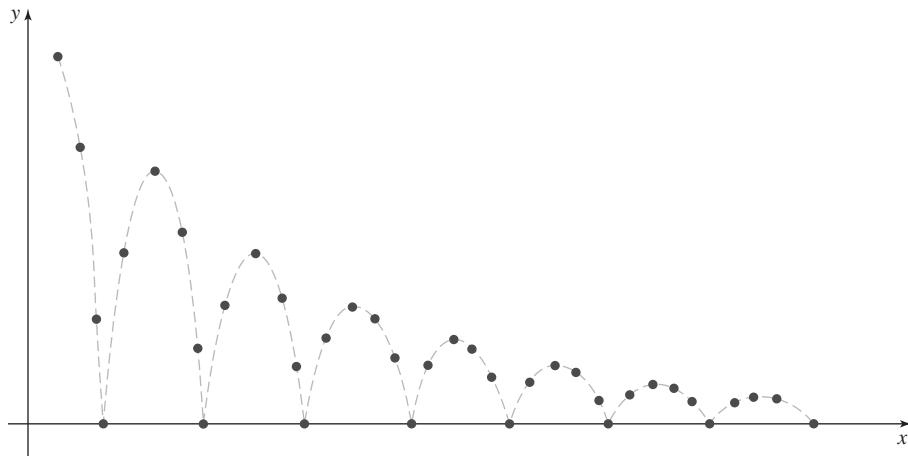
El procesamiento de los fotogramas intermedios se simplifica si modelamos inicialmente objetos «esqueleto» (alámbricos), de modo que puedan ajustarse interactivamente a las secuencias de movimiento. Después de definir completamente la secuencia de animación, puede procederse a obtener la representación de los objetos en imágenes.

## 13.7 ESPECIFICACIONES DE MOVIMIENTOS

Los métodos generales para describir una secuencia de animación van desde la especificación explícita de las trayectorias de movimiento hasta una descripción de las interacciones que producen dichos movimientos. Así, podemos definir el modo en que una animación debe transcurrir proporcionando los parámetros de transformación, los parámetros de las trayectorias de movimiento, las fuerzas que deben actuar sobre los objetos o los detalles sobre cómo interactúan los objetos con el fin de producir movimientos.

### Especificación directa del movimiento

El método más simple para definir una animación consiste en la *especificación directa del movimiento*, lo que nos da los parámetros de transformación geométrica. Con este método, lo que hacemos es configurar explícitamente



**FIGURA 13.18.** Aproximación del movimiento de una bola que rebota sobre el suelo mediante una función seno amortiguada (Ecuación 13.10).

tamente los valores de los ángulos de rotación y de los vectores de traslación. A continuación, se aplican las matrices de transformación geométrica para transformar las posiciones de coordenadas. Alternativamente, podríamos usar una ecuación de aproximación en la que aparezcan estos parámetros para especificar ciertos tipos de movimientos. Por ejemplo, podemos aproximar la trayectoria de una bola que rebote en el suelo utilizando una curva seno amortiguada y rectificada (Figura 13.18):

$$y(x) = A|\sin(\omega x + \theta_0)|e^{-kx} \quad (13.10)$$

donde  $A$  es la amplitud inicial (altura de la bola sobre el suelo),  $\omega$  es la frecuencia angular,  $\theta_0$  es el ángulo de fase y  $k$  es el coeficiente de amortiguamiento. Este método de especificación del movimiento resulta particularmente útil para las secuencias de animación simples programadas por el usuario.

## Sistemas dirigidos por objetivos

En el extremo opuesto, podemos especificar los movimientos que deben tener lugar en términos generales que describan de manera abstracta las acciones en función de los resultados finales. En otras palabras, una animación se especifica en términos del estado final de los movimientos. Estos sistemas se denominan *dirigidos por objetivos*, ya que los valores de los parámetros de movimiento se determinan a partir de los objetivos de la animación. Por ejemplo, podríamos especificar que un determinado objeto debe «caminar» o «correr» hasta un destino concreto. O podríamos especificar que queremos que el objeto «agarre» a algún otro objeto específico. Las directivas de entrada se interpretan entonces en términos de los movimientos componentes que permitirán llevar a cabo la tarea descrita. El movimiento de los seres humanos, por ejemplo, puede definirse como una estructura jerárquica de submovimiento para el torso, los miembros, etc. Así, cuando se proporciona un objeto tal como «caminar hasta la puerta» se calculan los movimientos requeridos del torso y de los miembros para llevar a cabo esta acción.

## Cinemática y dinámica

También podemos construir secuencias de animación utilizando descripciones *cinemáticas* o *dinámicas*. Con una descripción cinemática, especificamos la animación proporcionando los parámetros de movimiento (posición, velocidad y aceleración) sin referencia a las causas ni a los objetivos del movimiento. Para una velocidad constante (aceleración cero), designamos los movimientos de los cuerpos rígidos de una escena proporcionando una posición inicial y un vector de velocidad para cada objeto. Como ejemplo, si la velocidad se

específica como  $(3, 0, -4)$  km/seg, entonces este vector proporcionará la dirección de la trayectoria lineal de movimiento y la velocidad (módulo del vector) será igual a 5 km/seg. Si también especificamos las aceleraciones (tasa de cambio de la velocidad), podemos modelar arranques, paradas y trayectos de movimiento curvos. La especificación cinemática de un movimiento también puede proporcionarse simplemente describiendo la trayectoria del movimiento. Esto se suele hacer mediante curvas de tipo *spline*.

Un enfoque alternativo consiste en utilizar *cinemática inversa*. Con este método, especificamos las posiciones inicial y final de los objetos en instantes determinados y es el sistema el que se encarga de calcular los parámetros del movimiento. Por ejemplo, suponiendo una aceleración cero, podemos determinar la velocidad constante que permitirá conseguir el movimiento de un objeto desde la posición inicial hasta la posición final. Este método se suele utilizar para objetos complejos proporcionando las posiciones y orientaciones de un nodo terminal de un objeto, como por ejemplo una mano o un pie. El sistema determina entonces los parámetros de movimiento de los otros nodos que hacen falta para conseguir el movimiento deseado.

Las descripciones dinámicas, por el contrario, requieren la especificación de las fuerzas que producen las velocidades y aceleraciones. La descripción del comportamiento de los objetos en términos de la influencia de las fuerzas se suele denominar *modelado físico* (Capítulo 8). Como ejemplos de fuerzas que afectan al movimiento de los objetos podemos citar las fuerzas electromagnéticas, gravitatorias, de fricción y otras fuerzas mecánicas.

Los movimientos de los objetos se obtienen a partir de las ecuaciones de las fuerzas que describen leyes físicas, como por ejemplo las leyes de Newton del movimiento para los procesos gravitatorios y de fricción, las ecuaciones de Euler o de Navier-Stokes que describen el flujo de fluidos y las ecuaciones de Maxwell para las fuerzas electromagnéticas. Por ejemplo, la forma general de la segunda ley de Newton para una partícula de masa  $m$  es:

$$\mathbf{F} = \frac{d}{dt}(m\mathbf{v}) \quad (13.11)$$

donde  $\mathbf{F}$  es el vector de la fuerza y  $\mathbf{v}$  es el vector velocidad. Si la masa es constante, resolvemos la ecuación  $\mathbf{F} = m\mathbf{a}$ , donde  $\mathbf{a}$  representa el vector de aceleración. En caso contrario, la masa será una función del tiempo, como sucede en el movimiento relativista o en el movimiento de naves espaciales que consuman cantidades no despreciables de combustible por unidad de tiempo. También podemos utilizar la *dinámica inversa* para obtener las fuerzas, dadas las posiciones inicial y final de los objetos y el tipo de movimiento requerido.

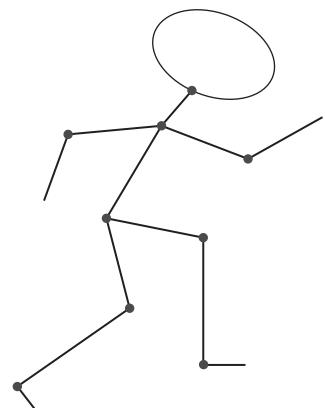
Entre las aplicaciones del modelado físico se incluyen los sistemas complejos de cuerpos rígidos y también otros sistemas no rígidos como las ropas y los materiales plásticos. Normalmente, se utilizan métodos numéricos para obtener los parámetros de movimiento incrementalmente a partir de las ecuaciones dinámicas, utilizando condiciones iniciales de valores de contorno.

## 13.8 ANIMACIÓN DE FIGURAS ARTICULADAS

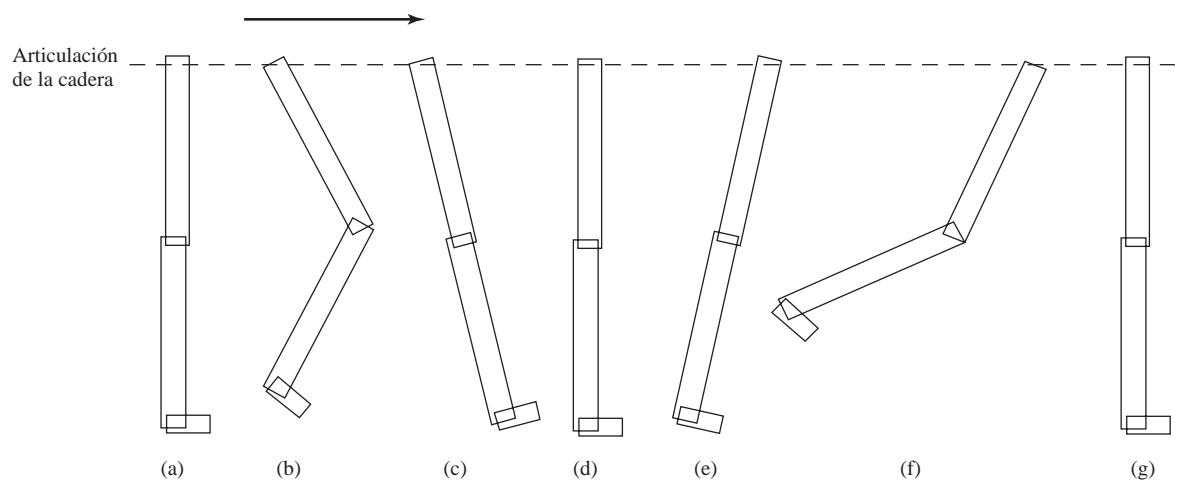
---

Una técnica básica para animar personajes humanos, animales, insectos y otras criaturas consiste en modelarlas como **figuras articuladas**, que son estructuras jerárquicas compuestas de un conjunto de enlaces rígidos conectados mediante uniones rotatorias (Figura 13.19). En términos menos formales, esto simplemente quiere decir que modelamos los objetos animados como si fueran esqueletos simplificados, los cuales podemos envolver luego con superficies que representen la piel, el pelo, las plumas, las ropas u otros tipos de recubrimientos.

Los puntos de conexión de una figura articulada se sitúan en los hombros, las caderas, las rodillas y otras articulaciones del esqueleto, y esos puntos de unión siguen unas trayectorias de movimiento especificadas a medida que el cuerpo se traslada. Por ejemplo, cuando se especifica un movimiento para un objeto, el hombro se mueve automáticamente de una cierta forma y, a medida que el hombro se mueve, los brazos también lo hacen. Con estos sistemas, se definen diferentes tipos de movimientos, como por ejemplo andar, correr o saltar, y esos movimientos se asocian con movimientos concretos de las uniones y de los enlaces conectados a ellas.



**FIGURA 13.19.** Una figura simple articulada con nueve uniones y doce enlaces conectados, sin contar la cabeza oval.



**FIGURA 13.20.** Posibles movimientos para un conjunto de enlaces conectados que representan una pierna llevando a cabo la acción de andar.

Por ejemplo, podríamos definir como en la Figura 13.20 un conjunto de movimientos para una pierna que estuviera efectuando la acción de andar. La articulación de la cadera se mueve hacia adelante según una línea horizontal, mientras que los enlaces conectados realizan una serie de movimientos en torno a las articulaciones de la cadera, de la rodilla y del talón. Comenzando con una pierna recta (Figura 13.20(a)), el primer movimiento consiste en doblar la rodilla a medida que la cadera se mueve hacia adelante (Figura 13.20(b)). Entonces, la pierna avanza, vuelve a la posición vertical y empuja hacia atrás, como se muestra en las Figuras 13.20(c), (d) y (e). Los movimientos finales son un movimiento amplio hacia atrás y una vuelta a la posición vertical, como en las Figuras 13.20(f) y (g). Este ciclo de movimiento se repite mientras dure la animación, a medida que el personaje recorre una distancia especificada o hasta que transcurra un determinado intervalo de tiempo.

A medida que se mueve un personaje, se incorporan otros movimientos a las diversas articulaciones. Puede aplicarse un movimiento sinusoidal, a menudo de amplitud variable, a las caderas para que éstas se muevan con respecto al torso. De la misma forma, puede aplicarse un movimiento giratorio a los hombros y también la cabeza puede moverse hacia arriba y hacia abajo.

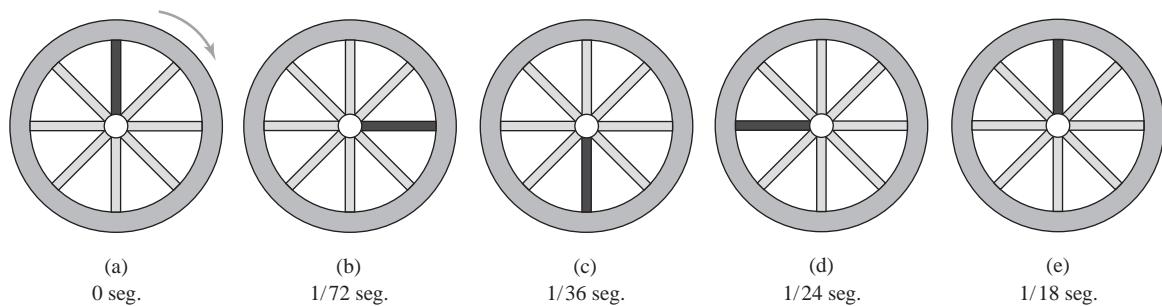
En la animación de personajes se utilizan tanto descripciones cinemáticas del movimiento como descripciones basadas en cinemática inversa. La especificación del movimiento de las articulaciones suele ser una tarea no demasiado complicada, pero la cinemática inversa puede también resultar útil para generar movi-

mientos simples sobre un terreno arbitrario. Para una figura complicada, la cinemática inversa puede no producir una secuencia de animación única, ya que, por ejemplo, puede que sean posibles muchos movimientos rotatorios distintos para un conjunto especificado de condiciones iniciales y finales. En tales casos, puede obtenerse una solución única añadiendo más restricciones al sistema, como por ejemplo el principio de conservación de la cantidad de movimiento.

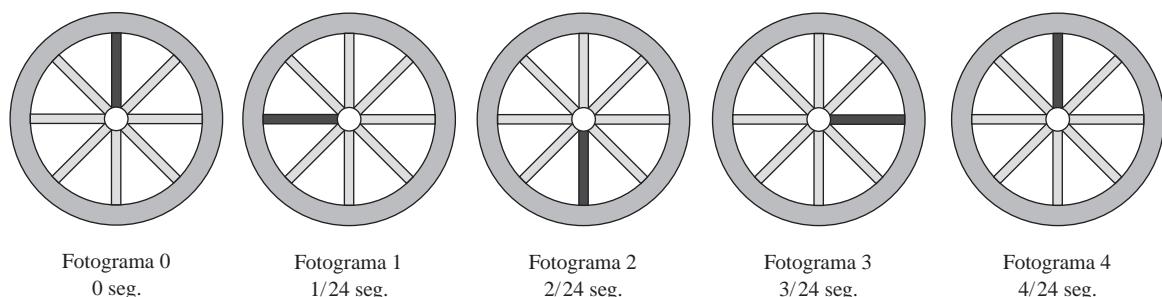
## 13.9 MOVIMIENTOS PERIÓDICOS

Cuando construimos una animación con patrones de movimiento repetitivos, como por ejemplo un objeto giratorio, necesitamos asegurarnos de muestrear el movimiento (Sección 4.17) con la suficiente frecuencia como para representar los movimientos correctamente. En otras palabras, el movimiento debe estar sincronizado con la tasa de generación de imágenes, para poder mostrar un número de imágenes por ciclo lo suficientemente alto como para que se perciba el movimiento real. En caso contrario, puede que la animación se visualice de forma incorrecta.

Un ejemplo típico de imágenes de movimiento periódico submuestreadas es la rueda de tren en una película del Oeste que parece estar girando en la dirección incorrecta. La Figura 13.21 ilustra un ciclo completo de la rotación de una rueda de tren, con un radio de distinto color que da 18 vueltas por segundo en el sentido de las agujas del reloj. Si este movimiento se graba en una película a la velocidad normal de proyección de 24 imágenes por segundo, entonces las primeras cinco imágenes correspondientes a este movimiento serían las que se muestran en la Figura 13.22. Puesto que la rueda completa  $\frac{3}{4}$  de vuelta cada  $\frac{1}{24}$  de segundo, sólo se genera un fotograma de la animación por cada ciclo y la rueda parece estar girando en la dirección opuesta (en sentido contrario a las agujas del reloj).



**FIGURA 13.21.** Cinco posiciones del radio durante un ciclo de movimiento de una rueda que está girando a 18 revoluciones por segundo.



**FIGURA 13.22.** Los cinco primeros fotogramas de película para la rueda giratoria de la Figura 13.21, producidos con una velocidad de 24 imágenes por segundo.

En las animaciones generadas por computadora, podemos controlar la velocidad de muestreo de un movimiento periódico ajustando los parámetros de movimiento. Por ejemplo, podemos configurar el incremento angular para el movimiento de un objeto giratorio de modo que se generen múltiples fotogramas en cada revolución. Así, un incremento de  $3^\circ$  para un ángulo de rotación produce 120 pasos de movimiento durante una revolución, mientras que un incremento de  $4^\circ$  genera 90 posiciones. Para movimientos más rápidos, pueden utilizarse pasos de rotación más amplios, siempre y cuando el número de muestras por ciclo no sea demasiado pequeño y el movimiento se visualice claramente. Cuando haya que animar objetos complejos, también deberemos tener en cuenta el efecto que el tiempo de generación del fotograma pueda tener sobre la tasa de refresco, como se explica en la Sección 13.1. El movimiento de un objeto complejo puede ser mucho más lento de lo que deseamos si se tarda demasiado en generar cada fotograma de animación.

Otro factor que tenemos que considerar en la visualización de un movimiento repetitivo es el efecto de los redondeos en los cálculos de los parámetros de movimiento. Como hemos observado en la Sección 5.4, podemos reinicializar periódicamente los valores de los parámetros para evitar que la acumulación de los errores produzca movimientos erráticos. Para una rotación continua, podríamos reinicializar los valores de los parámetros una vez por ciclo ( $360^\circ$ ).

## 13.10 PROCEDIMIENTOS DE ANIMACIÓN EN OpenGL

---

En la biblioteca básica hay disponibles operaciones de manipulación de imágenes de barrido (Sección 5.7) y funciones de asignación de índices de colores, mientras que en GLUT hay disponibles rutinas para modificar los valores de las tablas de colores (Sección 4.3). Otras operaciones de animación por barrido sólo están disponibles como rutinas GLUT, porque dependen del sistema de gestión de ventanas que se utilice. Además, algunas características para animación por computadora tales como el doble búfer pueden no estar incluidas en algunos sistemas hardware.

Las operaciones de doble búfer, si están disponibles, se activan utilizando el siguiente comando GLUT:

```
glutInitDisplayMode (GLUT_DOUBLE);
```

Esto proporciona dos búferes, denominados *búfer frontal* y *búfer trasero*, que podemos utilizar alternativamente para refrescar la imagen de pantalla. Mientras uno de los búferes actúa como búfer de refresco para la ventana de visualización actual, puede irse construyendo la siguiente imagen de la animación en el otro búfer. Podemos especificar cuándo hay que intercambiar los roles de los dos búferes mediante el comando:

```
glutSwapBuffers ( );
```

Para determinar si están disponibles las operaciones de doble búfer en un sistema, puede efectuarse la siguiente consulta:

```
glGetBooleanv (GL_DOUBLEBUFFER, status);
```

Se devolverá un valor `GL_TRUE` al parámetro de matriz `status` si hay disponibles en el sistema tanto un búfer frontal como otro trasero. En caso contrario, el valor devuelto es `GL_FALSE`. Para una animación continua, también podemos usar:

```
glutIdleFunc (animationFcn);
```

donde al parámetro `animationFcn` se le puede asignar el nombre de un procedimiento que se encargue de realizar las operaciones de incremento de los parámetros de animación. Este procedimiento se ejecutará de modo continuo cuando no haya sucesos de la ventana de visualización que procesar. Para desactivar la función `glutIdleFunc`, podemos asignar a este argumento el valor `NULL` o el valor 0.

En el siguiente fragmento de código se proporciona un ejemplo de programa de animación que hace girar de modo continuo un hexágono regular en el plano *xy* en torno a un eje *z*. El origen de las coordenadas de pantalla tridimensionales se coloca en el centro de la ventana de visualización, de modo que el eje *z* pasa a través

de esta posición central. En el procedimiento `init`, utilizamos una lista de visualización para especificar la descripción del hexágono regular, cuya posición central está originalmente en la posición (150, 150) de las coordenadas de pantalla y que tiene un radio (distancia desde el centro del polígono a cualquiera de sus vértices) igual a 100 píxeles. En la función de visualización, `displayHex`, especificamos una rotación inicial de 0° en torno al eje *z* e invocamos la rutina `glutSwapBuffers`. Para activar la rotación, utilizamos el procedimiento `mouseFcn` que incrementa continuamente el ángulo de rotación en 3° cuando pulsamos el botón central del ratón.. El cálculo del ángulo de rotación incrementado se lleva a cabo en el procedimiento `rotateHex`, que es invocado por la rutina `glutIdleFunc` en el procedimiento `mouseFcn`. Detenemos la rotación pulsando el botón derecho del ratón, lo que hace que se invoque `glutIdleFunc` con un argumento `NULL`.

---

```
#include <GL/glut.h>
#include <math.h>
#include <stdlib.h>

const double TWO_PI = 6.2831853;

GLsizei winWidth = 500, winHeight = 500; // Tamaño inicial ventana visualización.
GLuint regHex; // Definir nombre para lista visualización.
static GLfloat rotTheta = 0.0;

class scrPt {
public:
    GLint x, y;
};

static void init (void)
{
    scrPt hexVertex;
    GLdouble hexTheta;
    GLint k;

    glClearColor (1.0, 1.0, 1.0, 0.0);

    /* Establecer lista de visualización para un hexágono regular de color rojo.
     * Los vértices del hexágono son seis puntos equiespaciados
     * situados sobre una circunferencia. */
    regHex = glGenLists (1);
    glNewList (regHex, GL_COMPILE);
    glColor3f (1.0, 0.0, 0.0);
    glBegin (GL_POLYGON);
        for (k = 0; k < 6; k++) {
            hexTheta = TWO_PI * k / 6;
            hexVertex.x = 150 + 100 * cos (hexTheta);
            hexVertex.y = 150 + 100 * sin (hexTheta);
            glVertex2i (hexVertex.x, hexVertex.y);
        }
    glEnd ( );
    glEndList ( );
}

void displayHex (void)
{
```

```
glClear (GL_COLOR_BUFFER_BIT);

glPushMatrix ();
glRotatef (rotTheta, 0.0, 0.0, 1.0);
glCallList (regHex);
glPopMatrix ();

glutSwapBuffers ( );

glFlush ( );
}

void rotateHex (void)
{
    rotTheta += 3.0;
    if (rotTheta > 360.0)
        rotTheta -= 360.0;

    glutPostRedisplay ( );
}

void winReshapeFcn (GLint newWidth, GLint newHeight)
{
    glViewport (0, 0, (GLsizei) newWidth, (GLsizei) newHeight);

    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    gluOrtho2D (-320.0, 320.0, -320.0, 320.0);

    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity ();

    glClear (GL_COLOR_BUFFER_BIT);
}

void mouseFcn (GLint button, GLint action, GLint x, GLint y)
{
    switch (button) {
        case GLUT_MIDDLE_BUTTON: // Comenzar la rotación.
            if (action == GLUT_DOWN)
                glutIdleFunc (rotateHex);
            break;
        case GLUT_RIGHT_BUTTON: // Detener la rotación.
            if (action == GLUT_DOWN)
                glutIdleFunc (NULL);
            break;
        default:
            break;
    }
}

void main (int argc, char** argv)
{
    glutInit (&argc, argv);
    glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);
```

```

glutInitWindowPosition (150, 150);
glutInitWindowSize (winWidth, winHeight);
glutCreateWindow ("Animation Example");

init ();
glutDisplayFunc (displayHex);
glutReshapeFunc (winReshapeFcn);
glutMouseFunc (mouseFcn);

glutMainLoop ();
}

```

## 13.11 RESUMEN

Podemos construir una secuencia de animación fotograma a fotograma o podemos generarla en tiempo real. Cuando se construyen y almacenan fotogramas independientes de una animación, los fotogramas pueden posteriormente transferirse a una película o mostrarse en una rápida sucesión sobre un monitor de vídeo. Las animaciones que incluyen escenas y movimientos complejos suelen crearse de fotograma en fotograma, mientras que las secuencias de movimiento más simples se muestran en tiempo real.

En un sistema de barrido, pueden usarse métodos de doble búfer para facilitar la visualización del movimiento. Se utiliza un búfer para registrar la pantalla, mientras que se carga en un segundo búfer los valores de los píxeles correspondientes al siguiente fotograma de la secuencia. Después, se intercambian los papeles de los dos búferes, usualmente al final de un ciclo de refresco.

Otro método de barrido para mostrar una animación consiste en realizar secuencias de movimiento utilizando transferencias en bloque de los píxeles. Las traslaciones se llevan a cabo mediante un simple movimiento de un bloque rectangular de píxeles desde una posición de búfer de imagen a otra. Asimismo, las rotaciones en incrementos de 90° pueden llevarse a cabo mediante una combinación de traslaciones e intercambios de filas-columnas dentro de la matriz de píxeles.

Pueden usarse métodos basados en tablas de colores para construir animaciones de barrido simples, almacenando una imagen de un objeto en múltiples ubicaciones del búfer de imagen y utilizando diferentes valores en la tabla de colores. Una imagen se almacena con el color de primer plano y las copias de la imagen en las otras ubicaciones tendrán un color de fondo. Intercambiando rápidamente los valores de color de primer plano y de fondo almacenados en la tabla de colores, podemos mostrar el objeto en varias posiciones de pantalla.

Son varias las etapas de desarrollo necesarias para producir una animación, comenzando con el guión, las definiciones de los objetos y la especificación de los fotogramas clave. El guión es un resumen de la acción, mientras que los fotogramas clave definen los detalles de los movimientos de los objetos para posiciones seleccionadas dentro de una secuencia de animación. Una vez definidos los fotogramas clave, se generan los fotogramas intermedios para conseguir un movimiento suave entre un fotograma clave y el siguiente. Una animación infográfica puede incluir especificaciones de movimiento para la «cámara», además de trayectorias de movimientos para los objetos y personajes que participen en la animación.

Se han desarrollado diversas técnicas para simular y enfatizar los efectos de movimiento. Los efectos de compresión y expansión son métodos estándar para resaltar las aceleraciones, y la modificación del tiempo entre unos fotogramas y otros permite conseguir variaciones de velocidad. Otros métodos incluyen movimientos preliminares de preparación, movimientos de seguimiento al final de la acción y métodos de variación del punto de atención que centran la imagen sobre una acción importante que esté teniendo lugar en la escena. Normalmente, se utilizan funciones trigonométricas para determinar el espaciado temporal de los fotogramas intermedios cuando los movimientos incluyen aceleraciones.

Las animaciones pueden generarse mediante software de propósito especial o utilizando un paquete gráfico de propósito general. Entre los sistemas disponibles para la animación automática por computadora se incluyen los sistemas basados en fotogramas clave, los sistemas parametrizados y los sistemas basados en *scripts*.

Muchas animaciones incluyen efectos de morfismo, en los que se hace que cambie la forma de un objeto. Estos efectos se consiguen utilizando fotogramas intermedios para efectuar la transición, transformando los puntos y líneas que definen un objeto en los puntos y líneas que definen el objeto final.

Los movimientos dentro de una animación pueden describirse por especificación directa del movimiento o pueden estar dirigidos por objetivos. Así, una animación puede definirse en términos de los parámetros de traslación y rotación, o los movimientos pueden describirse mediante ecuaciones o mediante parámetros cinemáticos o dinámicos. Las descripciones cinemáticas del movimiento especifican las posiciones, velocidades y aceleraciones; las descripciones dinámicas del movimiento se proporcionan en términos de las fuerzas que actúan sobre los objetos incluidos en una escena.

A menudo se utilizan figuras articuladas para modelar el movimiento de las personas y de los animales. Con este método, se definen enlaces rígidos dentro de una estructura jerárquica, conectados mediante articulaciones giratorias. Cuando se imprime movimiento a un objeto, cada subparte está programada para moverse de una forma concreta en respuesta al movimiento global.

La velocidad de muestreo para los movimientos periódicos debe producir los suficientes fotogramas por ciclo como para mostrar correctamente la animación. En caso contrario, pueden producirse movimientos erráticos o confusos.

Además de las operaciones de barrido y de los métodos basados en tablas de colores, hay disponibles algunas funciones en GLUT (OpenGL Utility Toolkit) para desarrollar programas de animación. Estas funciones proporcionan rutinas para las operaciones de doble búfer y para incrementar los parámetros de movimiento durante los intervalos de inactividad durante el procesamiento. En la Tabla 13.1 se enumeran las funciones GLUT para generar animaciones con programas OpenGL.

**TABLA 13.1. RESUMEN DE FUNCIONES DE ANIMACIÓN OpenGL.**

Función	Descripción
glutInitDisplayMode (GLUT_DOUBLE)	Activa las operaciones de doble búfer.
glutSwapBuffers	Intercambia los búferes de refresco frontal y trasero.
glGetBooleanv (GL_DOUBLEBUFFER, status)	Consulta al sistema para determinar si están disponibles las operaciones de doble búfer.
glutIdleFunc	Especifica una función para incrementar los parámetros de animación.

## REFERENCIAS

Los sistemas de animación por computadora se analizan en Thalmann y Thalmann (1985), Watt and Watt (1992), O'Rourke (1998), Maestri (1999 y 2002), Kerlow (2000), Gooch y Gooch (2001), Parent (2002), Pocock y Rosebush (2002) y Strothotte y Schlechtweg (2002). Las técnicas tradicionales de animación se exploran en Lasseter (1987), Thomas, Johnston y Johnston (1995) y Thomas y Lefkon (1997). Los métodos de morfismos se estudian en Hughes (1992), Kent, Carlson y Parent (1992), Sederberg y Greenwood (1992) y Gomes, Darsa, Costa y Velho (1999).

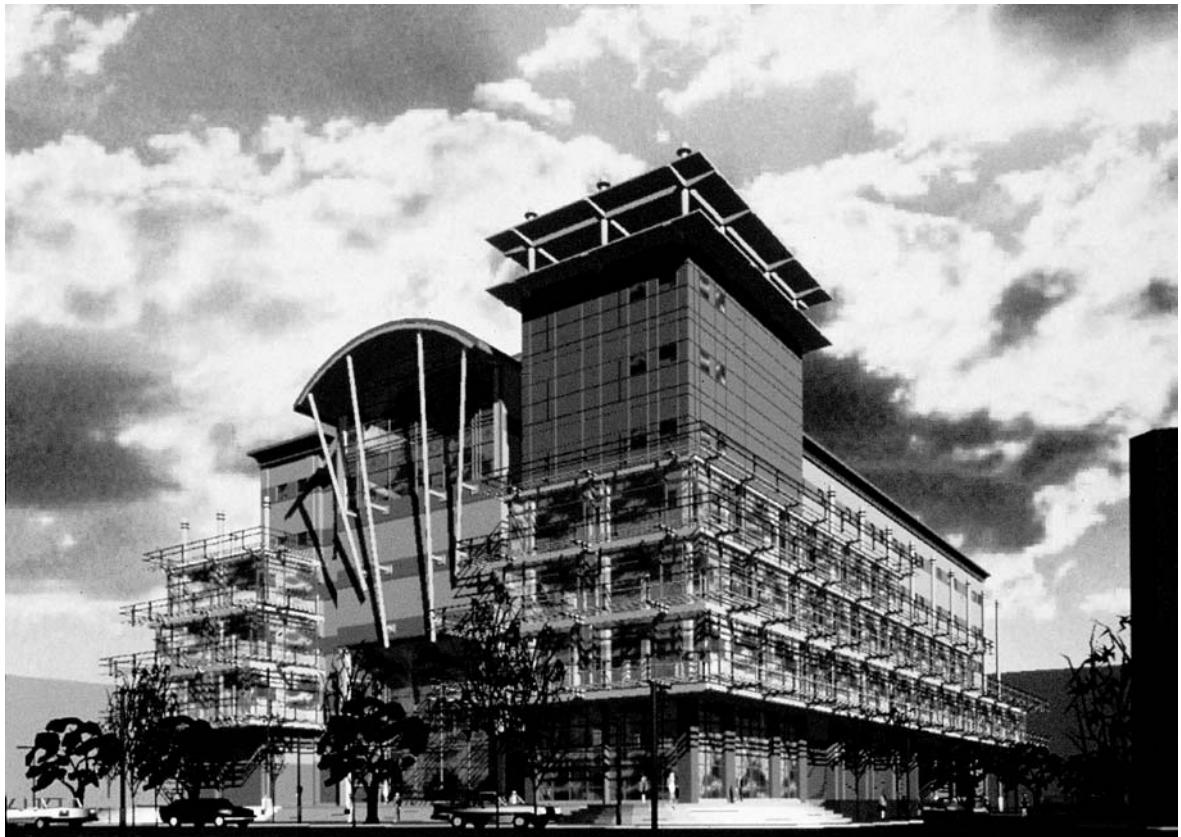
Hay disponibles diversos algoritmos para aplicaciones de animación en Glassner (1990), Arvo (1991), Kirk (1992), Gascuel (1993), Snyder, Woodbury, Fleischer, Currin y Barr (1993) y Paeth (1995). Para ver una explicación de las técnicas de animación en OpenGL, consulte Woo, Neider, Davis y Shreiner (1999).

## EJERCICIOS

---

- 13.1 Diseñe un guión y los fotogramas clave correspondientes para una animación de una figura articulada simple, como en la Figura 13.19.
- 13.2 Escriba un programa para generar los fotogramas intermedios para los fotogramas clave especificados en el Ejercicio 13.1 utilizando interpolación lineal.
- 13.3 Expanda la secuencia de animación del Ejercicio 13.1 para que incluya dos o más objetos móviles.
- 13.4 Escriba un programa para generar los fotogramas intermedios para los fotogramas clave del Ejercicio 13.3 utilizando interpolación lineal.
- 13.5 Escriba un programa de morfismo para transformar cualquier polígono en otro polígono especificado, utilizando cinco fotogramas intermedios.
- 13.6 Escriba un programa de morfismo para transformar una esfera en un poliedro especificado, utilizando cinco fotogramas intermedios.
- 13.7 Defina una especificación de una animación que incluya aceleraciones y que implemente la Ecuación 13.7.
- 13.8 Defina la especificación de una animación que incluya tanto aceleraciones como deceleraciones, implementando los cálculos de espaciado de los fotogramas intermedios dados en las Ecuaciones 13.7 y 13.8.
- 13.9 Defina la especificación de una animación que implemente los cálculos de aceleración-deceleración de la Ecuación 13.9.
- 13.10 Escriba un programa para simular los movimientos lineales bidimensionales de un círculo relleno dentro de un área rectangular especificada. Hay que dar al círculo una posición y una velocidad iniciales y el círculo debe rebotar en las paredes, siendo el ángulo de reflexión igual al ángulo de incidencia.
- 13.11 Convierta el programa del ejercicio anterior en un juego de frontón, sustituyendo un lado del rectángulo por un lado del rectángulo por un corto segmento de línea que pueda moverse adelante y atrás a lo largo de dicho lado del rectángulo. El movimiento interactivo del segmento de línea simula una raqueta que puede colocarse para evitar que escape la bola. El juego terminará cuando el círculo escape del interior del rectángulo. Los parámetros iniciales de entrada incluyen la posición del círculo, la dirección y la velocidad. La puntuación del juego puede incluir el número de veces que la raqueta golpea a la bola.
- 13.12 Modifique el juego de frontón del ejercicio anterior para variar la velocidad de la bola. Después de un corto intervalo fijo, como por ejemplo cinco rebotes, la velocidad de la bola puede incrementarse.
- 13.13 Modifique el ejemplo de la bola bidimensional dentro de un rectángulo, para convertirlo en una esfera que se mueva tridimensionalmente en el interior de un paralelepípedo. Pueden especificarse parámetros de visualización interactivos para ver el movimiento desde distintas direcciones.
- 13.14 Escriba un programa para implementar la simulación de una bola que rebote utilizando la Ecuación 13.10.
- 13.15 Expanda el programa del ejercicio anterior para incluir efectos de comprensión y expansión.
- 13.16 Escriba un programa para implementar el movimiento de una bola que rebote utilizando ecuaciones dinámicas. El movimiento de la bola deberá estar gobernado por una fuerza gravitatoria dirigida hacia abajo y una fuerza de fricción con el plano de tierra. Inicialmente, se proyecta la bola hacia el espacio con un vector de velocidad dado.
- 13.17 Escriba un programa para implementar especificaciones de movimiento dinámico. Especifique una escena con dos objetos o más, con unos parámetros de movimiento iniciales y con unas fuerzas especificadas. Después, genere la animación resolviendo las ecuaciones de fuerza. (Por ejemplo, los objetos podrían ser la Tierra, la Luna y el Sol, con fuerzas gravitatorias atractivas que sean proporcionales a la masa e inversamente proporcionales al cuadrado de la distancia).
- 13.18 Modifique el programa del hexágono giratorio para permitir al usuario seleccionar interactivamente el objeto que hay que girar, a partir de una lista de opciones de menú.
- 13.19 Modifique el programa del hexágono giratorio para que la rotación sea alrededor de una trayectoria elíptica.
- 13.20 Modifique el programa del hexágono giratorio para permitir una variación interactiva de la velocidad de rotación.

# Modelado jerárquico



Una escena infográfica que contiene un complejo de edificios modelado jerárquicamente.  
(Cortesía de *Silicon Graphics, Inc.*)

- |  |   |
|--|---|
| <b>14.1</b> Conceptos básicos de modelado            | <b>14.4</b> Modelado jerárquico mediante listas de visualización OpenGL |
| <b>14.2</b> Paquetes de modelado                     | <b>14.5</b> Resumen   |
| <b>14.3</b> Métodos generales de modelado jerárquico |   |

Al definir un sistema de objeto complejo, lo más fácil suele ser especificar primero las subpartes y luego describir cómo encajan estas subpartes para formar el objeto o sistema global. Por ejemplo, una bicicleta puede describirse en términos de un chasis, unas ruedas, unos pedales, un asiento, etc., junto con las reglas para posicionar estos componentes con el fin de formar la bicicleta. Una descripción jerárquica de este tipo puede proporcionarse en forma de estructura de árbol, que estará compuesta de las subpartes en los nodos del árbol y de las reglas de construcción como las ramas del árbol.

Los sistemas de arquitectura e ingeniería, como por ejemplo los planos de edificios, los diseños de automóviles, los circuitos electrónicos y los electrodomésticos, se desarrollan hoy en día utilizando siempre paquetes de diseño asistido por computadora. Asimismo, se usan métodos de diseño gráfico para representar sistemas económicos, financieros, organizativos, científicos, sociales y medioambientales. A menudo, se construyen simulaciones para estudiar el comportamiento de un sistema en diversas condiciones, y el resultado de la simulación puede servir como herramienta de formación o como base para tomar decisiones acerca del sistema. Los paquetes de diseño proporcionan, generalmente, rutinas para crear y gestionar modelos jerárquicos y algunos paquetes también contienen formas predefinidas, como por ejemplo ruedas, puertas, ejes o componentes de circuitos eléctricos.

## 14.1 CONCEPTOS BÁSIDOS DE MODELADO

---

La creación y manipulación de una representación de un sistema se denomina **modelado**. Cualquier representación se denominará **modelo del sistema** y esos modelos pueden definirse de manera gráfica o puramente descriptiva, como por ejemplo mediante un conjunto de ecuaciones que describan las relaciones entre los parámetros del sistema. Los modelos gráficos se suelen denominar **modelos geométricos**, porque las partes componentes del sistema se representan mediante entidades geométricas tales como segmentos de línea recta, polígonos, poliedros, cilindros o esferas. Puesto que lo único que nos interesa aquí son las aplicaciones gráficas, utilizaremos el término modelo para referirnos a una representación geométrica de un sistema generada por computadora.

### Representaciones de los sistemas

La Figura 14.1 muestra una representación gráfica de un circuito lógico, donde se ilustran las características comunes a muchos modelos de sistemas. Las partes componentes del sistema se muestran como estructuras geométricas, denominadas **símbolos**, y las relaciones entre los símbolos se representan en este ejemplo mediante una red de líneas de conexión. Se utilizan tres símbolos estándar para representar las puertas lógicas correspondientes a las operaciones booleanas: *and*, *or* y *not*. Las líneas de conexión definen las relaciones en términos del flujo de entrada y de salida (de izquierda a derecha) a través de los componentes del sistema. Uno de los símbolos, la puerta *and*, se muestra en dos posiciones diferentes dentro del circuito lógico. El posicionamiento repetido de unos cuantos símbolos básicos es un método común para construir modelos complejos. Cada una de estas apariciones de un símbolo dentro de un modelo se denomina **instancia** de

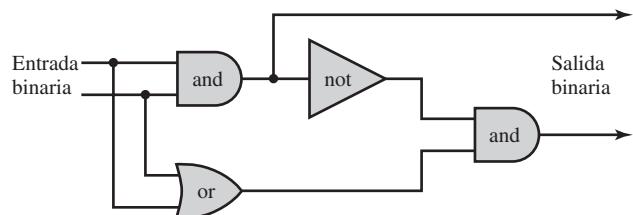


FIGURA 14.1. Modelo de un circuito lógico.

dicho símbolo. Tenemos una instancia para los símbolos *or* y *not* en la Figura 14.1 y dos instancias para el símbolo *and*.

En muchos casos, los símbolos gráficos concretos elegidos para representar las partes de un sistema están dictados por la descripción del propio sistema. Para los modelos de circuitos, se utilizan símbolos lógicos o eléctricos estándar, pero con modelos que representen conceptos abstractos, como por ejemplo sistemas políticos, financieros o económicos, podemos utilizar como símbolos cualquier patrón geométrico que nos resulte conveniente.

La información que describe un modelo suele proporcionarse como una combinación de datos geométricos y no geométricos. La información geométrica incluye las coordenadas para ubicar las partes componentes, las primitivas de salida y las funciones de atributo que definen la estructura de los componentes, así como datos para construir las conexiones entre esos componentes. La información no geométrica incluye las etiquetas de texto, los algoritmos que describen las características de operación del modelo y las reglas para determinar las relaciones o conexiones entre las partes componentes, si es que éstas no están especificadas como datos geométricos.

Hay dos métodos para especificar la información necesaria para construir y manipular un modelo. Uno de los métodos consiste en almacenar la información en una estructura de datos, como por ejemplo una tabla o una lista enlazada. El otro método se basa en especificar la información mediante procedimientos. En general, una especificación de un modelo contendrá tanto estructuras de datos como procedimientos, aunque algunos modelos están definidos completamente mediante estructuras de datos y otros utilizan sólo especificaciones procedimentales. Una aplicación para realizar el modelado de objetos sólidos puede que utilice principalmente información extraída de algún tipo de estructura de datos con el fin de definir las coordenadas, empleándose muy pocos procedimientos. Un modelo climático, por el contrario, puede que necesite principalmente procedimientos para poder calcular los gráficos de temperatura y las variaciones de presión.

Como ejemplo de utilización de combinaciones de estructuras de datos y procedimientos, vamos a considerar algunas especificaciones alternativas para el modelo del circuito lógico de la Figura 14.1. Un método consiste en definir los componentes lógicos en una tabla de datos (Tabla 14.1), empleando una serie de procedimientos de procesamiento para especificar cómo deben realizarse las conexiones de red y cómo opera el circuito. Los datos geométricos contenidos en esta tabla incluyen las coordenadas de parámetros necesarios para dibujar y posicionar las puertas. Estos símbolos pueden dibujarse como formas poligonales o pueden formarse mediante combinación de segmentos de línea recta y arcos elípticos. Las etiquetas para cada una de las puertas componentes también se han incluido en la tabla, aunque podrían omitirse si los símbolos se mostraran mediante formas comúnmente reconocibles. Después de esto, utilizaríamos procedimientos para mostrar las puertas y construir las líneas de conexión, basándonos en las coordenadas de las puertas y en un orden especificado para interconectarlas. Se emplearía también otro procedimiento adicional para producir la salida del circuito (valores binarios) a partir de cualquier entrada dada. Este procedimiento podría definirse de modo que sólo mostrara la salida final, o bien se le podría diseñar para que mostrara también los valores de salida intermedios, con el fin de ilustrar el funcionamiento interno del circuito.

Alternativamente, podríamos especificar la información gráfica correspondiente al modelo de circuito mediante estructuras de datos. Las líneas de conexión, así como las puertas, podrían entonces definirse en una tabla de datos que enumerara explícitamente los extremos de cada una de las líneas del circuito. Con esto, podríamos emplear un único procedimiento para visualizar el circuito y calcular la salida. En el otro extremo,

**TABLA 14.1.** TABLA DE DATOS QUE DEFINE LA ESTRUCTURA Y POSICIÓN DE CADA PUERTA EN EL CIRCUITO DE LA FIGURA 14.1

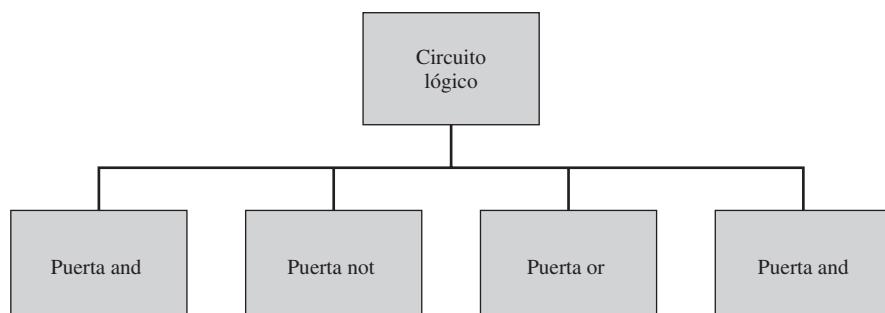
Código del símbolo	Descripción geométrica	Etiqueta identificativa
Puerta 1	(Coordenadas y otros parámetros)	and
Puerta 2	:	or
Puerta 3	:	not
Puerta 4	:	and

sería también posible definir completamente el modelo mediante procedimientos, sin utilizar ninguna estructura de datos externa.

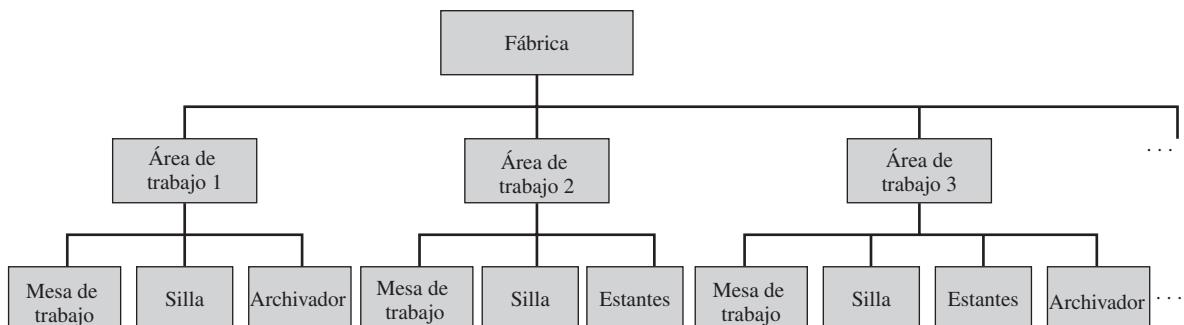
### Jerarquías de símbolos

Muchos modelos pueden organizarse como una jerarquía de símbolos. Los elementos básicos del modelo se definen como formas geométricas simples que resulten apropiadas para el tipo de modelo que se esté considerando. Estos símbolos básicos pueden usarse para formar objetos compuestos, algunas veces denominados **módulos**, que a su vez pueden agruparse para formar objetos de nivel superior, etc., para los diversos componentes del modelo. En el caso más simple, podemos describir un modelo mediante una jerarquía de un único nivel formada por las partes componentes, como en la Figura 14.2. Para este ejemplo de circuito, asumimos que las puertas se colocan y se conectan unas a otras con líneas rectas, de acuerdo con una serie de reglas de conexión especificadas con cada descripción de puertas. Los símbolos básicos en esta descripción jerárquica son las puertas lógicas. Aunque las propias puertas podrían describirse como jerarquías (formadas por líneas rectas, arcos elípticos y textos) dicha descripción no resultaría cómoda para construir circuitos lógicos, ya que en este tipo de circuitos los bloques componentes más simples son las puertas. En una aplicación en la que estuviéramos interesados en diseñar diversas formas geométricas, los símbolos básicos podrían definirse mediante segmentos de línea recta y arcos.

En la Figura 14.3 se muestra un ejemplo de una jerarquía de símbolos de dos niveles. Aquí, el plano de una fábrica se construye mediante una disposición de áreas de trabajo. Cada área de trabajo está equipada con un conjunto de muebles. Los símbolos básicos son los muebles: mesa de trabajo, silla, estantes, archivador, etc. Los objetos de nivel superior son las áreas de trabajo, que se componen mediante diferentes organizaciones de los muebles. Cada instancia de un símbolo básico se define especificando su posición, su tamaño y su orientación dentro de cada área de trabajo. Las posiciones se especifican mediante las coordenadas dentro del área de trabajo y las orientaciones se especifican como rotaciones que determinan hacia qué dirección apuntan los símbolos. En el primer nivel debajo del nodo raíz del árbol correspondiente a la fábrica, cada área de



**FIGURA 14.2.** Una descripción jerárquica de un solo nivel para un circuito formado mediante puertas lógicas.



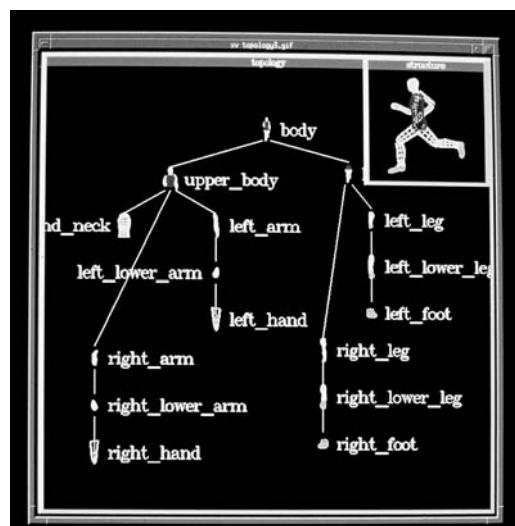
**FIGURA 14.3.** Una descripción jerárquica en dos niveles para el plano de una fábrica.

trabajo se define especificando su posición, su tamaño y su orientación dentro del plano de la fábrica. La frontera de cada área de trabajo puede estar definida mediante un divisor que encierre el área de trabajo y proporcione una serie de entornos independientes dentro de la fábrica.

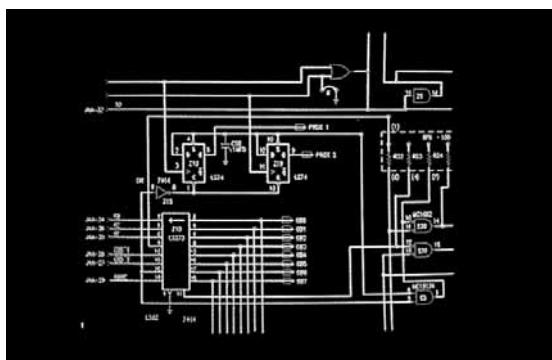
Otras jerarquías de símbolos más complejas pueden formarse mediante el agrupamiento repetido de conjuntos de símbolos en cada nivel superior. La disposición de la fábrica mostrada en la Figura 14.3 podría ampliarse para incluir grupos de símbolos que formaran diferentes habitaciones, diferentes pisos de un edificio, diferentes edificios dentro de un complejo y diferentes complejos en ubicaciones geográficas ampliamente separadas.

## 14.2 PAQUETES DE MODELADO

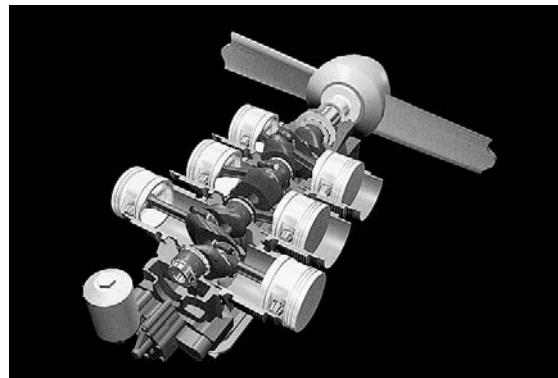
Aunque pueden diseñarse y manipularse modelos de sistemas utilizando un paquete infográfico de tipo general, también hay disponibles sistemas de modelado especializados para facilitar la labor de modelado en aplicaciones concretas. Los sistemas de modelado proporcionan un medio para definir y reordenar las representaciones de los modelos en términos de jerarquías de símbolos, que luego son procesadas por las rutinas gráficas con el fin de poderlas visualizar. Los sistemas gráficos de propósito general no suelen proporcionar rutinas que permitan utilizar aplicaciones extensivas de modelado, pero algunos paquetes gráficos, como GL y PHIGS, sí que incluyen conjuntos integrados de funciones de modelado y gráficas. Un ejemplo de jerar-



**FIGURA 14.4.** Una jerarquía de objetos generada utilizando el paquete PHIGS Toolkit desarrollado en la Universidad de Manchester. El propio árbol de objetos visualizado es una estructura PHIGS. (Cortesía de T. L. J. Howard, J. G. Williams y W. T. Hewitt, Departamento de Computer Science, Universidad de Manchester, Reino Unido.)



**FIGURA 14.5.** Modelado bidimensional utilizando un diseño de circuitos. (Cortesía de Summagraphics.)

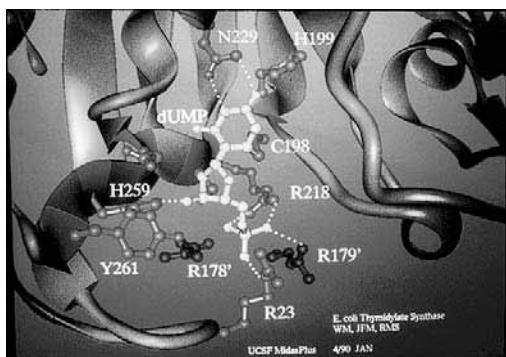


**FIGURA 14.6.** Un modelo CAD que muestra los componentes individuales de un motor, representado por Ted Malone, FTI/3D-Magic. (Cortesía de Silicon Graphics, Inc.)

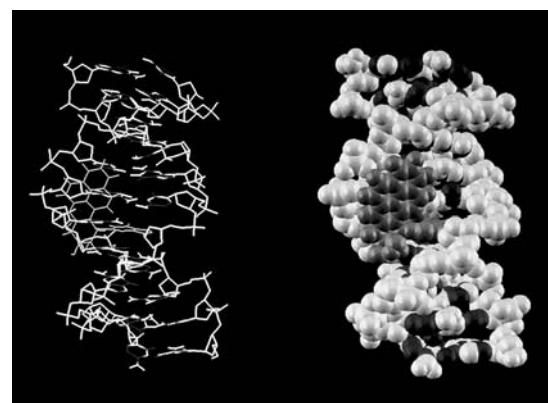
quiero de estructuras PHIGS es el que se muestra en la Figura 14.4. Esta imagen fue generada utilizando el software PHIGS Toolkit, desarrollado en la Universidad de Manchester y que proporciona un editor, ventanas, menús y otras herramientas de interfaz para aplicaciones PHIGS.

Si una biblioteca gráfica no contiene funciones de modelado, podemos utilizar a menudo una interfaz que algún paquete de modelado ofrezca para esas rutinas gráficas. Alternativamente, podemos crear nuestras propias rutinas de modelado utilizando las transformaciones geométricas y otras funciones disponibles en la biblioteca gráfica.

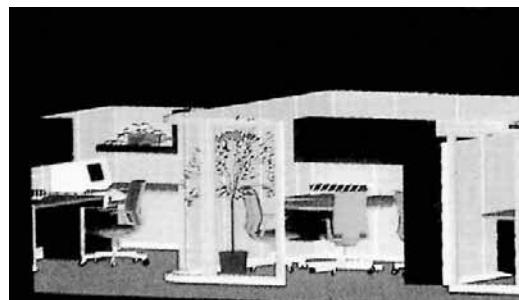
Los paquetes de modelado especializados, como algunos sistemas CAD, se definen y estructuran de acuerdo con el tipo de aplicación para la que el paquete haya sido diseñado. Estos paquetes proporcionan menús con las formas de los símbolos, así como funciones para la aplicación pretendida. Dichos paquetes pueden estar diseñados para modelado bidimensional o tridimensional. La Figura 14.5 muestra un esquema bidimensional generado por un paquete CAD configurado para aplicaciones de diseño de circuitos, mientras que la



**FIGURA 14.7.** Una representación mediante nodos de aristas para residuos de aminoácidos, modelada y representada por Julie Newdollar, UCSF Computer Graphics Lab. (Cortesía de Silicon Graphics, Inc.)



**FIGURA 14.8.** Una mitad de una pareja de imágenes estereoscópicas que muestran un modelo molecular tridimensional del ADN. Datos suministrados por Schlick, NYU y Wilma K. Olson, Rutgers University; visualización realizada por Jerry Greenberg, SDSC. (Cortesía de Stephanie Sides, San Diego Supercomputer Center.)



**FIGURA 14.9.** Una vista tridimensional del plano de una oficina. (Cortesía de Intergraph Corporation.)

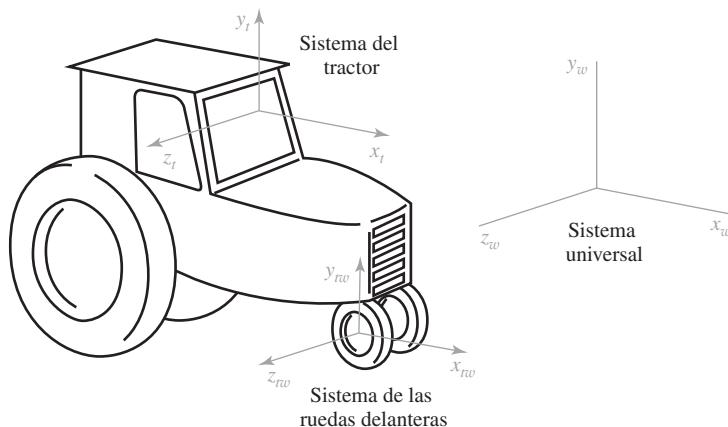
Figura 14.6 ilustra una aplicación CAD tridimensional. En las Figuras 14.7 y 14.8 se muestran ejemplos de modelado molecular, mientras que la Figura 14.9 incluye un modelo tridimensional de una vivienda.

## 14.3 MÉTODOS GENERALES DE MODELADO JERÁRQUICO

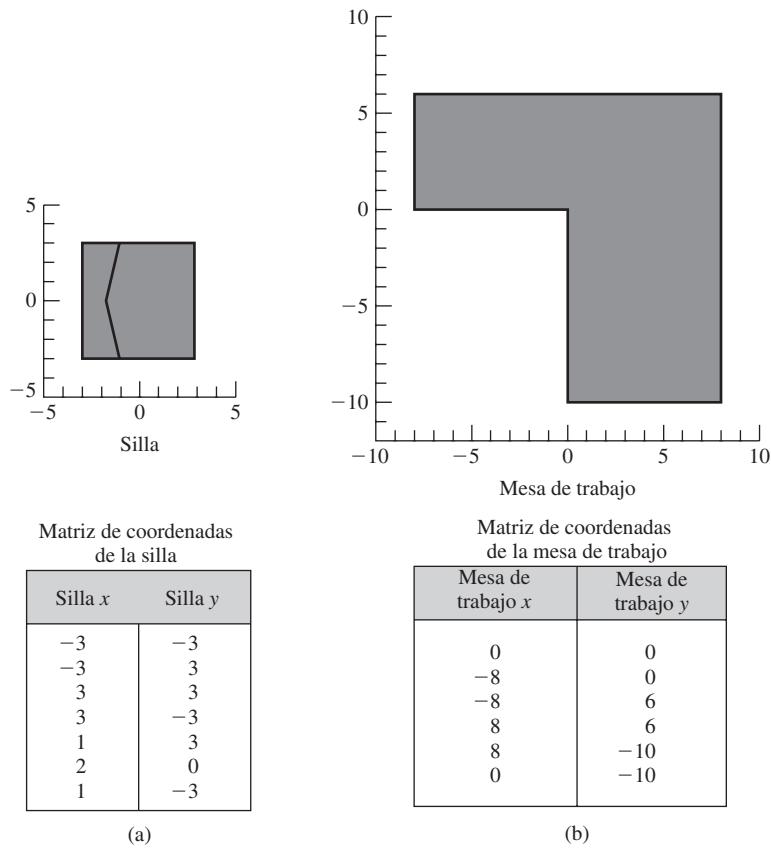
Para crear un modelo jerárquico de un sistema, anidamos las descripciones de sus subpartes con el fin de formar una organización en árbol. A medida que se introduce cada nodo en la jerarquía, se le asigna un conjunto de transformaciones con el fin de posicionarlo apropiadamente dentro del modelo global. Para el diseño de un edificio de oficinas, por ejemplo, las áreas de trabajo y los despachos se forman disponiendo los diversos muebles. Los despachos y áreas de trabajo se colocan entonces en departamentos, etc., hasta llegar a la parte superior de la jerarquía. En la Figura 14.10 se proporciona un ejemplo de utilización de múltiples sistemas de coordenadas y de métodos de modelado jerárquico con objetos tridimensionales. Esta figura ilustra la simulación del movimiento de un tractor. A medida que el tractor se mueve, el sistema de coordenadas del tractor y las ruedas se desplazan dentro del sistema de coordenadas universales. Las ruedas rotan en el sistema de coordenadas de las ruedas y el sistema de las ruedas rota con respecto al sistema del tractor cuando éste efectúa un giro.

### Coordenadas locales

En aplicaciones generales de diseño, los modelos se construyen con instancias (copias transformadas) de las formas geométricas definidas en un conjunto básico de símbolos. Cada instancia se coloca, con la orientación



**FIGURA 14.10.** Sistemas de coordenadas posibles utilizados a la hora de simular el movimiento de un tractor. Una rotación del sistema de las ruedas delanteras hace que el tractor gire. Tanto el marco de referencia de las ruedas como el del tractor se desplazan dentro del sistema de coordenadas universales.



**FIGURA 14.11.** Objetos definidos en coordenadas locales.

apropiada, en el sistema de coordenadas universales correspondiente a la estructura global del modelo. Los diversos objetos gráficos que hay que usar en una aplicación se definen en relación con el sistema de referencia de coordenadas universales, refiriéndonos a este nuevo sistema como *sistema de coordenadas locales* de dicho objeto. Las coordenadas locales se denominan también *coordenadas de modelado* o, en ocasiones, *coordenadas maestras*. La Figura 14.11 ilustra las definiciones en coordenadas locales para dos símbolos que podrían usarse en una aplicación de diseño bidimensional de planos de oficinas.

## Transformaciones de modelado

Par construir un modelo gráfico, aplicamos transformaciones a las definiciones de los símbolos en coordenadas locales, con el fin de producir instancias de los símbolos dentro de la estructura global del modelo. Las transformaciones aplicadas a las definiciones de los símbolos en coordenadas de modelado, con el fin de dar a esos símbolos una posición y orientación concretas dentro de un modelo, se denominan *transformaciones de modelado*. Las transformaciones típicamente disponibles en un paquete de modelado son las de translación, rotación y cambio de escala, aunque algunas aplicaciones pueden también utilizar otros tipos de transformaciones.

## Creación de estructuras jerárquicas

Un primer paso en el modelado jerárquico consiste en construir módulos que estén compuestos de símbolos básicos. Entonces, los propios módulos pueden combinarse para formar módulos de nivel superior, etc.

Definimos cada módulo inicial como una lista de instancias de símbolos, junto con los apropiados parámetros de transformación para cada símbolo. En el siguiente nivel, definimos cada módulo de nivel superior como una lista de símbolos y de instancias de módulo de nivel inferior, junto con sus parámetros de transformación. Este proceso continúa hasta alcanzar la raíz del árbol, que representa el modelo total en coordenadas universales.

En un paquete de modelado, un módulo se crea con una secuencia de comandos del tipo:

```
crearMódulo1
    establecerTransformaciónSímbolo1
    insertarSímbolo1
    establecerTransformaciónSímbolo2
    insertarSímbolo2
    .
    .
    .
    cerrarMódulo1
```

A cada instancia de un símbolo básico se le asigna un conjunto de parámetros de transformación para dicho módulo. De forma similar, los módulos se combinan para formar otros módulos de primer nivel con funciones tales como:

```
crearMódulo6
    establecerTransformaciónMódulo1
    insertarMódulo1
    establecerTransformaciónMódulo2
    insertarMódulo2
    establecerTransformaciónSímbolo5
    insertarSímbolo5
    .
    .
    .
    cerrarMódulo6
```

La función de transformación para cada módulo o símbolo especifica cómo hay que encajar dicho enfoque dentro del módulo de nivel superior. A menudo, se proporcionan opciones para que una determinada matriz de transformación pueda premultiplicar, postmultiplicar o sustituir a la matriz de transformación actual.

Aunque en un paquete de modelado puede haber disponible un conjunto básico de símbolos, puede que ese conjunto de símbolos no contenga las formas que necesitamos para una aplicación concreta. En tal caso, podemos crear formas adicionales dentro de un programa de modelado. Como ejemplo, el siguiente pseudocódigo ilustra la especificación de un modelo simple de bicicleta:

```
crearSímboloRueda
crearSímboloChasis
crearMóduloBicicleta
    especificarTransformaciónChasis
    insertarSímboloChasis
```

```

especificarTransformaciónRuedaDelantera
insertarSímboloRueda

especificarTransformaciónRuedaTrasera
insertarSímboloRueda
cerrarMóduloBicicleta

```

En los sistemas diseñados para modelado jerárquico suele haber disponibles varias otras rutinas de modelado. Suele ser habitual que los módulos puedan visualizarse selectivamente o extraerse temporalmente de la representación de un sistema. Estos permiten al diseñador experimentar con diferentes formas y estructuras de diseño. Asimismo, los módulos pueden resaltarse o desplazarse a través de la imagen durante el proceso de diseño.

## 14.4 MODELADO JERARQUICO MEDIANTE LISTAS DE VISUALIZACIÓN OpenGL

En OpenGL podemos describir objetos complejos utilizando listas de visualización anidadas con el fin de formar un modelo jerárquico. Cada símbolo y módulo del modelo se crea mediante una función `glNewList`, y podemos insertar una lista de visualización dentro de otra utilizando la función `glCallList` dentro de la definición de la lista de nivel superior. Se pueden asociar transformaciones geométricas con cada objeto insertado con el fin de especificar una posición, una orientación o un tamaño dentro de un módulo de nivel superior. Como ejemplo, el siguiente código podría utilizarse para describir una bicicleta que estuviera compuesta simplemente por un chasis y dos ruedas idénticas:

```

glNewList (bicycle, GL_COMPILE);
    glCallList (frame);

    glTranslatef (tx1, ty1, tz1);
    glCallList (wheel);

    glTranslatef (tx2, ty2, tz2);
    glCallList (wheel);
glEndList ( );

```

De forma similar, la lista de visualización `frame` (chasis) podría a su vez estar compuesta de listas de visualización individuales que describieran el manillar, la cadena, los pedales y otros componentes.

## 14.5 RESUMEN

El término «modelo», en aplicaciones infográficas, hace referencia a una representación gráfica de un sistema. Los componentes gráficos de un sistema se representan como símbolos definidos en sistemas de referencia de coordenadas locales, las cuales se denominan también coordenadas maestras o de modelado. Podemos crear un modelo, como por ejemplo un circuito eléctrico, colocando instancias de los símbolos en ubicaciones seleccionadas y con orientaciones prescritas.

Muchos modelos se construyen como jerarquías de símbolos. Podemos construir un modelo jerárquico a nivel de módulos, que estarán compuestos de instancias de símbolos básicos y de otros módulos. Este proce-

so de anidamiento debe continuar hacia abajo, hasta alcanzar símbolos que estén definidos mediante primitivas gráficas de salida y sus correspondientes atributos. A medida que se anida cada símbolo o módulo dentro de un módulo de nivel superior, se especifica una transformación de modelado asociada con el fin de describir su papel concreto dentro de la estructura anidada.

Podemos definir un modelo jerárquico en OpenGL utilizando listas de visualización. La función `glNewList` puede utilizarse para definir la estructura global de un sistema y sus módulos componentes. Los símbolos individuales u otros módulos se insertan dentro de un módulo utilizando la función `glCallList`, precedida mediante un conjunto apropiado de transformaciones que especifiquen la posición, orientación y tamaño del componente insertado.

## REFERENCIAS

---

Puede encontrar ejemplos de aplicaciones de modelado con OpenGL en Woo, Neider, Davis y Shreiner (1999).

## EJERCICIOS

---

- 14.1 Explique las representaciones de modelos que serían apropiadas para diferentes tipos de sistemas que tengan características muy distintas. Asimismo, explique cómo pueden implementarse las representaciones gráficas para cada sistema.
- 14.2 Diseñe un paquete de diseño bidimensional de oficinas. Hay que proporcionar al diseñador un menú de formas de muebles y el diseñador puede poder utilizar un ratón para seleccionar y colocar un objeto en cualquier ubicación dentro de una habitación (jerarquía de un único nivel). Las transformaciones de instancias pueden limitarse a traslaciones y rotaciones.
- 14.3 Amplíe el ejercicio anterior para poder también cambiar la escala de las formas de los muebles.
- 14.4 Diseñe un paquete bidimensional de modelado de oficinas que presente un menú de formas de muebles. Utilice una jerarquía de dos niveles, de modo que los muebles puedan colocarse en diversas áreas de trabajo y que las áreas de trabajo puedan disponerse dentro de un área mayor. Los muebles deben poder colocarse en las áreas de trabajo utilizando únicamente transformaciones de instancia de traslación y rotación.
- 14.5 Amplíe el ejercicio anterior de modo que también pueda cambiarse la escala de los muebles.
- 14.6 Escriba un conjunto de rutinas para crear y visualizar símbolos para el diseño de circuitos lógicos. Como mínimo, el conjunto de símbolos debe incluir las puertas *and*, *or* y *not* mostradas en la Figura 14.1.
- 14.7 Desarrolle un paquete de modelado para el diseño de circuitos lógicos que permita al diseñador colocar símbolos eléctricos dentro de la red de un circuito. Utilice el conjunto de símbolos del ejercicio anterior y emplee únicamente traslaciones para colocar en la red una instancia de las formas contenidas en el menú. Una vez colocado un componente en la red, debe podérselo conectar con otros componentes especificados mediante segmentos de línea recta.
- 14.8 Escriba un conjunto de rutinas para editar módulos que hayan sido creados en un programa de aplicación. Las rutinas deben proporcionar las siguientes modalidades de edición: adición, inserción, sustitución y borrado de módulos.
- 14.9 Dadas las extensiones de coordenadas de todos los objetos visualizados en un modelo, escriba una rutina para borrar cualquier objeto seleccionado.
- 14.10 Escriba procedimientos para visualizar y borrar un módulo especificado dentro de un modelo.
- 14.11 Escriba una rutina que permita extraer selectivamente módulos de la imagen de un modelo o volver a activar su visualización.

- 14.12 Escriba un procedimiento para resaltar de alguna manera un módulo seleccionado. Por ejemplo, el módulo seleccionado podría mostrarse con un color distinto o podría encerrárselo dentro de un contorno rectangular.
- 14.13 Escriba un procedimiento para resaltar un módulo seleccionado en un modelo, haciendo que el módulo parpadee.

# Formatos de archivos gráficos



Una escena generada por computadora creada con golpes de pincel de acuarela simulada.  
(Cortesía de Aydin Controls, División de Aydin Corporation.)

- 15.1** Configuraciones de archivos de imagen
- 15.2** Métodos de reducción de color
- 15.3** Técnicas de compresión de archivos

- 15.4** Composición de la mayoría de los formatos de archivo
- 15.5** Resumen

Cualquier representación ilustrativa almacenada se denomina **archivo gráfico** o **archivo de imagen**. Para sistemas de gráficos digitalizados, un dispositivo de pantalla en color se representa en el búfer de imagen como un conjunto de valores de píxel RGB. Como se pudo ver en la Sección 2.1, el contenido del búfer de imagen, o cualquier sección rectangular del mismo, se llama pixmap. Aunque las imágenes monocromáticas pueden almacenarse en formato bitmap (usando un bit para cada píxel) la mayoría de los gráficos digitalizados se almacenan como pixmaps. En general, cualquier representación digitalizada correspondiente a un gráfico es lo que se denomina **archivo digitalizado** o **rasterizado**. Se han desarrollado muchos formatos para organizar la información en un archivo de imagen de varias maneras, y los archivos digitalizados de color completo pueden ser bastante grandes, por lo que la mayoría de los formatos de archivo aplican algún tipo de compresión para reducir el tamaño del mismo, tanto para archivado definitivo como para su transmisión. Además, el número de valores de color en un archivo de imagen de color completo debe reducirse cuando la imagen va a visualizarse en un sistema con capacidades de color limitadas, o cuando el archivo va a almacenarse en un formato que no soporta 24 bits por píxel. Aquí, se ofrece una breve introducción a los formatos de archivos gráficos y los métodos usados comúnmente para reducir el tamaño tanto del archivo como del número de colores que se van a usar en la visualización de una imagen.

## 15.1 CONFIGURACIONES DE ARCHIVOS DE IMAGEN

---

Los valores de color de píxel en un archivo de imagen digitalizada se almacenan típicamente como enteros no negativos, y el rango de los valores de color depende del número de bits disponibles por posición de píxel. Para una imagen RGB a color completo (24 bits por píxel), el valor para cada componente de color se almacena en un byte, con los valores *R*, *G* y *B* en un rango entre 0 y 255. Un archivo gráfico digitalizado no comprimido compuesto por valores de color RGB, a veces, se denomina **archivo de datos en bruto** o **sin formato** o **archivo digitalizado sin formato**. Otros modelos de color, incluyendo HSV, HSB e YC<sub>r</sub>C<sub>b</sub>, se usan en formatos de archivos comprimidos. El número de bits disponibles por píxel depende del formato.

Los formatos de archivo normalmente incluyen una **cabecera** que ofrece información sobre la estructura del archivo. Para archivos comprimidos, la cabecera debe también contener tablas y otros detalles necesarios para decodificar y visualizar la imagen comprimida. La cabecera puede incluir una amplia variedad de información, tal como el tamaño del archivo (número de líneas de barrido y número de píxeles por línea de barrido) el número de bits o bytes asignados por píxel, el método de compresión usado para reducir el tamaño del archivo, el rango de color para los valores de píxel y el color de fondo de la imagen.

Otra característica de los archivos de imagen digitalizados es el orden de los bytes dentro del archivo. La mayoría de los procesadores de computadora almacenan enteros multibyte con el byte más significativo en primer lugar, aunque algunos procesadores almacenan enteros multibyte con el byte menos significativo en primer lugar. El concepto de **big endian** se usa para indicar que el byte más significativo va el primero, y el concepto de **little endian** hace referencia a que es el byte menos significativo el que va primero.

Algunos formatos de archivo almacenan una imagen en una **representación geométrica**, que es una lista de posiciones de coordenadas y otra información para segmentos de línea recta, áreas de relleno, arcos circu-

lares, curvas de *splines* y otras primitivas. Las representaciones geométricas pueden contener también información sobre los atributos y los parámetros de visualización. Este tipo de representación de imágenes se llama comúnmente *formato vectorial*, incluso cuando no todas las estructuras geométricas se definen mediante segmentos de línea recta. Originalmente, el término de archivo «vectorial» se empleaba para describir una lista de segmentos de línea que se visualizaban en un sistema vectorial (barrido aleatorio). Aunque los sistemas vectoriales han sido sustituidos por sistemas digitalizados, y las descripciones de objetos no lineales han sido añadidas a los archivos «vectoriales», el nombre continua siendo aplicado a cualquier archivo que usa una representación geométrica para una imagen. Los formatos de archivo que soportan tanto representación de imágenes geométricas como digitalizadas son lo que se denominan **formatos híbridos o metaarchivos**.

Las aplicaciones de visualización científica a menudo usan un archivo de imagen que es un conjunto de valores de datos generados a partir de instrumentos de medición o de simulaciones numéricas por computadora. Hay disponibles diversos programas que se usan para ofrecer visualizaciones de datos particulares, tales como visualizaciones de pseudo-color, representaciones de isosuperficies o representaciones de volúmenes.

## 15.2 MÉTODOS DE REDUCCIÓN DE COLOR

---

Se han ideado muchos métodos para reducir el número de colores usados en la representación de una imagen. Los métodos más populares son aquellos que intentan generar muestras de color que se aproximan mucho al conjunto de colores original.

A veces, cuando se habla de métodos de reducción de color se habla de “cuantización”, término que se usa en las áreas de física y matemáticas (como la mecánica cuántica y las teorías de muestreo), para un proceso que genera un conjunto discreto de valores a partir de una distribución continua. En cualquier caso, un archivo digitalizado de imagen no es una distribución continua; contiene un conjunto discreto y finito de valores de color. Por tanto, cualquier método de reducción de color simplemente sustituye un conjunto discreto de colores por un conjunto discreto de colores más pequeño. Además, los procesos de reducción de color, en su uso común, no generan un conjunto de colores tal que cada color del conjunto es un múltiplo de algún valor seleccionado. En otras palabras, la reducción de colores no produce un conjunto cuantizado de colores.

### Reducción uniforme de color

Un método sencillo para reducir colores en un archivo digitalizado consiste en dividir cada uno de los niveles de color *R*, *G* y *B* entre un entero y truncar el resultado. Por ejemplo, si se divide entre 2, se reduce cada uno de los componentes *R*, *G* y *B* en una representación a color completo con 128 niveles. Así, la reducción uniforme de color sustituye grupos de niveles de colores contiguos con un nivel de color reducido, como se ilustra en la Figura 15.1.

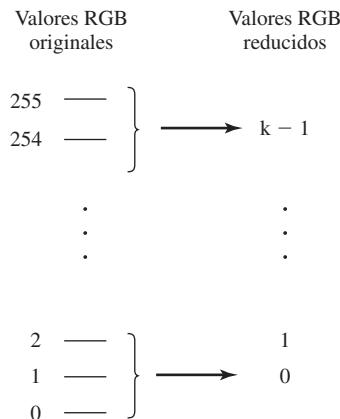
Otro estrategia consiste en sustituir un grupo de valores de píxel con el valor del píxel medio del grupo. O se puede sustituir el grupo de píxeles con el color promedio para el grupo.

En general, no se puede esperar que los 256 valores aparecerán en el archivo de imagen para cada uno de los componentes RGB. Por tanto, se puede aplicar un método de reducción uniforme de color a los niveles de color entre los niveles máximo y mínimo que realmente están en el archivo de imagen.

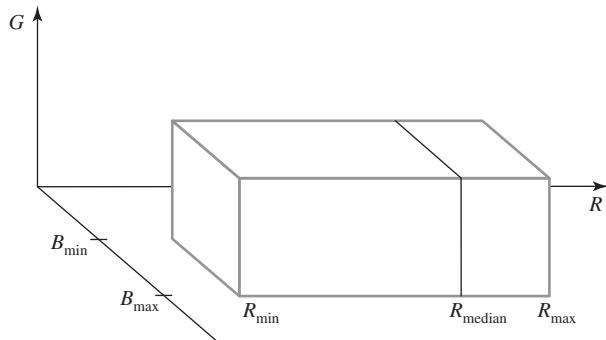
Se pueden también aplicar diferentes criterios de reducción a los distintos componentes RGB. Por ejemplo, se podría reducir una imagen de color completo, de tal forma que las componentes de los colores rojo y verde se representasen con 3 bits cada una (8 niveles) y la componente azul se representase con 2 bits (4 niveles).

### Reducción de color por popularidad

Otra forma de hacer una reducción de color consiste en mantener sólo los valores de color que aparecen más frecuentemente en la representación de una imagen. Se puede procesar primero la entrada del archivo de ima



**FIGURA 15.1.** Reducción uniforme de color de los valores RGB de una imagen de color completo a  $k$  niveles.



**FIGURA 15.2.** Bloque de colores de imagen y división de este bloque en la posición de la componente roja media.

gen para reducir la representación de bits para las componentes individuales RGB. Luego, se explora este conjunto modificado de colores para producir un recuento, o un histograma, de la frecuencia de aparición de cada componente de color RGB. Para producir un archivo de color reducido con  $k$  colores, se selecciona la  $k$  de los colores que aparecen más frecuentemente en el archivo de imagen.

### Reducción de color de corte medio

En este algoritmo, se subdivide el espacio de color para el archivo de imagen en  $k$  subregiones y se calcula el color medio para cada una de las subregiones. Para formar las subregiones, primero se determinan los valores mínimo y máximo para cada componente RGB:  $R_{\min}$ ,  $R_{\max}$ ,  $G_{\min}$ ,  $G_{\max}$ ,  $B_{\min}$  y  $B_{\max}$ . Estos valores proporcionan los saltos en el bloque de colores dentro del cubo de color RGB que está presente en la imagen. Para el mayor de estos tres intervalos, se determina el valor medio y se usa este valor para formar dos bloques de color más pequeños. Por ejemplo, si el componente rojo tiene el rango más largo, se calcula el valor  $R_{\text{median}}$  de tal forma que la mitad de los colores de píxel estén por encima de este valor y la mitad por debajo. Después se recorta la imagen de color dentro de dos subbloques en la posición  $R_{\text{median}}$ , como se muestra en la Figura 15.2. Cada uno de los dos subbloques de color son procesados usando el mismo procedimiento de subdivisión. Este proceso continua hasta que se tiene subdividido el bloque de color de la imagen original en  $k$  subbloques. En cada paso, se puede aplicar el procedimiento de subdivisión al subbloque mayor. Un color medio con la precisión deseada se calcula para cada subbloque, y todos los colores de la imagen dentro de un subbloque se sustituyen con el color promedio del subbloque.

## 15.3 TÉCNICAS DE COMPRESIÓN DE ARCHIVOS

Hay disponible una amplia variedad de técnicas de compresión que permiten reducir el número de bytes en un archivo de imagen, pero la eficacia de un método de compresión particular depende del tipo de imagen. Los métodos sencillos que necesitan patrones en el archivo de imagen son más efectivos con diseños geométricos que contienen amplias áreas de colores sencillos, mientras que los esquemas de compresión complejos producen mejores resultados con gráficos por computadora fotorrealistas y fotografías digitalizadas. La técnica general empleada para reducir el tamaño de los archivos gráficos consiste en sustituir los valores de color con una codificación que ocupe menos bytes que en el archivo original. Además, se incorporan códigos a los archivos comprimidos para indicar cosas como el final de una línea de barrido y el final del archivo de imagen.

Algunos algoritmos de compresión implican operaciones en punto flotante, que pueden introducir errores de redondeo. Además, algunos métodos usan aproximaciones que también modifican los colores de la imagen. Como resultado, un archivo que ha sido decodificado a partir de un archivo comprimido a menudo contiene valores de color que no son exactamente los mismos que había en la imagen original. Por ejemplo, un color entero RGB que se especifica como (247, 108, 175) en un archivo de imagen de entrada puede convertirse en el color (242, 111, 177) después de decodificar el archivo comprimido. Pero tales cambios de color, normalmente, son tolerables porque el ojo humano no es sensible a pequeñas diferencias de color.

Los métodos de reducción de archivos que no cambian los valores en un archivo de imagen emplean técnicas **sin pérdida de compresión**, y aquéllos que dan lugar a cambios de color usan técnicas **con pérdidas de compresión**. En la mayoría de los casos, los métodos con pérdidas de compresión dan lugar a una tasa de compresión mucho mayor para un archivo, siendo la tasa de compresión el número de bytes del archivo original dividido entre el número de bytes del archivo comprimido.

## Codificación de longitud de recorrido

Este esquema de compresión sencillamente busca en el archivo de imagen los valores repetidos contiguos. Se forma un archivo reducido almacenando cada secuencia de valores repetidos como un único valor de archivo junto con el número de repeticiones. Por ejemplo, si el valor 125 aparece 8 veces en sucesión a lo largo de una línea de barrido, se almacenan los dos valores, 8 y 125, en el archivo comprimido. Esto reduce los ocho bytes originales de almacenamiento a dos bytes. Para imágenes con grandes áreas de un único color, este esquema de codificación funciona bien. Pero para las imágenes tales como fotografías digitalizadas que tienen frecuentes cambios de color y pocas repeticiones de valores consecutivos, muchos valores de color se almacenarían con un factor de repetición de 1.

Se han desarrollado variantes para mejorar la eficiencia del algoritmo básico de codificación de longitud de recorrido. Por ejemplo, se podría usar un factor de repetición negativo para indicar una secuencia de valores no repetitivos, en lugar de simplemente almacenar un factor de repetición de 1 con cada uno de los valores de la secuencia no repetitiva. Por ejemplo, la siguiente lista de valores:

```
{ 20, 20, 20, 20, 99, 68, 31, 40, 40, 40, 40, 40, 40, . . . }
```

podría codificarse como,

```
{ 4, 20, 3, 99, 68, 31, 8, 40, . . . }
```

lo que indica que el valor 20 ocurre 4 veces, seguido de 3 valores no repetitivos 99, 68 y 31, el cual va seguido por 8 ocurrencias del valor 40. En este ejemplo de codificación, los primeros 15 bytes del archivo de entrada se comprimen en 8 bytes.

## Codificación LZW

Desarrollado por Lempel, Ziv y Welch, el método LZW es una modificación de los anteriores LZ, LZ77 y LZ78 algoritmos de reconocimiento de patrones. En el esquema LZW, los patrones repetidos en un archivo de imagen son sustituidos por un código. Por ejemplo, la siguiente lista de 12 valores contiene dos ocurrencias para cada patrón {128, 96} y {200, 30, 10}:

```
{ 128, 96, 200, 30, 10, 128, 96, 50, 240, 200, 30, 10, . . . }
```

Se pueden sustituir estos dos patrones con los códigos  $c_1$  y  $c_2$  y, al patrón restante {50, 240} puede asignársele el código  $c_3$ . Esto reduce los primeros doce valores de la lista de entrada a los siguientes 5 bytes:

```
{  $c_1$ ,  $c_2$ ,  $c_1$ ,  $c_3$ ,  $c_2$ , . . . }
```

Alternativamente, cualquier secuencia de valores no repetitivos, tales como {50, 240} podrían almacenarse en el archivo comprimido sin asignar un código a la secuencia.

Básicamente, el algoritmo LZW busca secuencias repetidas y construye una tabla de tales secuencias junto con sus códigos asignados. Así, este esquema de codificación se denomina *algoritmo sustitucional* o *algoritmo basado en diccionario*. El archivo comprimido se decodifica entonces con la tabla de códigos.

## Otros métodos de compresión mediante reconocimiento de patrones

Se pueden usar esquemas de reconocimiento de patrones para localizar repeticiones de combinaciones de blanco y negro o color RGB a lo largo de todo un archivo de imagen. Se pueden detectar líneas de barrido duplicadas y otros patrones y codificarse para reducir más aún el tamaño de los archivos de imagen. Además, se han aplicado métodos de fractales para obtener pequeños conjuntos codificados autosimilares de valores de color.

## Codificación Huffman

Con el método Huffman la compresión del archivo se logra usando un código de longitud variable para los valores de un archivo de imagen. El método de codificación Huffman asigna el código más corto al valor del archivo que tiene una ocurrencia más frecuente, y el código más largo se asigna al valor que ocurre con menor frecuencia.

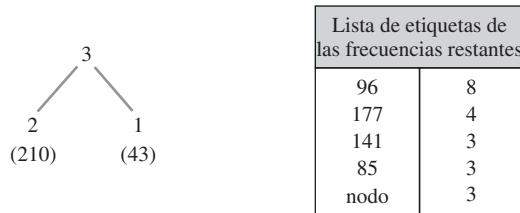
La idea básica en el algoritmo Huffman es la misma que en el código Morse, el cual asigna códigos de caracteres de longitud variable a letras del alfabeto. A las letras con mayor frecuencia en el esquema Morse, se les asigna códigos de un carácter, y las letras con frecuencia más baja tienen asignado códigos de cuatro caracteres. Por ejemplo, la letra E se codifica con un «punto» (.), la letra T se codifica como «guión» (-) y la letra Q se codifica como una secuencia de cuatro caracteres con un punto y tres guiones (---.). En cambio, en lugar de usar códigos de caracteres, el código Huffman asigna códigos de bits de longitud variable a los valores de un archivo de imagen, que ofrecen mayores tasas de compresión.

El primer paso en el algoritmo Huffman consiste en contar el número de ocurrencias para cada valor contenido en el archivo de imagen de entrada. Luego, los códigos de bits se asignan a los valores de acuerdo con la frecuencia contada. Un método para asignar los códigos de bits de longitud variable es construir un árbol binario con los valores de alta frecuencia cerca de la raíz del árbol y los valores con frecuencias más bajas como nodos hoja. Comenzando con los valores de baja frecuencia, se pueden crear subárboles de abajo a arriba. A cada nodo rama de un subárbol se le asigna una etiqueta numérica, que es la suma de las cuentas de frecuencia o los nodos etiqueta de sus dos hijos. Cuando el árbol está completo, todos los subárboles de la izquierda se etiquetan con el valor binario 0 y todos los árboles de la derecha se etiquetan con el valor binario 1. El código de bit para cada valor del archivo se forma mediante la concatenación de las etiquetas de bit de las ramas, desde la raíz del árbol hacia abajo hasta el nodo posición de aquel valor de archivo del árbol.

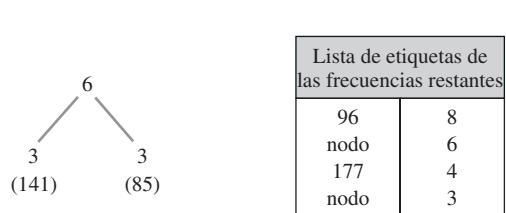
Para ilustrar los pasos generales de construcción de un árbol se usa el conjunto de seis valores de la Tabla 15.1. Este conjunto representa una pequeña imagen de ejemplo que contiene 21 elementos, en la que el valor 96 aparece 8 veces, el valor 177 aparece 4 veces, y así sucesivamente para los otros cuatro valores del archivo.

**TABLA 15.1. RECUENTO DE FRECUENCIAS PARA LOS VALORES DE UN ARCHIVO PEQUEÑO DE EJEMPLO**

Valor de archivo	Recuento de frecuencias
96	8
177	4
141	3
85	3
210	2
43	1
Total de valores en el archivo	21

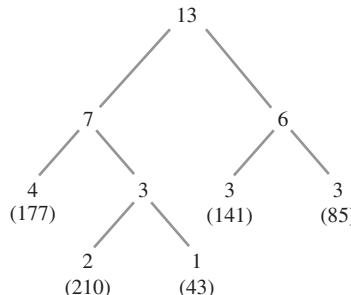
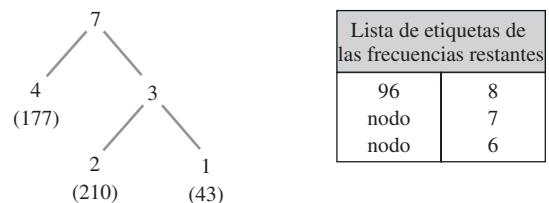


**FIGURA 15.3.** Formación de un subárbol Huffman usando los valores de archivo 210 y 43.

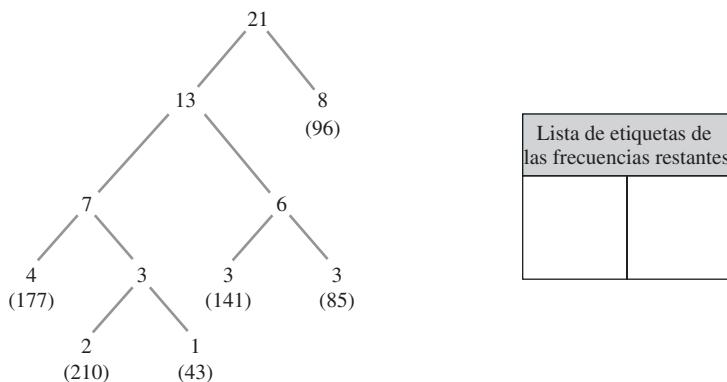


**FIGURA 15.4.** Formación de un subárbol Huffman usando los valores de archivo 141 y 85.

**FIGURA 15.5.** Formación de un subárbol Huffman usando los valores de archivo 177 y un subárbol creado previamente.

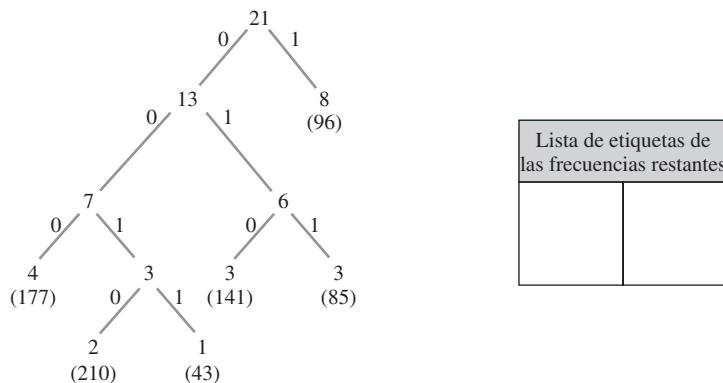


**FIGURA 15.6.** Formación de un subárbol Huffman uniendo dos subárboles previamente creados.



**FIGURA 15.7.** Árbol binario Huffman completo para los valores de archivo de la Tabla 15.1.

Los valores 210 y 43 de esta tabla tienen el recuento de frecuencias más bajo, por lo que se usan estos dos valores para formar el primer subárbol (Figura 15.3). A la raíz de este subárbol se le asigna una etiqueta nodo que es igual a la suma del número de ocurrencias de sus dos hijos:  $3 = 2 + 1$ . Se borran esos dos valores de archivo (210 y 43) de la lista activa de tal forma que el siguiente recuento de frecuencia menor es 3. Pero justo



**FIGURA 15.8.** Árbol binario Huffman completo con el etiquetado de las ramas.

se ha creado un subárbol que también tiene la etiqueta de nodo 3. Por tanto, se puede formar el siguiente subárbol usando cualquier par de los tres elementos que tienen la etiqueta 3. Se eligen los dos valores de archivo para formar el subárbol mostrado en la Figura 15.4, y se borran los valores 141 y 85 de la lista activa. El siguiente subárbol se construye con el valor de archivo 177, el cual presenta una frecuencia de 4, y el subárbol cuya rama tiene la etiqueta 3 (Figura 15.5). Se borra el valor de archivo 177 y el nodo del árbol con la etiqueta 3 de la lista activa, y ahora los dos «recuentos» más bajos de la lista representan subárboles. Estos dos subárboles se combinan para producir el nuevo subárbol mostrado en Figura 15.6. Finalmente, se completa la construcción del árbol binario (Figura 15.7) uniendo el valor de archivo 96 con el último subárbol creado. El valor asignado a la raíz del árbol es el recuento total (21) de todos los valores del archivo de imagen.

Ahora que se tienen todos los valores del archivo en el árbol binario, se pueden etiquetar las ramas de la izquierda del árbol con el valor binario 0 y las ramas de la derecha con el valor binario 1, como se muestra en la Figura 15.8. Comenzando desde la raíz del árbol, se concatenan las etiquetas de las ramas hacia abajo en cada uno de los nodos hoja. Esto forma el conjunto de códigos de longitud variable para cada uno de los valores de archivo, y después se establece la Tabla 15.2, la cual se almacenará con el archivo comprimido. En este ejemplo, hay un valor de archivo con un código binario de un dígito, tres valores de archivo con un código binario de tres dígitos, y dos valores de archivo con un código binario de cuatro dígitos. Los valores de baja frecuencia tienen códigos más largos, y los valores con frecuencia más alta tienen códigos más cortos.

Una característica importante de los códigos Huffman es que ningún código de bits es un prefijo para ningún otro código de bits. Esto permite decodificar una lista codificada de valores de archivo proporcionando la Tabla 15.3 junto con la Tabla 15.2. Para demostrar el algoritmo de decodificación, se supone que el archivo comprimido contiene la trama de bits {100100100...}. El primer valor de bits en este archivo es 1, así que

**TABLA 15.2. CÓDIGOS HUFFMAN INDEXADOS PARA EL ARCHIVO DE EJEMPLO**

Índice	Valor de archivo	Código binario
1	96	1
2	177	000
3	141	010
4	85	011
5	210	0010
6	43	0011

**TABLA 15.3.** TABLA DE REFERENCIA DE CÓDIGOS DE BIT

<i>Longitud de código de bits</i>	<i>Valor mínimo de código</i>	<i>Valor máximo de código</i>	<i>Primer índice</i>
1	1	1	1
3	000	011	2
4	0010	0011	5

debe representar el valor de archivo 96 porque hay un código de bit 1 y no puede ser un prefijo para ningún otro código. Después, se tiene el valor de bit 0. No hay ningún otro código de bits además del 1 y no hay códigos de dos bits, así que el siguiente código debe ser 001 ó 0010. Verificando la tabla de códigos indexada, se encuentra un valor de archivo de 210 con un código 0010, lo que significa que no puede haber un valor de archivo con el código 001. En este punto, se tienen decodificados los dos primeros valores de archivo, 96 y 210. El siguiente código en la trama de bits debe ser 010 ó 0100. Hay un valor de archivo con el código 010, así que no puede haber un código de cuatro bits con dicho prefijo. Por tanto, el tercer valor de archivo decodificado es 141. Se continua analizando la trama de bits de esta manera hasta que el archivo comprimido haya sido completamente decodificado.

Se pueden usar también otros esquemas para generar y asignar códigos de bits Huffman. Una vez que se tiene el recuento de frecuencias, se podría asignar una longitud de código a cada valor de archivo. Usando la longitud de código y el recuento de la frecuencia, se puede usar un algoritmo de combinación de listas para deducir los códigos de bits específicos. También se puede emplear un conjunto de códigos predefinido para asignar códigos a los valores de archivo, lo que elimina la necesidad de almacenar los códigos con el archivo comprimido.

### Codificación aritmética

En este esquema de compresión, el recuento de frecuencias de un archivo se usa para obtener códigos numéricos para secuencias de valores de archivo. El algoritmo de codificación aritmética primero calcula la fracción del archivo que está ocupada por cada valor. Esto crea un conjunto de subintervalos dentro del intervalo de 0.0 a 1.0. Después, cada fracción de archivo se mapea repetidamente sobre estos subintervalos para establecer intervalos numéricos para las diferentes combinaciones de valores de archivo. Los límites numéricos de estos intervalos se usan para codificar estas combinaciones.

Para ilustrar el método, consideramos un archivo con 80 entradas y sólo tres valores distintos. El recuento de frecuencias y las correspondientes fracciones de archivos para los tres valores se enumeran en la Tabla 15.4. Así, el valor  $V_1$  se asocia con el subintervalo de 0.00 a 0.20 dentro del intervalo unidad, el valor  $V_2$  se asocia con el subintervalo de 0.20 a 0.50, y el valor  $V_3$  se asocia con el subintervalo de 0.50 a 1.00. En otras palabras, el 20 por ciento del intervalo unidad se asocia con  $V_1$ , el 30 por ciento con  $V_2$  y el 50 por ciento con

**TABLA 15.4.** RECUENTO DE FRECUENCIAS Y FRACCIÓN DE OCURRENCIAS PARA LOS VALORES DE UN ARCHIVO PEQUEÑO DE EJEMPLO

<i>Valor del archivo</i>	<i>Recuento de frecuencias</i>	<i>Fracción de archivo</i>	<i>Rango del intervalo unidad</i>
$V_1$	16	0.20	0.00–0.20
$V_2$	24	0.30	0.20–0.50
$V_3$	40	0.50	0.50–1.00
Total	80	1.00	

**TABLA 15.5.** RANGO DEL INTERVALO UNIDAD PARA CADA SECUENCIA DE DOS VALORES QUE COMIENZA CON EL VALOR  $V_3$

Secuencia	Rango del intervalo unidad
$V_3V_1$	0.50–0.60
$V_3V_2$	0.60–0.75
$V_3V_3$	0.75–1.00

$V_3$ . Si ahora mapeamos  $V_1$  sobre el subintervalo  $V_3$ , éste ocupará el 20 por ciento de la mitad superior del intervalo unidad. Este nuevo subintervalo, con un rango de 0.50 a 0.60, representa la secuencia  $V_3V_1$ . Resultados similares se obtienen para los mapeos de  $V_2$  y  $V_3$  sobre el subintervalo  $V_3$ . La Tabla 15.5 enumera los rangos para estas tres secuencias de dos valores. Continuando de esta forma, se pueden mapear los intervalos para las secuencias de dos valores sobre otros subintervalos para obtener las secuencias para combinaciones más largas de los valores de archivo. Los valores límite para los subintervalos se usan entonces para codificar y decodificar las secuencias dentro del archivo.

Pueden emplearse varios algoritmos para determinar las subdivisiones del intervalo unidad y asignar códigos numéricos a combinaciones de valores de archivo. Y el algoritmo de codificación aritmética se implementa típicamente usando números binarios en lugar de valores en punto flotante dentro del intervalo unidad. El archivo comprimido es entonces una secuencia de valores binarios.

### Trasformada discreta del coseno

Una serie de métodos de trasformadas numéricas, incluyendo las transformadas de Fourier y Hadamard, han sido aplicadas a la compresión de archivos, pero la trasformada discreta del coseno es el método más comúnmente usado. Los algoritmos de implementación eficientes para una trasformada discreta del coseno ofrecen una ejecución más rápida y mejor fidelidad de color en una imagen reconstruida con unas tasas de compresión más altas.

Para una lista de  $n$  valores numéricos  $V_k$ , con  $k = 0, 1, \dots, n - 1$ , el método discreto del coseno genera el siguiente conjunto de valores transformados.

$$V'_j = c_j \sum_{k=0}^{n-1} V_k \cos\left[\frac{(2k+1)j\pi}{2n}\right], \quad \text{para } j = 0, 1, \dots, n-1 \quad (15.1)$$

donde,

$$c_j = \begin{cases} \frac{1}{\sqrt{n}}, & \text{para } j = 0 \\ \sqrt{\frac{2}{n}}, & \text{para } j \neq 0 \end{cases}$$

Por tanto, este método de transformación calcula una suma discreta de términos coseno con una frecuencia que se va incrementando y con amplitudes que son proporcionales a los valores de entrada. Excepto por la posibilidad de errores de redondeo, los valores originales se recuperan con la trasformada inversa:

$$V_k = \sum_{j=0}^{n-1} c_j V'_j \cos\left[\frac{(2k+1)j\pi}{2n}\right], \quad \text{para } k = 0, 1, \dots, n-1 \quad (15.2)$$

Muy a menudo, los valores de transformación  $V'_j$  se refieren a los «coeficientes» de las funciones coseno en la ecuación de la transformada inversa. Pero ésta es una terminología incorrecta, ya que los coeficientes de los términos coseno en el sumatorio son los productos  $c_j V'_j$ .

Para ilustrar este método de transformación, consideremos la siguiente lista de 8 valores de entrada:

$$\{215, 209, 211, 207, 192, 148, 88, 63\}$$

Los valores transformados, calculados con dos posiciones decimales, para esta entrada son:

$$\{471.29, 143.81, -67.76, 16.33, 7.42, -4.73, 5.49, 0.05\}$$

En este ejemplo, observamos que las amplitudes de los valores transformados decrecen notablemente, de tal forma que los términos coseno con la frecuencia más alta contribuyen menos a la recuperación de los valores de entrada. Ésta es una característica básica de la transformada discreta del coseno, que permite aproximarse muy estrechamente a los valores originales usando sólo los primeros valores transformados. Por tanto, para obtener un archivo de imagen comprimido, se podrían calcular y almacenar sólo la primera mitad de los valores de transformación. La Tabla 15.6 muestra los resultados de la Ecuación 15.2 cuando se emplean 4, 5 o los 8 valores transformados para recuperar los valores de entrada. Todos los valores calculados en la tabla están redondeados a dos posiciones decimales.

Podemos mejorar la eficiencia de esta técnica de compresión transformando bloques rectangulares de valores de entrada, mejor que transformando conjuntos lineales de valores a través de una sola línea de barriodo. Para un bloque cuadrado de  $n$  por  $n$  valores de entrada, los valores de transformación se calculan como:

$$V'_{lm} = c_{lm} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} V_{jk} \cos\left[\frac{(2k+1)l\pi}{2n}\right] \cos\left[\frac{(2j+1)m\pi}{2n}\right] \quad (15.3)$$

con

$$l, m = 0, 1, \dots, n-1$$

y

$$c_{lm} = \begin{cases} \frac{1}{n}, & \text{si } l = 0 \text{ o } m = 0 \\ \frac{2}{n}, & \text{si } l \neq 0 \text{ y } m \neq 0 \end{cases}$$

Además, el conjunto de  $n$  por  $n$  valores de entrada se recupera usando la transformada inversa:

$$V_{jk} = \sum_{l=0}^{n-1} \sum_{m=0}^{n-1} c_{lm} V'_{jl} \cos\left[\frac{(2j+1)l\pi}{2n}\right] \cos\left[\frac{(2k+1)m\pi}{2n}\right] \quad (15.4)$$

**TABLA 15.6. CÁLCULOS DE LA TRANSFORMACIÓN INVERSA DISCRETA DEL COSENO**

Valores de entrada	215	209	211	207	192	148	88	63
<i>Términos suma</i>	<i>Valores de la transformada inversa discreta del coseno</i>							
4	212.63	211.85	211.53	207.42	188.43	147.65	95.47	58.02
5	215.26	209.23	208.91	210.04	191.06	145.02	92.84	60.64
8	215.00	209.00	211.00	207.00	192.00	148.00	88.00	63.00

donde

$$j, k = 0, 1, \dots, n - 1$$

Esta trasformada y su inversa se implementan típicamente usando grupos de 8 por 8 valores de entrada, de tal forma que los grupos de valores de color a lo largo de 8 líneas de barrido se procesan simultáneamente.

## 15.4 COMPOSICIÓN DE LA MAYORÍA DE FORMATOS DE ARCHIVO

---

Cientos de formatos de archivo han sido desarrollados para representar datos gráficos dentro de diferentes contextos para diferentes sistemas. Los sistemas operativos, por ejemplo, normalmente usan una serie de formatos especialmente diseñados en diversas rutinas de procesamiento del sistema. Existen formatos individuales para aplicaciones específicas, tales como el modelado tridimensional, las animaciones, las interfaces gráficas de usuario, el software de trazado de rayos, las grabaciones de vídeo, el software para visualizaciones científicas, los programas de dibujo, los procesadores de texto, paquetes de hojas de cálculo, comunicaciones por Internet, multidifusión por televisión y transmisión de faxes. Además, los comités de estandarización ISO y ANSI han propuesto distintos formatos y sistemas de compresión de archivos para uso general.

La mayoría de los formatos de archivo están diseñados para acomodar imágenes de color, pero algunos se aplican sólo a bitmaps. Sin embargo, el nombre del formato a menudo es engañoso, ya que el término mapa de bits o bitmap se usa frecuentemente para hacer referencia a imágenes de color (pixmaps). Esta situación es simplemente el resultado del uso continuado de la vieja etiqueta, bitmap, para un archivo digitalizado. Antes del desarrollo de los dispositivos de visualización en color, todas las imágenes digitalizadas se almacenaban como bitmaps (un bit por píxel) representando los patrones de píxel blanco y negro en una imagen. Cuando se desarrollaron las técnicas de color, los archivos pixmap (múltiples bits por píxel) sustituyeron a los bitmaps. Pero muy a menudo, estos archivos se siguen denominando bitmaps. Como resultado, muchos esquemas de codificación de color en uso hoy en día para archivos de imagen se etiquetan como «formatos bitmap» incluso aunque sean realmente formatos pixmap (múltiples bits por píxel). En cualquier caso, la documentación para cada formato puede consultarse para determinar el número de bits que realmente se asignan a cada posición de píxel en el archivo.

Para la mayor parte, los formatos de archivo descritos en esta sección son no estáticos. Los mismos, sufren constantes revisiones y actualizaciones, y a menudo existen muchas variantes para un formato particular.

### JPEG: Joint Photographic Experts Group

En su forma básica, este sistema complejo y ampliamente utilizado, desarrollado por el comité JPEG de la organización internacional de estándares ISO consiste en una larga colección de opciones para compresión de archivos. Más de dos docenas de variantes se dan para la definición de JPEG, así que se puede implementar de un número de maneras diferente, desde un simple algoritmo con pérdidas hasta un método de alta compresión sin pérdidas. Pero la definición básica de JPEG no especifica completamente cómo el archivo de imagen comprimido debería estructurarse para que pueda usarse en distintos sistemas informáticos o por diferentes aplicaciones. Por ejemplo, no hay una organización específica para la información de cabecera y no hay especificación para el modelo de color que debería usarse en el archivo comprimido.

El estándar JPEG define cuatro modos generales para la compresión de archivos, que son los modos sin pérdidas, secuencial, progresivo y jerárquico. En el **modo JPEG sin pérdidas**, un esquema de reconocimiento de patrones se combina tanto con codificación Huffman como con codificación aritmética. Sin embargo, el modo original de JPEG sin pérdidas no es tan eficiente como otros formatos sin pérdidas disponibles, por lo que raramente se implementan. El **modo secuencial JPEG básico** es la versión de JPEG más comúnmente utilizada. Los valores numéricos para los componentes de color en una imagen se almacenan en 8 bits, y el algoritmo de compresión combina la trasformada discreta del coseno con la codificación Huffman o aritmética.

tica. También se ha definido un *modo secuencial extendido* con más opciones que el modo secuencial básico y cuyos componentes de color pueden especificarse usando 16 bits. En el **modo JPEG progresivo**, un archivo de imagen se procesa usando distintas pasadas para que las “capas” de la imagen puedan generarse con distintas resoluciones. Este modo, generalmente llamado *JPEG progresivo*, se está haciendo popular para aplicaciones de Internet, porque una aproximación de tipo borrador de una imagen puede visualizarse rápidamente antes de descargarse la imagen de archivo completa. Otra colección de procedimientos para obtener versiones de una imagen mejoradas incrementalmente está contenida en el **modo JPEG jerárquico**, que divide una imagen en un conjunto de subimágenes. Esto permite seleccionar secciones de una imagen para construirlas progresivamente. Debido a su complejidad, el JPEG jerárquico no es muy usado.

Pueden ofrecerse opciones en implementaciones de JPEG a gran escala para la selección del modo de compresión y los parámetros de compresión, como el número de términos que hay que usar en los cálculos del sumatorio de la transformada discreta inversa del coseno. También, las definiciones de compresión JPEG especifican que tanto la codificación Huffman como la codificación aritmética pueden combinarse con la transformada discreta del coseno. Pero las implementaciones de JPEG nunca usan los algoritmos de codificación aritmética, porque estos algoritmos están patentados y requieren el pago de una licencia.

A pesar de que la especificación de JPEG no define una estructura específica del archivo de imagen comprimido, ahora las implementaciones usan el formato **JPEG File Interchange** (JFIF) propuesto por Eric Hamilton en C-Cube Microsystems y basado en sugerencias de muchos usuarios de JPEG. En este formato, la cabecera del archivo contiene un identificador JFIF único (que es lo que se denomina “firma” del archivo), la versión del JFIF utilizado para configurar el archivo, el tamaño de la imagen (en píxeles por cm o en píxeles por pulgada), la altura y anchura de la vista preliminar opcional RGB del archivo (denominada imagen “miniatura”) y los valores RGB para la vista preliminar opcional de la imagen. Los valores de los píxeles del archivo comprimido se almacenan usando el modelo de color  $YC_rC_b$ , y los componentes de color se almacenan en orden, en primer lugar  $Y$ , en segundo  $C_b$  y, por último,  $C_r$ . Para una imagen en escala de grises, sólo se usa la componente  $Y$ . Otra información del archivo incluye las tablas necesarias para los algoritmos de compresión. Los enteros se almacenan en archivos JPEG usando el formato big-endian.

La codificación secuencial básica JPEG/JFIF de un archivo de imagen consiste normalmente en las siguientes operaciones.

- (1) **Conversión de color:** los valores de color de píxel RGB en un archivo de imagen son convertidos en componentes de color  $YC_rC_b$ .
- (2) **Muestreo de color:** el número de valores de color del archivo puede reducirse usando sólo los valores para los píxeles seleccionados o haciendo la media de las componentes de color para grupos de píxeles adyacentes. Una implementación sencilla para estas operaciones de muestreo debe tomar los valores de color de todos los otros píxeles, todos los píxeles terceros o todos los píxeles cuartos. Usualmente, las componentes de color se muestran en diferentes frecuencias, por lo que los valores de mayor luminancia, las componentes  $Y$ , se seleccionan. Esto permite alcanzar mayores tasas de compresión, ya que se almacenan muy pocos valores diferentes de crominancia, las componentes  $C_r$  y  $C_b$ .
- (3) **Trasformada discreta del coseno:** a continuación, los grupos de 8 por 8 valores de píxeles de color se convierten en valores de la trasformada discreta del coseno usando la Ecuación 15.3.
- (4) **Reducción de los valores transformados:** para una compresión mayor del archivo de imagen codificado, se almacena un conjunto reducido de valores de la trasformada (Sección 15.3). El número de valores del conjunto reducido puede fijarse, o puede calcularse usando un algoritmo para determinar la influencia de los diversos términos de la trasformada.
- (5) **Codificación Huffman:** se realiza una operación de compresión final convirtiendo los valores de la trasformada discreta del coseno en códigos Huffman, como se ha explicado en la Sección 15.3.

El formato **SPIFF (Still-Picture Interchange File Format)** desarrollado por Eric Hamilton y el comité ISO JPEG, es una extensión de JFIF. Este formato tiene muchas más características y opciones que JFIF, y se

espera que SPIFF sustituya eventualmente a JFIF en las implementaciones JPEG. Sin embargo, como JPEG, esta extensión del formato JFIF contiene muchas más opciones que pueden resultar prácticas en una implementación. Por ejemplo, JFIF usa sólo un modelo de color ( $YC_rC_b$ ), pero SPIFF ofrece opciones para trece modelos de color diferentes.

Para imágenes fotorrealistas de gráficos por computadora y fotografías digitalizadas, las implementaciones JPEG actuales ofrecen una mayor tasa de compresión que cualquier otro sistema. Pero otros formatos pueden ofrecer tasas de compresión comparables sin pérdida de información de color para imágenes sencillas que contengan grandes áreas de un mismo color.

### CGM: Computer-Graphics Metafile Format

Este formato es otro estándar desarrollado por ISO y ANSI. Está diseñado para usarse en cualquier sistema informático y en cualquier área de los gráficos por computadora, incluyendo la visualización científica, los sistemas CAD, las artes gráficas, la maquetación electrónica, publicidad electrónica y cualquier aplicación que utilice bibliotecas gráficas GKS o PHIGS. Así, CGM soporta una gran variedad de características y opciones.

Como indica la denominación *metafile* (metaarchivo), CGM permite que una descripción de imagen sea dada como un pixmap o como un conjunto de definiciones geométricas, incluyendo atributos tales como tamaño de la línea, tipo de línea, estilo del relleno y especificaciones de las cadenas de caracteres. En un archivo de imagen pueden incluirse muchos otros parámetros, tal como el valor máximo para las componentes de color, el tamaño de una tabla de color, la lista de fuentes usadas en el archivo y los límites de la ventana de recorte.

En CGM, se usa un esquema de codificación de caracteres para minimizar el tamaño del archivo y se optimiza un código numérico binario para codificar y decodificar el archivo de imagen más rápidamente. Los valores de píxeles pueden darse usando varios esquemas de color, tales como los modelos RGB, CMYK,  $YC_rC_b$ , CIE y tablas de color. Además, los archivos pixmap pueden comprimirse usando variantes de la codificación de longitud de recorrido y de la codificación Huffman.

### TIFF: Tag Image-File Format

Un consorcio de compañías de computadoras encabezadas por Aldus Corporation desarrollaron TIFF como un formato eficiente para la transferencia de imágenes digitalizadas entre diferentes aplicaciones y sistemas informáticos. Aunque es altamente complejo, TIFF es uno de los formatos más versátiles y puede personalizarse para aplicaciones individuales. Es ampliamente usado en aplicaciones diversas como son las imágenes médicas, la edición electrónica, las interfaces gráficas de usuario, el almacenamiento de imágenes por satélite y la transmisión de fax.

El formato TIFF puede usarse con nivel-b, escala de grises, imágenes a color completo, y los archivos TIFF se diseñan para almacenar múltiples imágenes digitalizadas. La información de píxeles de color puede proporcionarse como componentes RGB o como tablas de color. Se ofrecen más alternativas de compresión en TIFF que en cualquier otro sistema. Estos esquemas de compresión incluyen combinaciones de codificación de longitud de recorrido, codificación LZW, codificación Huffman y la serie de métodos JPEG.

### PNG: Portable Network-Graphics Format

Diseñado por un grupo de desarrolladores independientes, PNG proporciona un esquema de compresión sin pérdidas enormemente eficiente para el almacenamiento de imágenes. Los algoritmos de compresión en PNG incluyen codificación Huffman y variaciones de la codificación LZ. Este formato está ganando popularidad en Internet para el almacenamiento y la transmisión de imágenes. También es útil para almacenar imágenes temporalmente para ediciones repetidas. Para imágenes sencillas de gráficos por computadora, PNG genera archivos con tasas de compresión muy altas, comparables a aquellos archivos comprimidos con JPEG.

Los valores enteros se almacenan en orden *big-endian* y las componentes de color pueden especificarse con una precisión superior a 16 bits por píxel. PNG soporta una serie de opciones, incluyendo componentes de color RGB, componentes de color XYZ, escala de grises, tablas de color y un valor alfa para la información de transparencia.

## XBM: X Window System Bitmap Format y XPM: X Window System Pixmap Format

A diferencia de otros formatos, XBM y XPM almacenan la información de las imágenes como código C o C++ que se procesa en estaciones de trabajo usando el sistema X Window. Así, los valores de píxel se representan en matrices, se almacenan en el orden de barrido, de izquierda a derecha. Como sus propios nombres indican, XBM es un formato para bitmaps (un bit por píxel) y XPM es un formato para pixmaps (múltiples bits por píxel). Estos formatos son soportados por la mayoría de los exploradores web.

Los formatos XBM y XPM no contienen algoritmos de compresión, pero el tamaño de los archivos puede reducirse usando programas de compresión especialmente diseñados. En lugar de cabeceras de archivos, estos formatos usan directivas de preprocesador `#define` para especificar información como el número de píxeles por línea de barrido y el número de líneas de barrido. En el formato XBM, los valores de bit iguales a 1 representan el color del primer plano actual y los valores de bit iguales a 0 representan el color del fondo actual. En el formato XPM, los valores pueden almacenarse en tablas de color usando componentes RGB o HSV.

## Formato Adobe Photoshop

Usado ampliamente en aplicaciones de procesamiento de imágenes, el formato Adobe Photoshop está optimizado para el acceso rápido a imágenes digitalizadas grandes y a color completo. Por el contrario, se alcanzan compresiones muy pequeñas con el esquema de codificación de longitud de recorrido usada en Photoshop, y las primeras versiones de Photoshop no contenían métodos de compresión.

Los valores de píxel se almacenan en orden *big-endian* y Photoshop ofrece una serie de opciones. Photoshop soporta pixmaps, bitmaps (imágenes monocromáticas) e imágenes en escala de grises. Los colores pueden almacenarse usando componentes RGB, componentes de color CMYK o en tablas de color. Y se ofrecen varios esquemas para la representación de múltiples colores por píxel e imágenes semitono, así como de parámetros de transparencia.

## MacPaint: Macintosh Paint Format

Producto de Apple Corporation, MacPaint es un formato estándar para todas las aplicaciones Macintosh. Los archivos de imagen para este formato son bitmaps con el valor 0 indicando blanco y el valor 1 indicando negro. El formato McPaint se usa típicamente para texto, dibujo de líneas y *clip arts*.

Los valores de píxel se almacenan en orden *big-endian* y los archivos McPaint siempre contienen 576 píxeles por línea de barrido y 720 líneas de barrido. El esquema de codificación de longitud de recorrido se usa para comprimir los archivos de imagen.

## PICT: Formato Picture Data

Este formato híbrido es otro producto para aplicaciones Macintosh de Apple Corporation. Soporta imágenes que se especifican como bitmaps, pixmaps o representaciones geométricas. Un archivo PICT en un formato de representación geométrica que contiene una lista de funciones Macintosh QuickDraw que definen una imagen como un conjunto de segmentos de línea, polígonos, arcos, bitmaps, otros objetos, parámetros de recorte, atributos y otros parámetros de estado.

Las imágenes pueden especificarse usando un formato monocromo (bitmap), componentes de color RGB o una tabla de color. Los archivos digitalizados pueden comprimirse usando un algoritmo de codificación de longitud de recorrido.

## BMP: Formato Bitmap

Aunque se denomina formato bitmap, BMP en realidad soporta archivos de imagen que contienen múltiples bits por píxel. Este formato fue desarrollado por Microsoft Corporation para aplicaciones del sistema operativo Windows. Los sistemas operativos IBM OS/2 también emplean otro formato pixmap similar, que también se denomina BMP.

Los valores de píxel en un archivo BMP se almacenan en orden *little-indian* usando 1, 2, 4, 8, 16, 24 ó 32 bits por píxel. Los valores de píxel de color pueden especificarse con componentes de color RGB o con tablas de color. Y las líneas de píxeles de barrido se almacenan de abajo hacia arriba, con el origen de coordenadas en la posición inferior izquierda del pixmap. Un archivo BMP normalmente no se comprime, pero el algoritmo de codificación de longitud de recorrido puede aplicarse a pixmaps con 4 u 8 bits por píxel.

## PCX: Formato de archivo PC Paintbrush

Desarrollado por Zsoft Corporation, PCX es otro formato pixmap usado por sistemas operativos Windows. Los archivos de imagen en el formato PCX pueden contener desde 1 a 24 bits por píxel, y los valores de píxel pueden especificarse usando componentes RGB o tablas de color. Los valores se almacenan en orden *little-endian*, con el orden de las líneas de barrido de arriba hacia abajo. Y los archivos digitalizados pueden comprimirse usando la codificación de longitud de recorrido.

## TGA: Formato Truevision Graphics-Adapter

Desarrollado por Truevision Corporation para su uso junto con los adaptadores gráficos Targa y Vista, el formato TGA pixmap es también conocido como **formato Targa**. Este formato es usado popularmente para la edición de vídeo.

En el formato TGA, los valores de píxel se almacenan en orden *little-endian*, y los archivos de imagen pueden contener 8, 16, 24 ó 32 bits por píxel. Los colores de píxel pueden especificarse como componentes RGB o en tablas, con dos posibles formatos de tabla. Puede usarse una única tabla de color RGB, o las componentes *R*, *G* y *B* pueden venir dadas en tablas separadas. Normalmente, los archivos TGA no se comprimen, pero los algoritmos de codificación de longitud de recorrido pueden aplicarse a archivos de imagen más grandes.

## GIF: Graphics Interchange Format

Este formato, diseñado para una transmisión eficiente a través de línea telefónica de archivos de imagen digitalizados, es un producto de CompuServe Corporation. Usando un algoritmo LZW, GIF ofrece tasas de compresión razonables para imágenes de gráficos por computadora. Pero las tasas de compresión generadas por GIF para imágenes fotorrealistas no son tan buenos como aquellos producidos por JPEG o PNG. A pesar de ello, GIF se ha usado en muchas aplicaciones, aunque su popularidad ha decaído drásticamente debido a las evidentes aportaciones asociadas a los algoritmos de compresión LZW.

Tanto imágenes monocromas como multicolor pueden procesarse con GIF, pero los valores de píxel sólo pueden especificarse en un rango de 1 a 8 bits, permitiendo un máximo de 256 colores. Los valores de píxel se almacenan en orden *little-endian* usando tablas de color RGB.

## 15.5 RESUMEN

---

Para un sistema de gráficos digitalizados, un archivo de imagen normalmente es un pixmap RGB, el cual a menudo se denomina archivo digitalizado sin formato. Los valores de píxel RGB se almacenan como enteros en un rango de 0 hasta un valor máximo, el cual viene determinado por el número de bits disponibles para cada píxel. Una imagen también puede almacenarse usando una representación que contenga descripciones geométricas de componentes de la imagen, tales como segmentos de línea, áreas de relleno y *splines*.

Cuando los archivos de imágenes digitalizadas se transfieren entre sistemas o se almacenan de una forma particular, es necesario reducir el número de valores de color representados en la imagen. El número de colores se puede reducir uniformemente mediante la combinación de niveles de color de varias maneras, tal como la media de niveles. El método para reducir colores por popularidad selecciona los valores de color que aparecen más frecuentemente. Y el método de corte medio subdivide el espacio de color en un conjunto de bloques, siendo todos los colores contenidos en cada bloque sustituidos por el color promedio del bloque.

Se han desarrollado diferentes formatos para el almacenamiento de archivos de imagen de forma conveniente para aplicaciones particulares o sistemas concretos. Estos formatos difieren en la estructura de la cabecera del archivo, el orden de los bytes (*big endian* o *little endian*) para valores enteros, y los métodos usados (si se usa alguno) para reducir el tamaño del archivo para su almacenamiento. La efectividad de los métodos de reducción de archivos se puede medir por la tasa de compresión, que es la relación entre el tamaño del archivo original y el tamaño del archivo comprimido. Los algoritmos para reducir el tamaño del archivo que alteran los valores de color en un archivo de imagen se denominan con pérdidas, y aquellos que pueden re establecer los valores de color exactamente se llaman sin pérdidas. Algunos formatos de archivo también pueden emplear esquemas de reducción de color.

Un método común de compresión de archivos de imagen es la codificación de longitud de recorrido, la cual sustituye una secuencia de valores de píxel repetidos por el valor y la longitud del recorrido. El esquema de compresión de archivos LZW es una variación de la codificación de longitud de recorrido que sustituye patrones de píxeles repetidos por un código. Otros métodos de compresión de reconocimiento de patrones incluyen la comparación de líneas de barrido y procedimientos fractales para identificar conjuntos de valores de píxel autosimilares. En la codificación Huffman, un código de longitud variable es asignado a valores de color de tal forma que los valores que ocurren más frecuentemente tienen el código más corto. La codificación aritmética usa el recuento de frecuencias para valores de color en un archivo de imagen para crear subdivisiones del intervalo unidad desde 0.0 hasta 1.0. Los límites de cada subintervalo se emplean entonces para codificar las secuencias de valores de color representadas por ese subintervalo. La transformada discreta del coseno multiplica los valores de color de píxel por los términos del coseno con frecuencia creciente, y luego suma dichos productos. Este proceso sumatorio convierte un conjunto de valores de píxeles de color en un conjunto de valores transformados. La compresión de archivos se consigue entonces mediante la eliminación de algunos valores transformados, que producen una compresión con pérdidas de la imagen.

Muchos formatos de archivo están disponibles para diversas aplicaciones gráficas y para diferentes sistemas informáticos. Algunos formatos fueron desarrollados por las organizaciones de estándares ISO y ANSI, algunos surgieron de compañías de hardware o software, y algunos son productos de grupos independientes. Los formatos más ampliamente utilizados son JPEG, TIFF, PNG, y aquellos para el sistema X Window, computadoras Apple Macintosh, y sistemas operativos Windows.

## REFERENCIAS

---

Los métodos de reducción de color se presentan en Heckbert (1982 y 1994), Glassner (1990), Arvo (1991) y Kirk (1992). González y Wintz (1987) exponen los métodos de transformada y las técnicas de procesamiento de imágenes en general. Y varios algoritmos de compresión de archivos se detallan en Huffman (1952), Ziv y Lempel (1977 y 1978), Welch (1984), Rao y Yip (1990), Arvo (1991), y Barnsley y Hurd (1993).

Puede encontrarse información de carácter general sobre formatos de archivos gráficos en Brown y Shepherd (1995), y Miano (1999). Para obtener información adicional acerca de JPEG, consulte Taubman y Marcellin (2001). El formato de archivo estándar CGM se detalla en Henderson y Mumford (1993).

## EJERCICIOS

---

- 15.1** Escribir un programa para implementar reducción de color uniforme para todos los valores de color en un sistema a color completo, donde cada componente de color RGB se especifica en un rango de enteros de 0 a 255. La

entrada será cualquier división entera del factor  $d$  que se aplicará a cada componente de color y la salida es un conjunto reducido de niveles de color enteros.

- 15.2 Modificar el programa del ejercicio anterior de modo que la entrada sea un entero  $k$  que especifique el número de niveles reducidos que se va a generar en lugar del factor de división.
- 15.3 Modificar el programa del Ejercicio 15.2 de modo que un número de reducción diferente se aplique a los componentes  $R$ ,  $G$  y  $B$ . Las reducciones pueden especificarse como el rango de enteros para cada componente o el número de bits.
- 15.4 Escribir un programa para implementar el esquema de reducción de color que reduzca un archivo de imagen de entrada a  $k$  colores. La entrada del programa es una matriz de valores de píxeles y el tamaño de la matriz, especificado por el número de líneas de barrido y el número de posiciones de píxel a lo largo de cada línea de barrido.
- 15.5 Escriba un programa para implementar el esquema de reducción de color de medio corte. Un archivo de imagen que contiene  $n$  valores de píxel RGB debe reducirse a  $k$  valores de color.
- 15.6 Escriba un programa para implementar la codificación de longitud de recorrido para un sola línea de barrido que contiene 1024 valores enteros, con cada valor dentro del rango de 0 a 255.
- 15.7 Modifique el programa del ejercicio anterior para codificar un archivo que contenga  $n$  líneas de barrido.
- 15.8 Escriba un programa para implementar un algoritmo de codificación LZ simplificado para una sola línea de barrido que contiene 1024 valores enteros, con cada valor dentro del rango de 0 a 255. El programa debe buscar sólo tres elementos de patrones, representando colores RGB repetidos. Use códigos enteros para los patrones.
- 15.9 Amplíe el programa del ejercicio anterior para procesar un archivo de entrada con  $n$  líneas de barrido.
- 15.10 Dado un archivo de imagen de entrada que contiene  $n$  líneas de barrido y  $m$  colores de píxel RGB en cada línea de barrido, escriba un programa para obtener una tabla de recuentos de frecuencias para los colores de píxel.
- 15.11 Con el recuento de frecuencias del ejercicio anterior, escriba un programa para comprimir el archivo de imagen usando codificación Huffman.
- 15.12 Con el recuento de frecuencias del Ejercicio 15.10, escriba un programa para comprimir el archivo de imagen usando codificación aritmética.
- 15.13 Dada una lista de 32 colores de píxel, con tres componentes de color RGB para cada píxel, escriba un programa para calcular los valores de la transformada discreta del coseno (Ecuación 15.1) para cada grupo sucesivo de 8 píxeles de la lista.
- 15.14 Usando la Ecuación 15.2 y los valores transformados del ejercicio anterior, escriba un programa para calcular los 32 colores de píxel originales (reestablecidos).
- 15.15 Modifique el ejercicio anterior para calcular los valores de la transformada inversa para cada conjunto de 8 píxeles, usando cualquier número  $n$  seleccionado de entre los valores transformados; es decir,  $n$  puede ser cualquier valor de 1 a 8, ambos inclusive.
- 15.16 Dado un archivo de imagen que contiene 32 por 32 colores de píxel, con tres componentes de color RGB, escriba un programa para calcular los valores de la transformada discreta del coseno (Ecuación 15.3) para cada grupo sucesivo de píxeles 8 por 8.
- 15.17 Usando la Ecuación 15.4 y los valores transformados del ejercicio anterior, escriba un programa para calcular los colores de píxel de 32 por 32 originales (reestablecidos).
- 15.18 Modifique el ejercicio anterior para calcular los valores de la transformada inversa para cada conjunto de píxeles de 8 por 8, usando cualquier número  $n$  por  $m$  seleccionado de entre los valores transformados; es decir,  $n$  y  $m$  pueden cada uno ser asignados a cualquier valor entero comprendido entre 1 y 8, ambos inclusive.



# Matemáticas para gráficos por computadora

En los algoritmos de gráficos por computadora, se utilizan diversos conceptos y técnicas matemáticos. En este apéndice, vamos a proporcionar una breve referencia de los métodos de geometría analógica, álgebra lineal, análisis vectorial, análisis tensorial, números complejos, cuaternios, cálculo, análisis numérico y otras áreas de las que se trata en las explicaciones contenidas en el libro.

## A.1 SISTEMAS DE COORDENADAS

---

Tanto los sistemas de referencias cartesianos como los no cartesianos resultan útiles en las aplicaciones infográficas. Normalmente, especificamos las coordenadas en un programa gráfico utilizando un sistema de referencia cartesiano, pero la especificación inicial de una escena podría proporcionarse en un sistema de referencia no cartesiano. A menudo, las simetrías escénicas, cilíndricas o de otros tipos pueden aprovecharse para simplificar las expresiones relativas a las descripciones o manipulaciones de objetos.

### Coordenadas de pantalla cartesianas bidimensionales

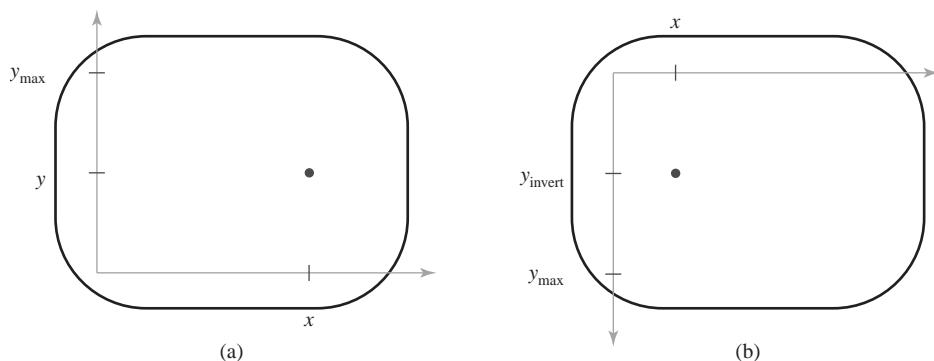
Para los comandos independientes del dispositivo incluidos dentro de un paquete gráfico, las coordenadas de pantalla se refieren dentro del primer cuadrante de un sistema cartesiano bidimensional en posición estándar, como se muestra en la Figura A.1(a). El origen de coordenadas de este sistema de referencia se encuentra situado en la esquina inferior izquierda de la pantalla. Sin embargo, las líneas de barrido se enumeran comenzando por 0 en la parte superior de la pantalla, por lo que las posiciones de pantalla están representadas internamente con respecto a la esquina superior izquierda de la misma. Por tanto, los comandos dependientes del dispositivo, como por ejemplo los relativos a la entrada interactiva y a las manipulaciones de las ventanas de visualización, suelen hacer referencia a las coordenadas de pantalla utilizando el sistema cartesiano invertido que se muestra en la Figura A.1(b). Los valores de las coordenadas horizontales en los dos sistemas son iguales, y un valor  $y$  invertido se convierte a un valor  $y$  medido desde la parte inferior de la pantalla mediante el cálculo:

$$y = y_{\max} - y_{\text{invert}} \quad (A.1)$$

En algunos paquetes de aplicación, el origen de las coordenadas de pantalla puede situarse en una posición arbitraria, como por ejemplo en el centro de la pantalla.

### Sistemas de referencia cartesianos bidimensionales estándar

Utilizamos sistemas cartesianos en posición estándar para las especificaciones en coordenadas universales, en coordenadas de visualización y para otras referencias dentro de la pipeline de visualización bidimensional. Las coordenadas en estos sistemas de referencias pueden ser positivas o negativas, con cualquier rango de valores. Para mostrar una vista de una imagen bidimensional, designamos una ventana de recorte y un visor con el fin de mapear una sección de la imagen sobre las coordenadas de pantalla.



**FIGURA A.1.** Coordenadas de pantalla cartesianas referenciadas con respecto a la esquina inferior izquierda de la pantalla (a) o a la izquierda superior izquierda de la pantalla (b).

### Coordenadas polares en el plano *xy*

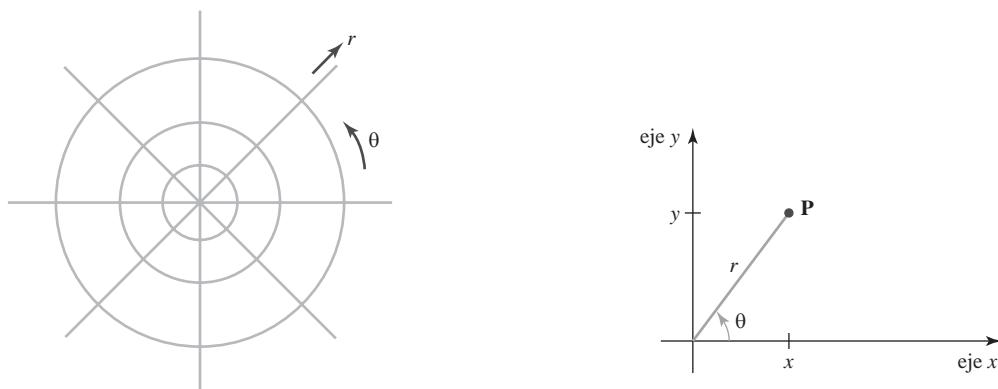
Un sistema no cartesiano bidimensional frecuentemente utilizado es el sistema de referencia en coordenadas polares (Figura A.2), en el que las coordenadas se especifican mediante una distancia radial  $r$  con respecto a un eje de coordenadas y un desplazamiento angular  $\theta$  con respecto a la horizontal. Los desplazamientos angulares positivos se definen en el sentido contrario a las agujas del reloj, mientras que los desplazamientos angulares negativos se definen en el sentido de las agujas del reloj. La relación entre coordenadas cartesianas y polares se muestra en la Figura A.3. Considerando el triángulo recto de la Figura A.4 y utilizando la definición de las funciones trigonométricas, podemos realizar la transformación de coordenadas polares a coordenadas cartesianas mediante las expresiones:

$$x = r \cos \theta, \quad y = r \sin \theta \quad (A.2)$$

La transformación inversa, de coordenadas cartesianas a coordenadas polares es:

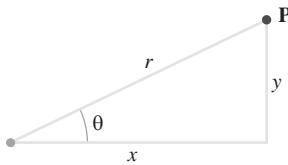
$$r = \sqrt{x^2 + y^2}, \quad \theta = \tan^{-1} \left( \frac{y}{x} \right) \quad (A.3)$$

Los valores angulares pueden medirse en grados o en unidades adimensionales (radianes). Un radian se define como el ángulo subtendido por un arco circular que tenga una longitud igual al radio del círculo. Esta

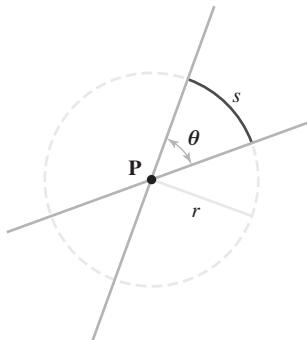


**FIGURA A.2.** Sistema de referencia en coordenadas polares, formado mediante círculos concéntricos y líneas radiales.

**FIGURA A.3.** Relación entre coordenadas polares y cartesianas.



**FIGURA A.4.** Triángulo recto con hipotenusa  $r$ , lados  $x$  e  $y$  y un ángulo interior  $\theta$ .



**FIGURA A.5.** Un ángulo  $\theta$  subtendido por un arco circular de longitud  $s$  y radio  $r$ .

definición se ilustra en la Figura A.5, que muestra dos líneas que se intersectan en un plano y un círculo centrado en el punto de intersección **P**. Para cualquier círculo centrado en **P**, el valor del ángulo  $\theta$  en radianes está dado por el cociente:

$$\theta = \frac{s}{r} \quad (\text{radianes}) \quad (A.4)$$

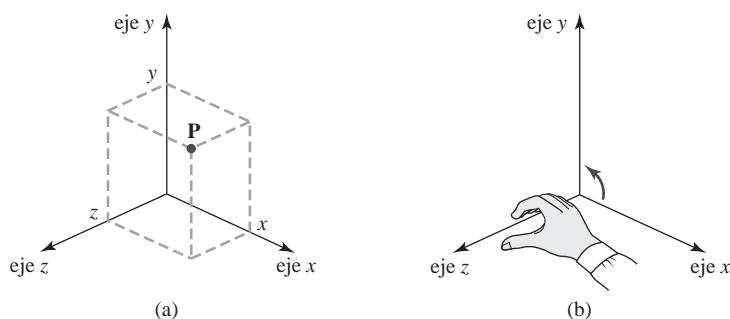
donde  $s$  es la longitud del arco circular que subtiende a  $\theta$  y  $r$  es el radio del círculo. La distancia angular total alrededor del punto P es la longitud del perímetro del círculo ( $2\pi r$ ) dividida por  $r$ , lo que es igual a  $2\pi$  radianes. Si hablamos en grados, una circunferencia se divide en 360 arcos de igual longitud, por lo que cada arco subtiende un ángulo de 1 grado. Por tanto,  $360^\circ = 2\pi$  radianes.

Pueden utilizarse otras cónicas, además de los círculos, para especificar las coordenadas. Por ejemplo, utilizando elipses concéntricas en lugar de círculos, podemos especificar los puntos en coordenadas elípticas. De forma similar, pueden aprovecharse otros tipos de simetrías para definir coordenadas planas hiperbólicas o parabólicas.

### Sistemas de referencia cartesianos tridimensionales estándar

La Figura A.6(a) muestra la orientación convencional para los ejes de coordenadas en un sistema de referencia cartesiano tridimensional. Decimos que este tipo de sistema cumple la regla de la mano derecha, porque el pulgar de la mano derecha apunta en la dirección  $z$  positiva si nos imaginamos encerrando el eje  $z$  al curvar los dedos desde el eje  $x$  positivo hacia el eje  $y$  positivo (abarcando  $90^\circ$ ), como se ilustra en la Figura A.6(b). En la mayoría de los programas infográficos, las descripciones de los objetos y otros tipos de coordenadas se especifican mediante coordenadas cartesianas que cumplen la regla de la mano derecha. En las explicaciones contenidas en el libro (incluyendo el apéndice), hemos supuesto que todos los sistemas de referencia cartesianos cumplen la regla de la mano derecha, excepto cuando indiquemos explícitamente lo contrario.

Los sistemas de referencia cartesianos son **sistemas de coordenadas ortogonales**, lo que simplemente significa que los ejes de coordenadas son perpendiculares entre sí. Asimismo, en los sistemas de referencia cartesianos, los ejes son líneas rectas. Aunque también los sistemas de coordenadas con ejes curvos resultan útiles en muchas aplicaciones. La mayoría de dichos sistemas son también ortogonales, en el sentido de que las direcciones de los ejes en cualquier punto del espacio son mutuamente perpendiculares.



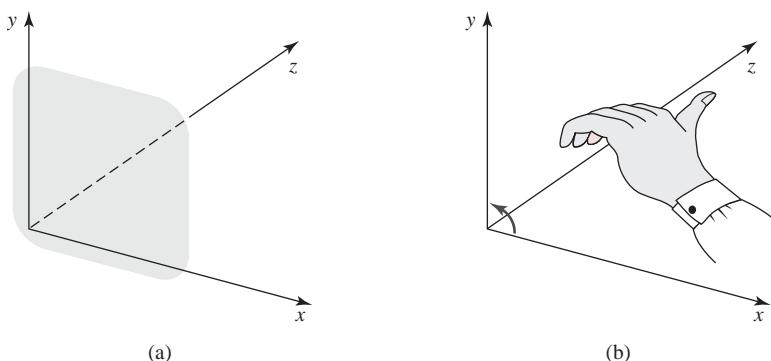
**FIGURA A.6.** Coordenadas de un punto  $P$  en la posición  $(x, y, z)$  en un sistema de referencia cartesiano estándar que cumple la regla de la mano derecha.

## Coordenadas de pantalla cartesianas tridimensionales

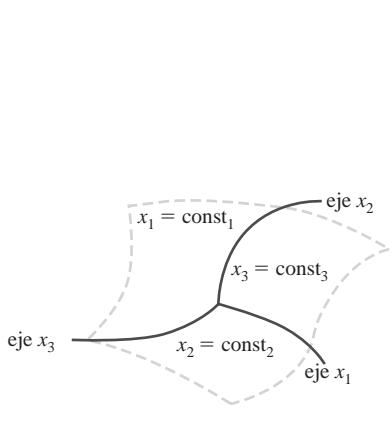
Cuando se muestra una vista de una escena tridimensional sobre un monitor de vídeo, se almacena información de profundidad para cada posición de pantalla. La posición tridimensional que corresponde a cada punto de pantalla suele estar referenciada mediante un sistema que cumple la regla de la mano izquierda, como se muestra en la Figura A.7. En este caso, el pulgar de la mano izquierda apunta en la dirección  $z$  positiva si nos imaginamos rodeando el eje  $z$  de modo que los dedos de la mano izquierda vayan desde el eje  $x$  positivo hasta el eje  $y$  positivo abarcando  $90^\circ$ . Los valores de  $z$  positivos indican posiciones situadas detrás de la pantalla para cada punto en el plano  $xy$ , y el valor positivo de  $z$  se incrementará a medida que los objetos se alejen del observador.

## Sistemas de coordenadas curvilíneas tridimensionales

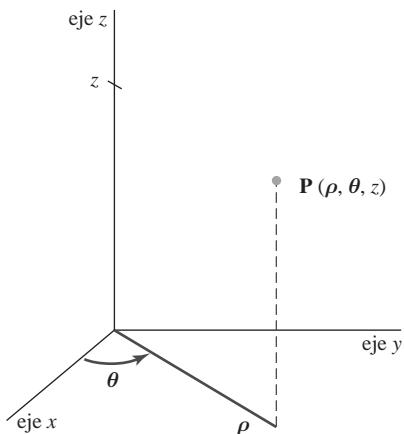
Los sistemas de referencia no cartesianos se denominan **sistemas de coordenadas curvilíneas**. La elección del sistema de coordenadas para una aplicación gráfica concreta dependerá de diversos factores, como la simetría, la facilidad de cálculo y las ventajas que puedan obtenerse de cara a la visualización. La Figura A.8 muestra un sistema de referencia general de coordenadas curvilíneas formado por tres *superficies de coordenadas*, donde en cada superficie una de las coordenadas tiene un valor constante. Por ejemplo, la superficie  $x_1x_2$  se define con  $x_3 = \text{const}_3$ . Los *ejes de coordenadas* de cualquier sistema de referencia son las curvas de intersección de las superficies de coordenadas. Si las superficies de coordenadas se intersectan siempre con ángulos rectos, tendremos un **sistema de coordenadas curvilíneas ortogonales**. Los sistemas de referencia curvilíneos no ortogonales son también útiles en algunas aplicaciones, como por ejemplo en la visualización de movimientos gobernados por las leyes de la relatividad general, pero se utilizan menos frecuentemente en gráficos por computadora que los sistemas ortogonales.



**FIGURA A.7.** Sistema de coordenadas cartesianas que cumple la regla de la mano izquierda, superpuesto sobre la superficie de un monitor de vídeo.



**FIGURA A.8.** Un sistema de coordenadas curvilíneas general.



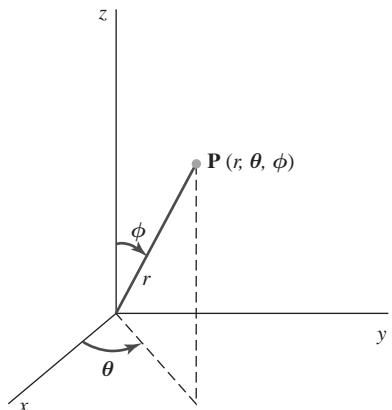
**FIGURA A.9.** Coordenadas cilíndricas  $\rho$ ,  $\theta$  y  $z$ .

En la Figura A.9 se muestra una especificación en *coordenadas cilíndricas* de un punto en el espacio en relación con un sistema de referencia cartesiano. La superficie de  $\rho$  constante es un cilindro vertical; la superficie de  $\theta$  constante es un plano vertical que contiene al eje  $z$ ; y la superficie de  $z$  constante es un plano horizontal paralelo al plano cartesiano  $xy$ . Podemos efectuar la transformación de coordenadas cilíndricas a un sistema de referencia cartesiano mediante las ecuaciones:

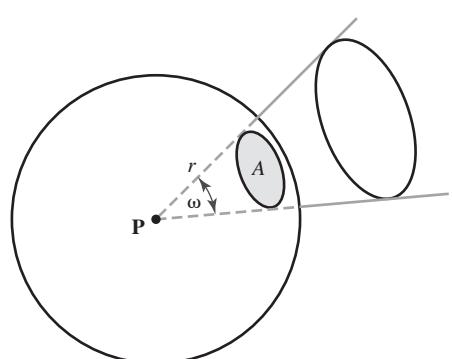
$$x = \rho \cos \theta, \quad y = \rho \sin \theta, \quad z = z \quad (A.5)$$

Otro sistema de coordenadas curvilíneas comúnmente utilizado es el sistema de *coordenadas esféricas* de la Figura A.10. Las coordenadas esféricas se denominan en ocasiones *coordenadas polares en el espacio tridimensional*. La superficie de  $r$  constante es una esfera; la superficie de  $\theta$  constante es, de nuevo, un plano vertical que contiene al eje  $z$ ; y al superficie de  $\phi$  constante es un cono cuyo vértice se encuentra en el origen de coordenadas. Si  $\phi < 90^\circ$ , el cono se encontrará por encima del plano  $xy$ . Si  $\phi > 90^\circ$ , el cono se encontrará por debajo del plano  $xy$ . Podemos efectuar la transformación de coordenadas esféricas a un sistema de referencia cartesiano mediante las ecuaciones:

$$x = r \cos \theta \sin \phi, \quad y = r \sin \theta \sin \phi, \quad z = r \cos \phi \quad (A.6)$$



**FIGURA A.10.** Coordenadas esféricas  $r$ ,  $\theta$  y  $\phi$ .



**FIGURA A.11.** Ángulo sólido  $\omega$  subtendido por un parche esférico superficial con área  $A$  y radio  $r$ .

## Ángulo sólido

La definición de ángulo sólido  $\omega$  se formula por analogía con la definición de ángulo en radianes bidimensionales  $\theta$  entre dos líneas que se intersectan (Ecuación A.4). Sin embargo, para un ángulo tridimensional, lo que hacemos es definir un cono cuyo vértice está en un punto  $\mathbf{P}$  y una esfera centrada en  $\mathbf{P}$ , como se muestra en la Figura A.11. El ángulo sólido  $\omega$  en la región cónica que tiene su vértice en el punto  $\mathbf{P}$  se define como

$$\omega = \frac{A}{r^2} \quad (A.7)$$

donde  $A$  es el área de la superficie esférica intersectada por el cono y  $r$  es el radio de la esfera.

Asimismo, por analogía con las coordenadas polares bidimensionales, la unidad adimensional para los ángulos sólidos se denomina **estereoradian**. El ángulo sólido total en torno al punto  $\mathbf{P}$  es el área total de la superficie esférica ( $4\pi r^2$ ) dividida por  $r^2$ , lo que es igual a  $4\pi$  estereoradianes.

## A.2 PUNTOS Y VECTORES

---

Existe una diferencia fundamental entre el concepto de punto geométrico y el concepto de vector. Un punto es una posición especificada mediante sus coordenadas en algún sistema de referencia, dependiendo las coordenadas y otras propiedades del punto de nuestra selección de sistema de referencia. Un vector, por el contrario, tiene propiedades que son independientes del sistema de referencia concreto que elijamos.

### Propiedades de los puntos

La Figura A.12 ilustra la especificación de un punto bidimensional  $\mathbf{P}$  mediante sus coordenadas en dos sistemas de referencias distintos. En el sistema A, el punto tiene unas coordenadas que están dadas por el par ordenado  $(x, y)$  y su distancia con respecto al origen es  $\sqrt{x^2 + y^2}$ . En el sistema B, el mismo punto tiene coordenadas  $(0, 0)$  y la distancia hasta el origen de coordenadas del sistema B se 0.

### Propiedades de los vectores

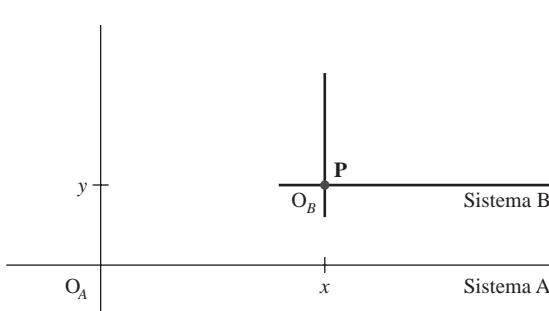
En un determinado sistema de coordenadas, podemos definir un vector como la diferencia entre dos puntos. Así, para los puntos bidimensionales  $\mathbf{P}_1$  y  $\mathbf{P}_2$  en la Figura A.13, podemos especificar un vector como:

$$\begin{aligned} \mathbf{V} &= \mathbf{P}_2 - \mathbf{P}_1 \\ &= (x_2 - x_1, y_2 - y_1) \\ &= (V_x, V_y) \end{aligned} \quad (A.8)$$

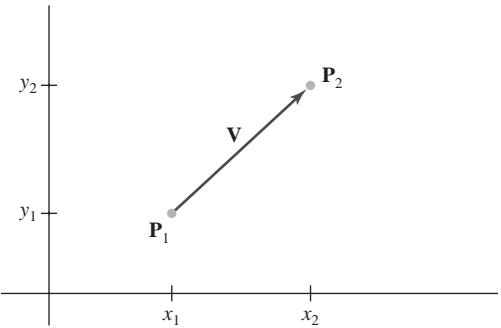
donde las *componentes* cartesianas (o *elementos* cartesianos)  $V_x$  y  $V_y$  son las proyecciones de  $\mathbf{V}$  sobre los ejes  $x$  e  $y$ . También podríamos obtener estas mismas componentes del vector utilizando otros dos puntos dentro del sistema de coordenadas. De hecho, existe un número infinito de parejas de puntos que nos dan las mismas componentes de vector, y los vectores se suelen definir mediante un único punto relativo al sistema de referencia actual. Por tanto, un vector no tiene una posición fija dentro de un sistema de coordenadas. Asimismo, si transformamos la representación de  $\mathbf{V}$  a otro sistema de referencia, las coordenadas de las posiciones  $\mathbf{P}_1$  y  $\mathbf{P}_2$  cambiarán, pero las propiedades básicas del vector no sufrirán modificación.

Podemos describir un vector como un *segmento de línea dirigido* que tiene dos propiedades fundamentales: módulo y dirección. Para el vector bidimensional de la Figura A.13, calculamos el módulo del vector utilizando el teorema de Pitágoras, que nos da la distancia entre sus dos extremos según la dirección del vector:

$$|\mathbf{V}| = \sqrt{V_x^2 + V_y^2} \quad (A.9)$$



**FIGURA A.12.** Coordenadas de un punto  $\mathbf{P}$  en dos sistemas de referencia cartesianos distintos.



**FIGURA A.13.** Un vector bidimensional  $\mathbf{V}$  definido en un sistema de referencia cartesiano como la diferencia entre dos puntos.

Podemos especificar la dirección del vector de diversas formas. Por ejemplo, podemos proporcionar la dirección en términos del desplazamiento angular con respecto a la horizontal de la forma siguiente:

$$\alpha = \tan^{-1} \left( \frac{V_y}{V_x} \right) \quad (A.10)$$

Un vector tiene el mismo módulo y dirección independientemente de dónde situemos el vector dentro de un cierto sistema de coordenadas. Asimismo, el módulo del vector es independiente del sistema de coordenadas que elijamos. Sin embargo, si transformamos el vector a otros sistema de referencia, los valores de esos componentes y su dirección dentro de ese sistema de referencia pueden cambiar. Por ejemplo, podríamos transformar el vector a un sistema de referencia cartesiano rotado, de modo que la dirección del vector esté definida ahora según la nueva dirección  $y$ .

Para una representación cartesiana tridimensional de un vector,  $\mathbf{V} = (V_x, V_y, V_z)$ , el módulo del vector será:

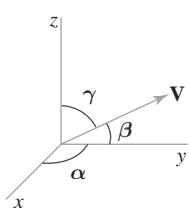
$$|\mathbf{V}| = \sqrt{V_x^2 + V_y^2 + V_z^2} \quad (A.11)$$

Y podemos dar la dirección del vector en términos de los *ángulos directores*,  $\alpha$ ,  $\beta$  y  $\gamma$ , que el vector forma con cada uno de los ejes de coordenadas (Figura A.14). Los ángulos directores son los ángulos positivos que el vector forma con cada uno de los ejes de coordenadas positivos. Podemos calcular estos ángulos de la forma siguiente:

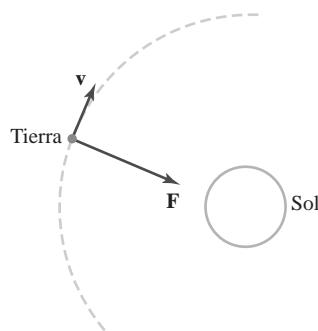
$$\cos \alpha = \frac{V_x}{|\mathbf{V}|}, \quad \cos \beta = \frac{V_y}{|\mathbf{V}|}, \quad \cos \gamma = \frac{V_z}{|\mathbf{V}|} \quad (A.12)$$

Los valores  $\cos \alpha$ ,  $\cos \beta$  y  $\cos \gamma$  se denominan cosenos directores del vector. En realidad, sólo hace falta especificar dos de los cosenos directores para proporcionar la dirección de  $\mathbf{V}$ , ya que:

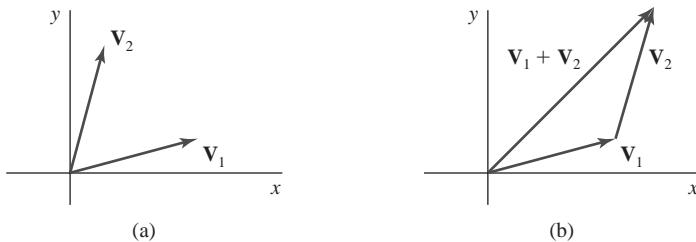
$$\cos^2 \alpha + \cos^2 \beta + \cos^2 \gamma = 1 \quad (A.13)$$



**FIGURA A.14.** Ángulos directores  $\alpha$ ,  $\beta$  y  $\gamma$ .



**FIGURA A.15.** Un vector de fuerza gravitatoria  $\mathbf{F}$  y un vector de velocidad  $\mathbf{v}$ .



**FIGURA A.16.** Dos vectores (a) pueden sumarse geométricamente situando los dos vectores uno a continuación de otro (b) y dibujando el vector resultante desde el extremo inicial del primer vector hasta el extremo final del segundo vector.

Los vectores se utilizan para representar cualquier tipo de magnitud que tenga como propiedades un módulo y una dirección. Dos ejemplos comunes son la fuerza y la velocidad (Figura A.15). Una fuerza puede considerarse como la intensidad con que se tira o empuja en una dirección concreta. Un vector de velocidad especifica la rapidez con la que un objeto se mueve en una cierta dirección.

## Suma de vectores y multiplicación escalar

Por definición, la suma de dos vectores se obtiene sumando las componentes correspondientes:

$$\mathbf{V}_1 + \mathbf{V}_2 = (V_{1x} + V_{2x}, V_{1y} + V_{2y}, V_{1z} + V_{2z}) \quad (A.14)$$

La Figura A.16 ilustra geométricamente la suma bidimensional de vectores. Obtenemos la suma de vectores colocando el extremo inicial de un vector sobre el extremo final del otro vector y dibujando la representación del vector suma desde el extremo inicial del primer vector hasta el extremo final del segundo. La suma de un vector con un escalar no está definida, ya que un escalar sólo tiene un valor numérico, mientras que un vector tiene  $n$  componentes numéricas en un espacio  $n$ -dimensional.

La multiplicación de un vector por un valor escalar  $s$  se define como

$$s\mathbf{V} = (sV_x, sV_y, sV_z) \quad (A.15)$$

Por ejemplo, si el parámetro escalar  $s$  tiene el valor 2, cada componente de  $\mathbf{V}$  se dobla y el módulo se dobla también.

También podemos combinar vectores utilizando procesos multiplicativos, de diversas formas. Un método muy útil consiste en multiplicar los módulos de los dos vectores, de modo que este producto se utilice para formar otro vector o una magnitud escalar.

## Producto escalar de dos vectores

Podemos obtener un valor escalar a partir de dos vectores mediante el cálculo:

$$\mathbf{V}_1 \cdot \mathbf{V}_2 = |\mathbf{V}_1| |\mathbf{V}_2| \cos \theta, \quad 0 \leq \theta \leq \pi \quad (A.16)$$

donde  $\theta$  es el más pequeño de los dos ángulos que pueden definirse entre las direcciones de ambos vectores (Figura A.17). Este esquema de multiplicación se denomina **producto escalar** de dos vectores. También se denomina *producto interno*, particularmente al hablar de productos escalares en el análisis tensorial. La Ecuación A.16 es válida en cualquier sistema de coordenadas y puede interpretarse como el producto de las componentes paralelas de los dos vectores, donde  $|\mathbf{V}_2| \cos \theta$  es la proyección del vector  $\mathbf{V}_2$  en la dirección de  $\mathbf{V}_1$ .

Además de expresar el producto escalar en forma independiente del sistema de coordenadas, podemos también expresar este cálculo en un sistema de coordenadas específico. Para un sistema de referencia cartesiano, el producto escalar se calcula como:

$$\mathbf{V}_1 \cdot \mathbf{V}_2 = V_{1x}V_{2x} + V_{1y}V_{2y} + V_{1z}V_{2z} \quad (A.17)$$

El producto escalar es una generalización del teorema de Pitágoras y el producto escalar de un vector por sí mismo da como resultado el cuadrado del módulo del vector. Asimismo, el producto escalar de dos vectores es cero si y sólo si los dos vectores son perpendiculares (ortogonales).

El producto escalar es commutativo:

$$\mathbf{V}_1 \cdot \mathbf{V}_2 = \mathbf{V}_2 \cdot \mathbf{V}_1 \quad (A.18)$$

porque esta operación produce un valor escalar. Asimismo, el producto escalar es distributivo con respecto a la suma de vectores:

$$\mathbf{V}_1 \cdot (\mathbf{V}_2 + \mathbf{V}_3) = \mathbf{V}_1 \cdot \mathbf{V}_2 + \mathbf{V}_1 \cdot \mathbf{V}_3 \quad (A.19)$$

## Producto vectorial de dos vectores

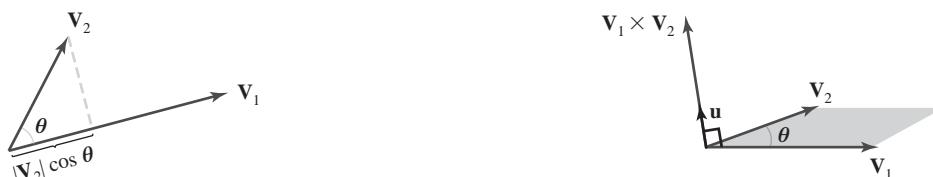
Podemos utilizar la siguiente fórmula con el fin de combinar dos vectores para producir otro vector:

$$\mathbf{V}_1 \times \mathbf{V}_2 = \mathbf{u} |\mathbf{V}_1| |\mathbf{V}_2| \sin \theta, \quad 0 \leq \theta \leq \pi \quad (A.20)$$

El parámetro  $\mathbf{u}$  en esta expresión es un vector unitario (módulo 1) perpendicular tanto a  $\mathbf{V}_1$  como a  $\mathbf{V}_2$  (Figura A.18). La dirección de  $\mathbf{u}$  está determinada por la *regla de la mano derecha*: rodeamos un eje perpendicular al plano que contiene a  $\mathbf{V}_1$  y  $\mathbf{V}_2$  de modo que los dedos de la mano derecha se curven desde  $\mathbf{V}_1$  a  $\mathbf{V}_2$ . El vector  $\mathbf{u}$  estará entonces en la dirección a la que apunta el dedo pulgar. Este cálculo se denomina **producto vectorial** de dos vectores y la Ecuación A.20 es válida en cualquier sistema de coordenadas. El producto vectorial de dos vectores es otro vector perpendicular al plano de esos dos vectores, y el módulo del producto vectorial es igual al área del paralelogramo formado por los dos vectores.

También podemos expresar el producto vectorial en términos de las componentes de los vectores dentro de un sistema de referencia específico. En un sistema de coordenadas cartesianas, calculamos las componentes del producto vectorial como:

$$\mathbf{V}_1 \times \mathbf{V}_2 = (V_{1y}V_{2z} - V_{1z}V_{2y}, V_{1z}V_{2x} - V_{1x}V_{2z}, V_{1x}V_{2y} - V_{1y}V_{2x}) \quad (A.21)$$



**FIGURA A.17.** El producto escalar de dos vectores se obtiene multiplicando las componentes paralelas.

**FIGURA A.18.** El producto vectorial de dos vectores es un vector que apunta en una dirección perpendicular a los dos vectores originales y con un módulo igual al área del paralelogramo sombreado.

Si designamos los vectores unitarios (módulo 1) según los ejes x, y, z como  $\mathbf{u}_x$ ,  $\mathbf{u}_y$  y  $\mathbf{u}_z$ , podemos escribir el producto vectorial en términos de las componentes cartesianas utilizando una notación de determinantes (Sección A.5):

$$\mathbf{V}_1 \times \mathbf{V}_2 = \begin{vmatrix} \mathbf{u}_x & \mathbf{u}_y & \mathbf{u}_z \\ V_{1x} & V_{1y} & V_{1z} \\ V_{2x} & V_{2y} & V_{2z} \end{vmatrix} \quad (A.22)$$

El producto vectorial de cualesquiera dos vectores paralelos es cero. Por tanto, el producto vectorial de un vector por sí mismo es cero. Asimismo, el producto vectorial no es conmutativo, sino anticomutativo:

$$\mathbf{V}_1 \times \mathbf{V}_2 = -(\mathbf{V}_2 \times \mathbf{V}_1) \quad (A.23)$$

También se verifica que el producto vectorial no es asociativo, es decir,

$$\mathbf{V}_1 \times (\mathbf{V}_2 \times \mathbf{V}_3) \neq (\mathbf{V}_1 \times \mathbf{V}_2) \times \mathbf{V}_3 \quad (A.24)$$

Sin embargo, el producto vectorial es distributivo con respecto a la suma y resta de vectores:

$$\mathbf{V}_1 \times (\mathbf{V}_2 + \mathbf{V}_3) = (\mathbf{V}_1 \times \mathbf{V}_2) + (\mathbf{V}_1 \times \mathbf{V}_3) \quad (A.25)$$

### A.3 TENSORES

---

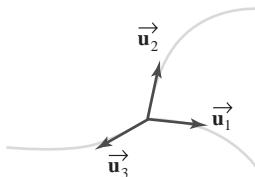
Una generalización del concepto de vector es la clase de objetos denominados tensores. Formalmente, un **tensor** se define como una magnitud con un *rango* especificado y con ciertas propiedades de transformación cuando se convierte el tensor de un sistema de coordenadas a otro. Para sistemas de coordenadas ortogonales, las propiedades de transformación son simples e iguales a las de los vectores. Diversas propiedades físicas de los objetos, como la tensión y la conductividad, son tensores.

El rango de un tensor, junto con la dimensión del espacio en la que el tensor está definido, determina el número de componentes (también denominados elementos o coeficientes) de dicho tensor. Las magnitudes escalares y los vectores son casos especiales de la clase más general de tensores. Un escalar es un tensor de rango cero, mientras que un vector es un tensor de rango uno. Básicamente, el rango de un tensor especifica el número de subíndices utilizados para designar los elementos del tensor, mientras que la dimensión espacial determina el número de valores que pueden asignarse a cada subíndice. Así, una magnitud escalar (tensor de rango cero) tiene cero subíndices, mientras que un vector (tensor de rango uno) tiene un subíndice. Algunas veces, cualquier parámetro con un subíndice se denomina incorrectamente “unidimensional” y cualquier parámetro con dos subíndices se denomina, también incorrectamente, «bidimensional». Sin embargo, la **dimensión** de una magnitud depende de la representación espacial, no del número de subíndices. En el espacio bidimensional, el único subíndice de un vector puede tener dos valores, y el vector bidimensional tendrá dos componentes. En el espacio tridimensional, el único subíndice de un vector puede tener tres valores y el vector tridimensional tiene tres componentes. De forma similar, un tensor de rango dos tiene dos subíndices y en el espacio tridimensional este tensor tendrá nueve componentes (tres valores por cada subíndice).

### A.4 VECTORES BASE Y TENSOR MÉTRICO

---

Podemos especificar las direcciones coordinadas para un sistema de referencia  $n$ -dimensional utilizando un conjunto de vectores de eje, denominados  $\bar{\mathbf{u}}_k$ , donde  $k = 1, 2, \dots, n$ , como en la Figura A.19, en la que se ilustran los vectores de eje en el origen de un espacio curvilíneo tridimensional. Cada vector de un eje de coordenadas proporciona la dirección de uno de los ejes espaciales en un punto determinado a lo largo de ese eje.



**FIGURA A.19.** Vectores de los ejes de coordenadas curvilíneas en el espacio tridimensional.

Estos vectores tangentes a los ejes forman un conjunto de vectores linealmente independiente, es decir, ninguno de los vectores de los ejes puede escribirse como combinación lineal de los restantes vectores de los ejes. Asimismo, todos los demás vectores del espacio podrán escribirse como combinación lineal de los vectores de los ejes, y el conjunto de vectores de los ejes se denomina **base** o conjunto de **vectores base**, del espacio. En general, el espacio se denominará *espacio vectorial* y la base contendrá el número mínimo de vectores necesario para representar cualquier otro vector del espacio como combinación lineal de los vectores base.

### Determinación de los vectores base para un espacio de coordenadas

Los vectores base en cualquier espacio se determinan a partir del **vector de posición**  $\vec{r}$ , que es la representación vectorial de cualquier punto del espacio. Por ejemplo, en el espacio cartesiano tridimensional, el vector de posición para cualquier punto  $(x, y, z)$  es:

$$\vec{r} = x\mathbf{u}_x + y\mathbf{u}_y + z\mathbf{u}_z \quad (A.26)$$

donde  $\mathbf{u}_x$ ,  $\mathbf{u}_y$  y  $\mathbf{u}_z$  son los vectores base unitarios para los ejes  $x$ ,  $y$  y  $z$ . A diferencia de otros sistemas de coordenadas, los vectores base cartesianos son constantes e independientes de las coordenadas en el espacio, por lo que tendremos:

$$\mathbf{u}_x = \frac{\partial \vec{r}}{\partial x}, \quad \mathbf{u}_y = \frac{\partial \vec{r}}{\partial y}, \quad \mathbf{u}_z = \frac{\partial \vec{r}}{\partial z} \quad (A.27)$$

De forma similar, para cualquier otro espacio tridimensional, podemos formular la expresión del vector de posición  $\vec{r}(x_1, x_2, x_3)$  en términos de las coordenadas de dicho espacio y luego determinar los vectores base como:

$$\vec{\mathbf{u}}_k = \frac{\partial \vec{r}}{\partial x_k}, \quad k = 1, 2, 3 \quad (A.28)$$

En general, los vectores base  $\vec{\mathbf{u}}_k$  no son ni constantes, ni vectores unitarios, sino que están en función de las coordenadas espaciales.

Como ejemplo, el vector de posición en un sistema de coordenadas polares bidimensional será:

$$\vec{r} = r \cos \theta \mathbf{u}_x + r \sin \theta \mathbf{u}_y \quad (A.29)$$

y los vectores base en coordenadas polares serán:

$$\begin{aligned} \vec{\mathbf{u}}_r &= \frac{\partial \vec{r}}{\partial r} = \cos \theta \mathbf{u}_x + \sin \theta \mathbf{u}_y \\ \vec{\mathbf{u}}_\theta &= \frac{\partial \vec{r}}{\partial \theta} = -r \sin \theta \mathbf{u}_x + r \cos \theta \mathbf{u}_y \end{aligned} \quad (A.30)$$

En este espacio,  $\vec{\mathbf{u}}_r$ , que es función de  $\theta$ , es un vector unitario. Pero  $\vec{\mathbf{u}}_\theta$ , que es función tanto de  $r$  como de  $\theta$ , no es un vector unitario.

## Bases ortonormales

A menudo, los vectores de una base se normalizan para que cada vector tenga un módulo igual a 1. Podemos obtener vectores base unitarios en cualquier espacio tridimensional mediante la fórmula:

$$\mathbf{u}_k = \frac{\vec{\mathbf{u}}_k}{|\vec{\mathbf{u}}_k|}, \quad k = 1, 2, 3 \quad (A.31)$$

y este conjunto de vectores unitarios se denomina **base normal**. Asimismo, para sistemas de referencia cartesianos, cilíndricos, esféricos y otros tipos de sistemas comunes, incluyendo las coordenadas polares, los ejes de coordenadas son mutuamente perpendiculares en cada punto del espacio y el conjunto de vectores base se denomina entonces **base ortogonal**. Un conjunto de vectores base unitario ortogonales se denomina **base ortonormal**, y estos vectores base satisfacen las siguientes ecuaciones:

$$\begin{aligned} \mathbf{u}_k \cdot \mathbf{u}_k &= 1, && \text{para todo } k \\ \mathbf{u}_j \cdot \mathbf{u}_k &= 0, && \text{para todo } j \neq k \end{aligned} \quad (A.32)$$

Aunque normalmente trataremos con sistemas ortogonales, los sistemas de referencia no ortogonales también son útiles en algunas aplicaciones, incluyendo la teoría de la relatividad y los esquemas de visualización para ciertos conjuntos de datos.

Un sistema cartesiano bidimensional tiene la base ortonormal:

$$\mathbf{u}_x = (1, 0), \quad \mathbf{u}_y = (0, 1) \quad (A.33)$$

Y la base ortonormal para un sistema de referencia cartesiano tridimensional es:

$$\mathbf{u}_x = (1, 0, 0), \quad \mathbf{u}_y = (0, 1, 0), \quad \mathbf{u}_z = (0, 0, 1) \quad (A.34)$$

## Tensor métrico

Para sistemas de coordenadas «ordinarios» (es decir, aquellos en los que podemos definir distancias y que formalmente se denominan *espacios riemannianos*), los productos escalares de los vectores base forman los elementos del denominado **tensor métrico** de dicho espacio:

$$g_{jk} = \vec{\mathbf{u}}_j \cdot \vec{\mathbf{u}}_k \quad (A.35)$$

Así, el tensor métrico tiene rango dos y es simétrico :  $g_{jk} = g_{kj}$ . Los tensores métricos tienen varias propiedades útiles. Los elementos de un tensor métrico pueden utilizarse para determinar (1) la distancia entre dos puntos del espacio, (2) las ecuaciones de transformación para la conversión a otro espacio y (3) las componentes de diversos operadores vectoriales diferenciales (como el gradiente, la divergencia y el rotacional) dentro de dicho espacio.

En un espacio ortogonal,

$$g_{jk} = 0, \quad \text{para } j \neq k \quad (A.36)$$

Por ejemplo, en un sistema de coordenadas cartesianas, en el que los vectores base son vectores unitarios constantes, el tensor métrico tiene las componentes:

$$g_{jk} = \begin{cases} 1, & \text{si } j = k \\ 0, & \text{en caso contrario} \end{cases} \quad (\text{espacio cartesiano}) \quad (A.37)$$

Y para los vectores base en coordenadas polares (Ecuaciones A.30), podemos escribir el tensor métrico en la forma matricial:

$$\mathbf{g} = \begin{bmatrix} 1 & 0 \\ 0 & r^2 \end{bmatrix} \quad (\text{coordenadas polares}) \quad (A.38)$$

Para un sistema de referencia en coordenadas cilíndricas, los vectores base son:

$$\bar{\mathbf{u}}_\rho = \cos\theta \mathbf{u}_x + \sin\theta \mathbf{u}_y, \quad \bar{\mathbf{u}}_\theta = -\rho \sin\theta \mathbf{u}_x + \rho \cos\theta \mathbf{u}_y, \quad \bar{\mathbf{u}}_z = \mathbf{u}_z \quad (A.39)$$

Y la representación matricial del vector métrico en coordenadas cilíndricas será:

$$\mathbf{g} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \rho & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (\text{coordenadas cilíndricas}) \quad (A.40)$$

En coordenadas esféricas, los vectores base son:

$$\begin{aligned} \bar{\mathbf{u}}_r &= \cos\theta \sin\phi \mathbf{u}_x + \sin\theta \sin\phi \mathbf{u}_y + \cos\phi \mathbf{u}_z \\ \bar{\mathbf{u}}_\theta &= -r \sin\theta \sin\phi \mathbf{u}_x + r \cos\theta \sin\phi \mathbf{u}_y \\ \bar{\mathbf{u}}_\phi &= r \cos\theta \cos\phi \mathbf{u}_x + r \sin\theta \cos\phi \mathbf{u}_y - r \sin\phi \mathbf{u}_z \end{aligned} \quad (A.41)$$

Utilizando estos vectores base en la Ecuación A.35, se obtiene la siguiente representación matricial para el tensor métrico:

$$\mathbf{g} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & r^2 \sin^2 \phi & 0 \\ 0 & 0 & r^2 \end{bmatrix} \quad (\text{coordenadas esféricas}) \quad (A.42)$$

## A.5 MATRICES

---

Una matriz es una disposición rectangular de magnitudes (valores numéricos, expresiones o funciones), denominados elementos de la matriz. Algunos ejemplos de matrices son:

$$\begin{bmatrix} 3.60 & -0.01 & 2.00 \\ -5.46 & 0.00 & 1.63 \end{bmatrix}, \quad \begin{bmatrix} e^x & x \\ e^{2x} & x^2 \end{bmatrix}, \quad [a_1 \quad a_2 \quad a_3], \quad \begin{bmatrix} x \\ y \\ z \end{bmatrix} \quad (A.43)$$

Las matrices se identifican de acuerdo con el número de filas y el número de columnas. Para los ejemplos anteriores, las matrices, de izquierda a derecha son 2 por 3, 2 por 2, 1 por 3 y 3 por 1. Cuando el número de filas es igual al número de columnas, como en el segundo ejemplo, la matriz se denomina *matriz cuadrada*. En general, podemos escribir una matriz  $r$  por  $c$  como:

$$\mathbf{M} = \begin{bmatrix} m_{11} & m_{12} & \cdots & m_{1c} \\ m_{21} & m_{22} & \cdots & m_{2c} \\ \vdots & \vdots & & \vdots \\ m_{r1} & m_{r2} & \cdots & m_{rc} \end{bmatrix} \quad (A.44)$$

donde  $m_{jk}$  representa los elementos de la matriz  $\mathbf{M}$ . El primer subíndice de cada elemento proporciona el número de fila y el segundo subíndice proporciona el número de columna.

Una matriz con una única fila o una única columna representa un vector. Así, los dos últimos ejemplos de matrices en A.43 son, respectivamente un *vector fila* y un *vector columna*. En general, una matriz puede considerarse como una colección de vectores fila o como una colección de vectores columna.

Cuando expresamos diversas operaciones en forma matricial, el convenio matemático estándar consiste en representar un vector mediante una matriz columna. De acuerdo con este convenio, escribiremos la representación matricial de un vector tridimensional en coordenadas cartesianas como:

$$\mathbf{V} = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} \quad (A.45)$$

Aunque utilicemos esta representación matricial estándar tanto para los puntos como para los vectores, existe una distinción importante entre ambos conceptos. La representación vectorial de un punto siempre asume que el vector está definido desde el origen hasta dicho punto y la distancia del punto al origen no será invariante cuando cambiemos de un sistema de coordenada a otro. Asimismo, no podemos «sumar» puntos y tampoco podemos aplicar a los puntos operaciones vectoriales, como el producto escalar y el producto vectorial.

### Multiplicación por un escalar y suma de matrices

Para multiplicar una matriz  $\mathbf{M}$  por un valor escalar  $s$ , multiplicamos cada elemento  $m_{jk}$  por dicho escalar. Como ejemplo, si

$$\mathbf{M} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

entonces,

$$3\mathbf{M} = \begin{bmatrix} 3 & 6 & 9 \\ 12 & 15 & 18 \end{bmatrix}$$

La suma de matrices sólo está definida para aquellas matrices que tengan el mismo número de filas  $r$  y el mismo número de columnas  $c$ . Para cualesquiera dos matrices  $r$  por  $c$ , la suma se obtiene sumando los correspondientes elementos. Por ejemplo,

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 0.0 & 1.5 & 0.2 \\ -6.0 & 1.1 & -10.0 \end{bmatrix} = \begin{bmatrix} 1.0 & 3.5 & 3.2 \\ -2.0 & 6.1 & -4.0 \end{bmatrix}$$

### Multiplicación de matrices

El producto de dos matrices se define como una generalización del producto escalar de vectores. Podemos multiplicar una matriz  $m$  por  $n$   $\mathbf{A}$  por una matriz  $p$  por  $q$   $\mathbf{B}$  para formar la matriz producto  $\mathbf{AB}$ , supuesto que el número de columnas de  $\mathbf{A}$  sea igual al número de filas de  $\mathbf{B}$ . En otras palabras, debe cumplirse que  $n = p$ . Entonces, se obtiene la matriz producto formando las sumas de los productos de los elementos de los vectores fila de  $\mathbf{A}$  por los elementos correspondientes de los vectores columna de  $\mathbf{B}$ . Así, para el siguiente producto:

$$\mathbf{C} = \mathbf{A} \mathbf{B} \quad (A.46)$$

obtenemos una matriz  $m$  por  $q$   $\mathbf{C}$  cuyos elementos se calculan como:

$$c_{ij} = \sum_{k=1}^n a_{ik} b_{kj} \quad (A.47)$$

En el siguiente ejemplo, una matriz 3 por 2 se postmultiplica por una matriz 2 por 2, para generar una matriz producto 3 por 2:

$$\begin{bmatrix} 0 & -1 \\ 5 & 7 \\ -2 & 8 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 0 \cdot 1 + (-1) \cdot 3 & 0 \cdot 2 + (-1) \cdot 4 \\ 5 \cdot 1 + 7 \cdot 3 & 5 \cdot 2 + 7 \cdot 4 \\ -2 \cdot 1 + 8 \cdot 3 & -2 \cdot 2 + 8 \cdot 4 \end{bmatrix} = \begin{bmatrix} -3 & -4 \\ 26 & 38 \\ 22 & 28 \end{bmatrix}$$

La multiplicación de vectores en notación matricial produce el mismo resultado que el producto escalar, supuesto que el primer vector se exprese como un vector fila y el segundo vector se exprese como un vector columna. Por ejemplo,

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} = [32]$$

Este producto de vectores da como resultado una matriz con un único elemento (una matriz 1 por 1). Sin embargo, si multiplicamos los vectores en orden inverso, obtendremos la siguiente matriz 3 por 3:

$$\begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} = \begin{bmatrix} 4 & 8 & 12 \\ 5 & 10 & 15 \\ 6 & 12 & 18 \end{bmatrix}$$

Como ilustran los dos productos de vectores anteriores, la multiplicación de matrices no es commutativa por regla general. Es decir,

$$\mathbf{A} \mathbf{B} \neq \mathbf{B} \mathbf{A} \quad (A.48)$$

Pero la multiplicación de matrices es distributiva con respecto a la suma de matrices:

$$\mathbf{A} (\mathbf{B} + \mathbf{C}) = \mathbf{A} \mathbf{B} + \mathbf{A} \mathbf{C} \quad (A.49)$$

## Traspuesta de una matriz

La **traspuesta**  $\mathbf{M}^T$  de una matriz se obtiene intercambiando las filas y columnas de la matriz. Por ejemplo,

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}^T = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}, \quad [a \ b \ c]^T = \begin{bmatrix} a \\ b \\ c \end{bmatrix} \quad (A.50)$$

Para un producto de matrices, la traspuesta es:

$$(\mathbf{M}_1 \mathbf{M}_2)^T = \mathbf{M}_2^T \mathbf{M}_1^T \quad (A.51)$$

## Determinante de una matriz

Si tenemos una matriz cuadrada, podemos combinar los elementos de una matriz para generar un único número, denominado **determinante** de la matriz. Las evaluaciones de determinantes resultan muy útiles a la hora de analizar y resolver un amplio rango de problemas. Para una matriz 2 por 2  $\mathbf{A}$ , el **determinante de segundo orden** se define como:

$$\det \mathbf{A} = \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} = a_{11}a_{22} - a_{12}a_{21} \quad (A.52)$$

Los determinantes de orden superior se obtienen recursivamente a partir de los valores de los determinantes de orden inferior. Para calcular un determinante de orden 2 o superior, podemos seleccionar cualquier columna  $k$  de una matriz  $n$  por  $n$  y calcular el determinante como:

$$\det \mathbf{A} = \sum_{j=1}^n (-1)^{j+k} a_{jk} \det \mathbf{A}_{jk} \quad (A.53)$$

donde  $\det \mathbf{A}_{jk}$  es el determinante  $(n-1)$  por  $(n-1)$  de la submatriz que se obtiene a partir de  $\mathbf{A}$  borrando la fila  $j$ -ésima y la columna  $k$ -ésima. Alternativamente, podemos seleccionar cualquier fila  $j$  y calcular el determinante como:

$$\det \mathbf{A} = \sum_{k=1}^n (-1)^{j+k} a_{jk} \det \mathbf{A}_{jk} \quad (A.54)$$

La evaluación de determinantes para matrices de gran tamaño (por ejemplo,  $n > 4$ ) puede realizarse de manera más eficiente utilizando métodos numéricos. Una forma de calcular un determinante consiste en descomponer una matriz en dos factores:  $\mathbf{A} = \mathbf{LU}$ , donde todos los elementos de la matriz  $\mathbf{L}$  por encima de la diagonal son cero y todos los elementos de la matriz  $\mathbf{U}$  por debajo de la diagonal son cero. Entonces, podemos calcular el producto de las diagonales tanto para  $\mathbf{L}$  como para  $\mathbf{U}$  y obtener  $\det \mathbf{A}$  multiplicando los dos productos diagonales. Este método se basa en la siguiente propiedad de los determinantes:

$$\det(\mathbf{A} \mathbf{B}) = (\det \mathbf{A})(\det \mathbf{B}) \quad (A.55)$$

Otro método numérico para calcular determinantes se basa en los procedimientos de eliminación gausiana que se explican en la Sección A.14.

## Inversa de una matriz

Con las matrices cuadradas, podemos obtener una *matriz inversa* si y sólo si el determinante de la matriz es distinto de cero. Si tiene una inversa, se dice que la matriz es una **matriz no singular**. En caso contrario, la matriz se denomina **matriz singular**. En la mayoría de las aplicaciones prácticas, en las que las matrices representan operaciones físicas, lo normal es que exista la inversa.

La inversa de una matriz (cuadrada)  $n$  por  $n$   $\mathbf{M}$  se designa  $\mathbf{M}^{-1}$ , cumpliéndose que:

$$\mathbf{MM}^{-1} = \mathbf{M}^{-1}\mathbf{M} = \mathbf{I} \quad (A.56)$$

donde  $\mathbf{I}$  es la matriz identidad. Todos los elementos diagonales de  $\mathbf{I}$  tienen el valor 1 y todos los demás elementos (los no situados en la diagonal) son cero.

Los elementos de la matriz inversa  $\mathbf{M}^{-1}$  pueden calcularse a partir de los elementos de  $\mathbf{M}$  mediante la fórmula:

$$m_{jk}^{-1} = \frac{(-1)^{j+k} \det \mathbf{M}_{kj}}{\det \mathbf{M}} \quad (A.57)$$

donde  $m_{jk}^{-1}$  es el elemento en la fila  $j$ -ésima y la columna  $k$ -ésima de  $\mathbf{M}^{-1}$  y  $\mathbf{M}_{kj}$  es la submatriz  $(n-1)$  por  $(n-1)$  que se obtiene al borrar la fila  $k$ -ésima y la columna  $j$ -ésima de la matriz  $\mathbf{M}$ . Para valores de  $n$ , podemos calcular de manera más eficiente los valores de los determinantes y los elementos de la matriz inversa utilizando métodos numéricos.

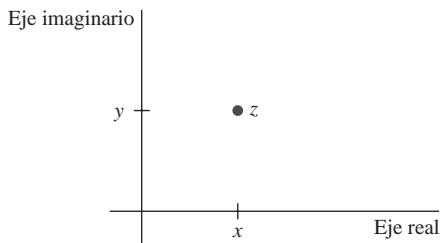
## A.6 NÚMEROS COMPLEJOS

---

Por definición, un **número complejo**  $z$  es una pareja ordenada de números reales, representada como:

$$z = (x, y) \quad (A.58)$$

donde  $x$  se denomina **parte real** de  $z$  e  $y$  se denomina **parte imaginaria** de  $z$ . Las partes real e imaginaria de un número complejo se designan como:



**FIGURA A.20.** Componentes real e imaginaria de un punto  $z$  en el plano complejo.

$$x = \operatorname{Re}(z), \quad y = \operatorname{Im}(z) \quad (A.59)$$

Geométricamente, un número complejo puede describirse como un punto en el *plano complejo*, como se ilustra en la Figura A.20.

Cuando  $\operatorname{Re}(z) = 0$ , decimos que el número complejo  $z$  es un *número imaginario puro*. De forma similar, cualquier número real puede representarse como un número complejo con  $\operatorname{Im}(z) = 0$ . Así, podemos escribir cualquier número real en la forma:

$$x = (x, 0)$$

Los números complejos surgen como solución de ecuaciones tales como:

$$x^2 + 1 = 0, \quad x^2 - 2x + 5 = 0$$

que no tienen ningún número real como solución. Así, el concepto de número complejo y las reglas para la aritmética compleja se han desarrollado como extensiones de las operaciones con números reales que proporcionan la solución a tales tipos de problemas.

## Aritmética compleja básica

La suma, la resta y la multiplicación por un escalar de los números complejos se llevan a cabo utilizando las mismas reglas que para los vectores bidimensionales. Por ejemplo, la suma de dos números complejos es:

$$z_1 + z_2 = (x_1, y_1) + (x_2, y_2) = (x_1 + x_2, y_1 + y_2)$$

y podemos expresar cualquier número complejo como la suma:

$$z = (x, y) = (x, 0) + (0, y)$$

El producto de dos números complejos,  $z_1$  y  $z_2$ , se define como:

$$(x_1, y_1)(x_2, y_2) = (x_1x_2 - y_1y_2, x_1y_2 + x_2y_1) \quad (A.60)$$

Esta definición de la multiplicación compleja da el mismo resultado que la multiplicación de números reales cuando las partes imaginarias son cero:

$$(x_1, 0)(x_2, 0) = (x_1x_2, 0)$$

## Unidad imaginaria

El número imaginario puro con  $y = 1$  se denomina **unidad imaginaria** y se escribe:

$$i = (0, 1) \quad (A.61)$$

(Los ingenieros eléctricos utilizan a menudo el símbolo  $j$  como unidad imaginaria, porque el símbolo  $i$  se utiliza para representar la corriente eléctrica).

A partir de la regla de la multiplicación compleja, tendremos que:

$$i^2 = (0, 1)(0, 1) = (-1, 0)$$

Por tanto,  $i_2$  es el número real  $-1$ , y

$$i = \sqrt{-1} \quad (A.62)$$

Podemos representar un número imaginario puro utilizando cualquiera de las dos siguientes formas:

$$z = iy = (0, y)$$

Y un número complejo general puede expresarse en la forma:

$$z = x + iy \quad (A.63)$$

Utilizando la definición de  $i$ , podemos verificar que esta representación satisface las reglas de la suma, resta y multiplicación complejas.

### Conjugado complejo y módulo de un número complejo

Otro concepto asociado con un número complejo es el *complejo conjugado*, que se define como

$$\bar{z} = x + iy \quad (A.64)$$

Así, el complejo conjugado  $\bar{z}$  es la reflexión de  $z$  con respecto al eje  $x$  (real).

El *módulo*, o *valor absoluto*, de un número complejo se define como:

$$|z| = \sqrt{z\bar{z}} = \sqrt{x^2 + y^2} \quad (A.65)$$

Este número nos da la distancia del punto  $z$  en el plano complejo con respecto al origen, lo que a veces se denomina «longitud del vector» del número complejo. Así, el valor absoluto del número complejo es simplemente una representación del teorema de Pitágoras en el plano complejo.

### División compleja

Para evaluar el cociente de dos números complejos, podemos simplificar la expresión multiplicando el numerador y el denominador por el conjugado complejo del denominador. Después, utilizamos las reglas de multiplicación para determinar los valores de las componentes del número complejo resultante. Así, las partes real e imaginaria del cociente de dos números complejos se obtienen como:

$$\begin{aligned} \frac{z_1}{z_2} &= \frac{\overline{z_1} \overline{z_2}}{\overline{z_2} \overline{z_2}} \\ &= \frac{(x_1, y_1)(x_2, -y_2)}{x_2^2 + y_2^2}, \\ &= \left( \frac{x_1 x_2 + y_1 y_2}{x_2^2 + y_2^2}, \frac{x_2 y_1 - x_1 y_2}{x_2^2 + y_2^2} \right) \end{aligned} \quad (A.66)$$

### Representación en coordenadas polares de un número complejo

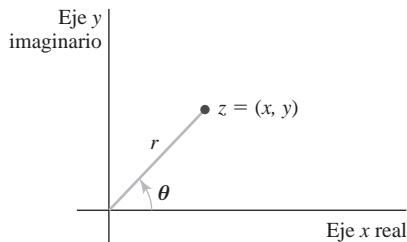
Las operaciones de multiplicación y división de los números complejos se pueden simplificar enormemente si expresamos las partes real e imaginarias en términos de las coordenadas polares (Figura A.21):

$$z = r(\cos \theta + i \sin \theta) \quad (A.67)$$

También podemos escribir la forma polar de  $z$  como:

$$z = re^{i\theta} \quad (A.68)$$

donde  $e$  es la base de los logaritmos naturales ( $e \approx 2.718281828$ ), y



**FIGURA A.21.** Parámetros en coordenadas polares dentro del plano complejo.

$$e^{i\theta} = \cos \theta + i \sin \theta \quad (A.69)$$

que es la *fórmula de Euler*.

Utilizando la forma en coordenadas polares, calculamos el producto de dos números complejos multiplicando sus valores absolutos y sumando sus ángulos polares. Así,

$$z_1 z_2 = r_1 r_2 e^{i(\theta_1 + \theta_2)} \quad (A.70)$$

Para dividir un número complejo por otro, dividimos sus valores absolutos y restamos sus ángulos polares:

$$\frac{z_1}{z_2} = \frac{r_1}{r_2} e^{i(\theta_1 - \theta_2)} \quad (A.71)$$

También podemos utilizar la representación polar para calcular las raíces de los números complejos. Las raíces  $n$ -ésimas de un número complejo se calculan como:

$$\sqrt[n]{z} = \sqrt[n]{r} \left[ \cos \left( \frac{\theta + 2k\pi}{n} \right) + i \sin \left( \frac{\theta + 2k\pi}{n} \right) \right], \quad k = 0, 1, 2, \dots, n-1 \quad (A.72)$$

Estas raíces descansan sobre un círculo de radio  $\sqrt[n]{r}$  cuyo centro está en el origen del plano complejo y forman los vértices de un polígono regular de  $n$  lados.

## A.7 CUATERNIOS

El concepto de número complejo se puede ampliar a un número mayor de dimensiones utilizando **cuaternios**, que son magnitudes con una parte real y tres partes imaginarias, que se escriben como:

$$q = s + ia + jb + kc \quad (A.73)$$

donde los coeficientes  $a, b$  y  $c$  en los términos imaginarios son números reales y el parámetro  $s$  es otro número real, que se denomina *parte escalar*. Los parámetros  $i, j, k$  se definen con las propiedades:

$$i^2 = j^2 = k^2 = -1, \quad ij = -ji = k \quad (A.74)$$

A partir de estas propiedades, se sigue que:

$$jk = -kj = i, \quad ki = -ik = j \quad (A.75)$$

La multiplicación por un escalar se define por analogía con las correspondientes operaciones relativas a vectores y números complejos. Es decir, cada una de las cuatro componentes del cuaternion se multiplica por el valor. De forma similar, la suma de cuaternios se define como la suma de los correspondientes elementos:

$$q_1 + q_2 = (s_1 + s_2) + i(a_1 + a_2) + j(b_1 + b_2) + k(c_1 + c_2) \quad (A.76)$$

La multiplicación de dos cuaternios se lleva a cabo utilizando las operaciones descritas en las Ecuaciones A.74 y A.75.

También podemos utilizar la siguiente notación de pares ordenados para un cuaterniono, que es similar a la representación de pares ordenados de un número complejo:

$$q = (s, \mathbf{v}) \quad (A.77)$$

El parámetro  $\mathbf{v}$  en esta representación es el vector  $(a, b, c)$ . Utilizando la notación de pares ordenados, podemos expresar la suma de cuaternios de la forma:

$$q_1 + q_2 = (s_1 + s_2, \mathbf{v}_1 + \mathbf{v}_2) \quad (A.78)$$

Podemos escribir la fórmula de la multiplicación de cuaternios de manera relativamente compacta si utilizamos las operaciones de producto escalar y producto vectorial de dos vectores, de la manera siguiente:

$$q_1 q_2 = (s_1 s_2 - \mathbf{v}_1 \cdot \mathbf{v}_2, s_1 \mathbf{v}_2 + s_2 \mathbf{v}_1 + \mathbf{v}_1 \times \mathbf{v}_2) \quad (A.79)$$

El módulo al cuadrado de un cuaterniono se define por analogía con las operaciones de números complejos, utilizando la siguiente suma de los cuadrados de los componentes de los cuaternios.

$$|q|^2 = s^2 + \mathbf{v} \cdot \mathbf{v} \quad (A.80)$$

Y la inversa de un cuaterniono se evalúa utilizando la expresión:

$$q^{-1} = \frac{1}{|q|^2} (s, -\mathbf{v}) \quad (A.81)$$

de modo que,

$$qq^{-1} = q^{-1}q = (1, 0)$$

## A.8 REPRESENTACIONES NO PARAMÉTRICAS

---

Cuando escribimos las descripciones de los objetos directamente en términos de las coordenadas correspondientes al sistema de referencias que estamos utilizando, la representación se denomina **no paramétrica**. Por ejemplo, podemos describir una superficie con cualquiera de las siguientes funciones cartesianas:

$$f_1(x, y, z) = 0, \quad \text{o} \quad z = f_2(x, y) \quad (A.82)$$

La primera forma de la Ecuación A.82 se denomina fórmula *implícita* de la superficie, mientras que la segunda forma se denomina representación *explícita*. En la representación explícita,  $x$  e  $y$  se denominan variables independientes,  $y$   $z$  se denomina variable dependiente.

De forma similar, podemos representar una línea curva tridimensional en forma no paramétrica como la intersección de dos funciones de superficie, o bien podemos representar la curva con la pareja de funciones:

$$y = f(x) \quad y \quad z = g(x) \quad (A.83)$$

con la coordenada  $x$  como variable independiente. Los valores de las variables dependientes  $y$  y  $z$  se determinan entonces a partir de las Ecuaciones A.83 asignando valores a  $x$  para algún número prescrito de intervalos.

Las representaciones no paramétricas resultan útiles para describir los objetos de un cierto sistema de referencia, pero presentan algunas desventajas a la hora de utilizarlas en algoritmos gráficos. Si queremos obtener una gráfica suave, debemos cambiar la variable independiente cuando la primera derivada (pendiente) de  $f(x)$  o  $g(x)$  sea superior a 1. Esto requiere controlar de manera continua los valores de las derivadas para determinar cuándo es necesario cambiar los roles de las variables dependientes e independientes. Asimismo, las Ecuaciones A.83 proporcionan un formato muy engorroso para representar funciones multivaluadas. Por ejemplo, la ecuación implícita de un círculo centrado en el origen dentro del plano  $xy$  es:

$$x^2 + y^2 - r^2 = 0$$

y la ecuación explícita de  $y$  es la función multivaluada:

$$y = \pm\sqrt{r^2 - x^2}$$

En general, una representación más conveniente para describir los objetos en los algoritmos gráficos es en términos de ecuaciones paramétricas.

## A.9 REPRESENTACIONES PARAMÉTRICAS

---

Podemos clasificar los objetos según el número de parámetros necesarios para describir las coordenadas de los mismos. Una curva, por ejemplo, en un sistema de referencia cartesiano se clasifica como un objeto euclídeo unidimensional, mientras que una superficie es un objeto euclídeo bidimensional. Cuando se proporciona la descripción de un objeto en términos de su parámetro de dimensionalidad, la descripción se denomina **representación paramétrica**.

La descripción cartesiana de los puntos situados a lo largo de la trayectoria de una curva puede proporcionarse en forma paramétrica utilizando la siguiente función vectorial:

$$\mathbf{P}(u) = (x(u), y(u), z(u)) \quad (A.84)$$

donde cada una de las coordenadas cartesianas es una función del parámetro  $u$ . En la mayoría de los casos, podemos normalizar las tres funciones de coordenadas de modo que el parámetro  $u$  varíe en el rango comprendido entre 0 y 1.0. Por ejemplo, un círculo en el plano  $xy$  con radio  $r$  y con centro en el origen de coordenadas puede definirse en forma paramétrica mediante las siguientes tres funciones:

$$x(u) = r \cos(2\pi u), \quad y(u) = r \sin(2\pi u), \quad z(u) = 0, \quad 0 \leq u \leq 1 \quad (A.85)$$

Puesto que esta curva está definida en el plano  $xy$ , podemos eliminar la función  $z(u)$ , que tiene el valor constante 0.

De forma similar, podemos representar las coordenadas de una superficie utilizando la siguiente función vectorial cartesiana:

$$\mathbf{P}(u, v) = (x(u, v), y(u, v), z(u, v)) \quad (A.86)$$

Cada una de las coordenadas cartesianas es ahora función de los dos parámetros de la superficie  $u$  y  $v$ . Una superficie esférica con radio  $r$  y con centro en el origen de coordenadas, por ejemplo, puede describirse mediante las ecuaciones:

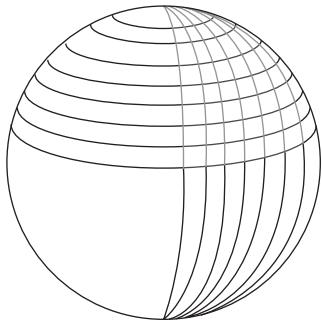
$$\begin{aligned} x(u, v) &= r \cos(2\pi u) \sin(\pi v) \\ y(u, v) &= r \sin(2\pi u) \sin(\pi v) \quad 0 \leq u, v \leq 1 \\ z(u, v) &= r \cos(\pi v) \end{aligned} \quad (A.87)$$

El parámetro  $u$  define líneas de longitud constante sobre la superficie, mientras que el parámetro  $v$  describe líneas de latitud constante. Las ecuaciones paramétricas, de nuevo, se suelen normalizar para asignar a  $u$  y a  $v$  valores en el rango comprendido entre 0 y 1.0. Manteniendo uno de estos parámetros fijo mientras se varía el otro en un subrango del intervalo unitario, podemos dibujar las líneas de latitud y longitud de cualquier sección esférica (Figura A.22).

## A.10 OPERADORES DIFERENCIALES

---

Para una función continua de una única variable independiente, como  $f(x)$ , podemos determinar la tasa a la que varía la función para cualquier valor  $x$  concreto utilizando una función denominada *derivada de  $f(x)$  con respecto a  $x$* . Esta función derivada se define como:



**FIGURA A.22.** Sección de una superficie esférica descrita por líneas de  $u$  constante y líneas de  $v$  constante en las Ecuaciones A.87.

$$\frac{df}{dx} \equiv \lim_{\Delta x \rightarrow 0} \frac{f(x + \Delta x) - f(x)}{\Delta x} \quad (A.88)$$

y esta definición es la base para obtener soluciones numéricas a problemas que impliquen operaciones de tipo diferencial. Las formas funcionales de las derivadas de las funciones más comunes, como los polinomios y las funciones trigonométricas, pueden consultarse en las tablas de derivadas. Y para los problemas de diferenciales que impliquen funciones simples, normalmente podremos obtener soluciones cerradas. Sin embargo, en muchos casos, será necesario resolver los problemas diferenciales utilizando métodos numéricos.

Cuando tenemos una función de varias variables, las operaciones diferenciales efectuadas con respecto a las variables individuales se denominan *derivadas parciales*. Por ejemplo, con una función tal como  $f(x, y, z, t)$ , podemos determinar la tasa de cambio de la función con respecto a cualquiera de los ejes de coordenadas,  $x$ ,  $y$  o  $z$ , o con respecto al parámetro temporal  $t$ . Una derivada parcial para una variable independiente concreta se define mediante la Ecuación A.88, manteniendo constantes todas las demás variables independientes. Así, por ejemplo, la derivada parcial de  $f$  con respecto al tiempo se define como:

$$\frac{\partial f}{\partial t} \equiv \lim_{\Delta t \rightarrow 0} \frac{f(x, y, z, t + \Delta t) - f(x, y, z, t)}{\Delta t}$$

evaluándose esta fórmula en una determinada posición espacial y en un determinado instante de tiempo.

Hay una serie de operadores de derivadas parciales que aparecen de manera lo suficientemente frecuente como para asignarles nombres especiales, como el gradiente, la laplaciana, la divergencia y el rotacional. Estos operadores resultan útiles en diversas aplicaciones, como por ejemplo a la hora de determinar la geometría y orientación de los objetos, a la hora de describir el comportamiento de los objetos en ciertas situaciones, a la hora de calcular los efectos de la radiación electromagnética o a la hora de utilizar conjuntos de datos en los trabajos de visualización científica.

## Operador gradiente

El operador vectorial con las siguientes componentes cartesianas se denomina *operador gradiente*:

$$\text{grad} = \nabla \equiv \left( \frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z} \right) \quad (A.89)$$

El símbolo  $\nabla$  se denomina *nabla*, *del* o simplemente *operador grad*. Una aplicación importante del operador gradiente es a la hora de calcular un vector normal a una superficie. Cuando se describe una superficie con la representación no paramétrica  $f(x, y, z) = \text{constante}$ , la normal a la superficie en cualquier posición se calcula como:

$$\mathbf{N} = \nabla f \quad (\text{Vector normal para una representación paramétrica de la superficie}) \quad (A.90)$$

Como ejemplo, una superficie esférica con radio  $r$  puede representarse en coordenadas locales con la representación cartesiana no paramétrica  $f(x, y, z) = x^2 + y^2 + z^2 = r^2$ , y el gradiente de  $f$  producirá el vector normal a la superficie  $(2x, 2y, 2z)$ . Pero si se representa la superficie con una función vectorial paramétrica  $\mathbf{P}(u, v)$ , entonces podemos determinar la normal a la superficie utilizando la operación de producto vectorial:

$$\mathbf{N} = \frac{\partial \mathbf{P}}{\partial u} \times \frac{\partial \mathbf{P}}{\partial v} \quad (\text{vector normal para una representación parámetrica de la superficie}) \quad (A.91)$$

## Derivada direccional

También podemos utilizar el operador gradiente y el producto escalar de vectores para formar un producto escalar denominado *derivada direccional* de una función  $f$ :

$$\frac{\partial f}{\partial u} = \mathbf{u} \cdot \nabla f \quad (A.92)$$

Esto nos proporciona la tasa de cambio de  $f$  en una dirección especificada por el vector unitario  $\mathbf{u}$ . Como ilustración, podemos determinar la derivada direccional para la función de superficie esférica  $f = x^2 + y^2 + z^2$  en la dirección  $z$  de la forma siguiente:

$$\frac{\partial f}{\partial z} = \mathbf{u}_z \cdot \nabla f = 2z$$

donde  $\mathbf{u}_z$  es el vector unitario según la dirección  $z$  positiva. Y para el siguiente vector unitario en el plano  $xy$ :

$$\mathbf{u} = \frac{1}{\sqrt{2}} \mathbf{u}_x + \frac{1}{\sqrt{2}} \mathbf{u}_y$$

la derivada direccional de  $f$  a partir de la Ecuación A.92 es:

$$\frac{\partial f}{\partial u} = \frac{1}{\sqrt{2}} \frac{\partial f}{\partial x} + \frac{1}{\sqrt{2}} \frac{\partial f}{\partial y} = \sqrt{2}x + \sqrt{2}y$$

## Forma general del operador gradiente

Dentro de cualquier sistema ortogonal de coordenadas tridimensionales, podemos obtener las componentes del operador gradiente utilizando los cálculos:

$$\nabla = \sum_{k=1}^3 \frac{\mathbf{u}_k}{\sqrt{g_{kk}}} \frac{\partial}{\partial x_k} \quad (A.93)$$

En esta expresión, cada  $\mathbf{u}_k$  representa el vector base unitario en la dirección de la coordenada  $x_k$ , y  $g_{kk}$  son los componentes diagonales del tensor métrico del espacio.

## Operador de Laplace

Podemos utilizar el operador de gradiente y el producto escalar de vectores para formar un operador escalar diferencial denominado *laplaciana* u *operador de Laplace*, que tiene la forma en coordenadas cartesianas:

$$\nabla^2 = \nabla \cdot \nabla = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2} \quad (A.94)$$

El símbolo  $\nabla^2$  se denomina en ocasiones *grado al cuadrado*, *del al cuadrado* o *nabla al cuadrado*. Y en cualquier sistema ortogonal de coordenadas tridimensionales, la laplaciana de una función  $f(x, y, z)$  se calcula como:

$$\nabla^2 f = \frac{1}{\sqrt{g_{11}g_{22}g_{33}}} \left[ \frac{\partial}{\partial x_1} \left( \frac{\sqrt{g_{22}g_{33}}}{\sqrt{g_{11}}} \frac{\partial f}{\partial x_1} \right) + \frac{\partial}{\partial x_2} \left( \frac{\sqrt{g_{33}g_{11}}}{\sqrt{g_{22}}} \frac{\partial f}{\partial x_2} \right) + \frac{\partial}{\partial x_3} \left( \frac{\sqrt{g_{11}g_{22}}}{\sqrt{g_{33}}} \frac{\partial f}{\partial x_3} \right) \right] \quad (A.95)$$

Son muchas las aplicaciones donde surgen ecuaciones en las que interviene la laplaciana, incluyendo la descripción de los efectos de la radiación electromagnética.

## Operador divergencia

El producto escalar de vectores también puede utilizarse para combinar el operador gradiente con una función vectorial con el fin de producir una magnitud escalar denominada *divergencia de un vector*, que tiene la siguiente forma cartesiana:

$$\operatorname{div} \mathbf{V} = \nabla \cdot \mathbf{V} = \frac{\partial V_x}{\partial x} + \frac{\partial V_y}{\partial y} + \frac{\partial V_z}{\partial z} \quad (A.96)$$

En esta expresión,  $V_x$ ,  $V_y$  y  $V_z$  son las componentes cartesianas del vector  $\mathbf{V}$ . La divergencia es una medida de la tasa de incremento o decremento de una función vectorial, como por ejemplo un campo eléctrico, en un determinado punto del espacio. En cualquier sistema ortogonal de coordenadas tridimensionales, la divergencia de un vector  $\mathbf{V}$  se calcula como:

$$\operatorname{div} \mathbf{V} = \nabla \cdot \mathbf{V} = \frac{1}{\sqrt{g_{11}g_{22}g_{33}}} \left[ \frac{\partial}{\partial x_1} (\sqrt{g_{22}g_{33}} V_1) + \frac{\partial}{\partial x_2} (\sqrt{g_{33}g_{11}} V_2) + \frac{\partial}{\partial x_3} (\sqrt{g_{11}g_{22}} V_3) \right] \quad (A.97)$$

con los parámetros  $V_1$ ,  $V_2$  y  $V_3$  como componentes del vector  $\mathbf{V}$  con respecto a los ejes de coordenadas, y siendo  $x_1$ ,  $x_2$  y  $x_3$  y  $g_{kk}$  los elementos diagonales del tensor métrico.

## Operador rotacional

Otro operador diferencial muy útil es el *rotacional de un vector*, que se aplica utilizando el operador gradiente y el producto vectorial. Las componentes cartesianas del rotacional de un vector son:

$$\operatorname{rot} \mathbf{V} = \nabla \times \mathbf{V} = \left( \frac{\partial V_z}{\partial y} - \frac{\partial V_y}{\partial z}, \frac{\partial V_x}{\partial z} - \frac{\partial V_z}{\partial x}, \frac{\partial V_y}{\partial x} - \frac{\partial V_x}{\partial y} \right) \quad (A.98)$$

Esta operación nos da una medida de los efectos rotacionales asociados con una magnitud vectorial, como por ejemplo en la dispersión de la radiación electromagnética. En un sistema ortogonal de coordenadas tridimensionales, podemos expresar las componentes del rotacional en términos de las componentes del tensor métrico, utilizando la siguiente representación en forma de determinante:

$$\operatorname{rot} \mathbf{V} = \nabla \times \mathbf{V} = \frac{1}{\sqrt{g_{11}g_{22}g_{33}}} \begin{vmatrix} \sqrt{g_{11}} \mathbf{u}_1 & \sqrt{g_{22}} \mathbf{u}_2 & \sqrt{g_{33}} \mathbf{u}_3 \\ \frac{\partial}{\partial x_1} & \frac{\partial}{\partial x_2} & \frac{\partial}{\partial x_3} \\ \sqrt{g_{11}} V_1 & \sqrt{g_{22}} V_2 & \sqrt{g_{33}} V_3 \end{vmatrix} \quad (A.99)$$

Los vectores  $\mathbf{u}_k$  son los vectores base unitarios del espacio, y  $g_{kk}$  son los elementos diagonales del tensor métrico.

## A.11 TEOREMAS DE TRANSFORMACIÓN INTEGRALES

---

En muchas aplicaciones, nos encontramos con problemas que implican operaciones diferenciales que es preciso integrar (sumar) en una cierta región del espacio, lo que puede ser a lo largo de un trayecto de línea recta, en toda una superficie o en todo un volumen del espacio. A menudo, el problema puede simplificarse aplicando un teorema de transformación que convierta una integral de superficie en una integral de línea, una integral de línea en una integral de superficie, una integral de volumen en una integral de superficie o una integral de superficie en una integral de volumen. Estos teoremas de transformación tienen una gran importancia a la hora de resolver un amplio rango de problemas prácticos.

### Teorema de Stokes

Para una función vectorial continua  $\mathbf{F}(x, y, z)$  definida sobre una cierta región superficial, el **teorema de Stokes** indica que la integral de la componente perpendicular del rotacional de  $\mathbf{F}$  es igual a la integral de línea de  $\mathbf{F}$  alrededor de la curva de perímetro  $C$  de la superficie. Es decir,

$$\iint_{\text{superf}} (\text{rot } \mathbf{F}) \cdot \mathbf{n} \, dA = \oint_C \mathbf{F} \cdot \mathbf{r} \, ds \quad (A.100)$$

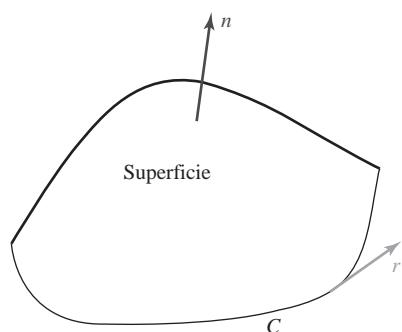
donde la frontera  $C$  debe ser «continua por tramos», lo que significa que  $C$  debe ser una curva continua o una curva compuesta por un número finito de secciones continuas, como por ejemplo arcos circulares o segmentos de línea recta. En esta fórmula,  $\mathbf{n}$  es el vector unitario normal a la superficie en cualquier punto,  $dA$  es un elemento diferencial de área superficial,  $\mathbf{r}$  es un vector unitario tangente a la curva de contorno  $C$  en cualquier punto y  $ds$  es el segmento de línea diferencial a lo largo de  $C$ . La dirección de integración alrededor de  $C$  es en sentido contrario a las agujas del reloj, si estamos mirando a la parte frontal de la superficie (Sección 3.15), como se muestra en la Figura A.23.

### Teorema de Green para una superficie plana

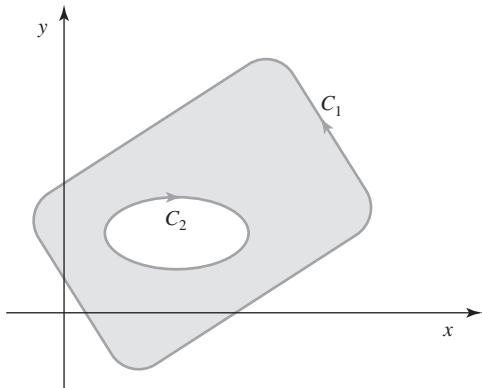
Si consideramos una región en el plano  $xy$  delimitada por una curva  $C$  que sea continua por tramos (como en el teorema de Stokes), podemos enunciar el **teorema de Green para el plano** en forma cartesiana de la manera siguiente:

$$\iint_{\text{area}} \left( \frac{\partial f_2}{\partial x} - \frac{\partial f_1}{\partial y} \right) dx dy = \oint_C (f_1 dx + f_2 dy) \quad (A.101)$$

Aquí,  $f_1(x, y)$  y  $f_2(x, y)$  son dos funciones continuas definidas en toda el área plana delimitada por la curva  $C$ , y la dirección de integración alrededor de  $C$  es en sentido contrario a las agujas del reloj. También pode-



**FIGURA A.23.** La integración alrededor de la línea de contorno  $C$  es en sentido contrario a las agujas del reloj en el teorema de Stokes cuando miramos a la superficie desde la región «exterior» del espacio.



**FIGURA A.24.** Las integrales de línea en el teorema de Green para el plano se evalúan recorriendo las curvas límite  $C_1$  y  $C_2$ , de modo que la región interior (sombreada) queda siempre a la izquierda.

mos aplicar el teorema de Green a una región que tenga agujeros internos, como en la Figura A.24, pero entonces deberemos integrar en el sentido de las agujas alrededor de las curvas de contorno interiores.

Aunque fue desarrollado de manera independiente, el teorema de Green para un plano es un caso especial del teorema de Stokes. Para demostrar esto, vamos a definir la función vectorial  $\mathbf{F}$  con componentes cartesianas  $(f_1, f_2, 0)$ . Entonces, el teorema de Green puede escribirse en forma vectorial:

$$\iint_{\text{area}} (\text{rot } \mathbf{F}) \cdot \mathbf{u}_z \, dA = \oint_C \mathbf{F} \cdot \mathbf{r} \, ds \quad (A.102)$$

donde  $\mathbf{u}_z$  es el vector unitario perpendicular al plano  $xy$  (en la dirección  $z$ ),  $dA = dx \, dy$ , y los otros parámetros son los mismos que en la Ecuación A.100.

Podemos utilizar el teorema de Green para un plano para calcular el área de una región plana, haciendo  $f_1 = 0$  y  $f_2 = x$ . Entonces, a partir del teorema de Green, el área  $A$  de una figura plana será

$$A = \iint_{\text{area}} dx \, dy = \oint_C x \, dy \quad (A.103)$$

De forma similar, si hacemos  $f_1 = -y$  y  $f_2 = 0$ , tendremos:

$$A = \iint_{\text{area}} dx \, dy = -\oint_C y \, dx \quad (A.104)$$

Sumando las dos ecuaciones de área anteriores, se obtiene:

$$A = \frac{1}{2} \oint_C (x \, dy - y \, dx) \quad (A.105)$$

Y también podemos convertir esta expresión cartesiana del área a la siguiente forma en coordenadas polares:

$$A = \frac{1}{2} \oint_C r^2 \, d\theta \quad (A.106)$$

El teorema de Green para el plano puede expresarse de muchas otras formas útiles. Por ejemplo, si definimos  $f_1 = \partial f / \partial y$  y  $f_2 = \partial f / \partial x$ , para alguna función continua  $f$ , tendremos:

$$\iint_{\text{area}} \nabla^2 f \, dx \, dy = \oint_C \frac{\partial f}{\partial n} \, ds \quad (A.107)$$

donde  $\partial f / \partial n$  es la derivada direccional de  $f$  en la dirección de la normal saliente a la curva de contorno  $C$ .

## Teorema de divergencia

Los dos teoremas anteriores nos proporcionan métodos para realizar la conversión entre integrales de superficies e integrales de línea. El **teorema de divergencia** proporciona una ecuación para convertir una integral de volumen en una integral de superficie, o a la inversa. Este teorema también se conoce con varios otros nombres, incluyendo los de **teorema de Green en el espacio** y **teorema de Gauss**. Para una función vectorial continua tridimensional  $\mathbf{F}$ , definida sobre un cierto volumen del espacio, podemos expresar el teorema de divergencia en forma vectorial:

$$\iiint_{\text{vol}} \operatorname{div} \mathbf{F} dV = \iint_{\text{surperf}} \mathbf{F} \cdot \mathbf{n} dA \quad (A.108)$$

donde  $dV$  es un elemento diferencial de volumen,  $\mathbf{n}$  es el vector normal a la superficie de contorno y  $dA$  es un elemento diferencial de área superficial.

Podemos utilizar el teorema de divergencia para obtener varias otras transformaciones integrales útiles. Por ejemplo, si  $\mathbf{F} = \nabla f$  para alguna función tridimensional continua  $f$ , tenemos la versión para volúmenes de la Ecuación A.107, que es:

$$\iiint_{\text{vol}} \nabla^2 f dV = \iint_{\text{surperf}} \frac{\partial f}{\partial n} dA \quad (A.109)$$

En esta ecuación,  $\partial f / \partial n$  es la derivada direccional de  $f$  en la dirección de la normal a la superficie.

A partir del teorema de divergencia, podemos hallar expresiones para calcular el volumen de una región del espacio utilizando una integral de superficie. Dependiendo de cómo representemos la función vectorial  $\mathbf{F}$ , podemos obtener cualquiera de las siguientes formas cartesianas para la integral de superficie:

$$\begin{aligned} V &= \iiint_{\text{vol}} dx dy dz \\ &= \iint_{\text{surperf}} x dy dz = \iint_{\text{surperf}} y dz dx = \iint_{\text{surperf}} z dx dy \\ &= \frac{1}{3} \iint_{\text{surperf}} (x dy dz + y dz dx + z dx dy) \end{aligned} \quad (A.110)$$

## Ecuaciones de transformación de Green

Podemos derivar varias otras transformaciones integrales a partir del teorema de divergencia. Las siguientes dos ecuaciones integrales suelen denominarse **ecuaciones de transformación de Green, primera y segunda fórmulas de Green o identidades de Green**:

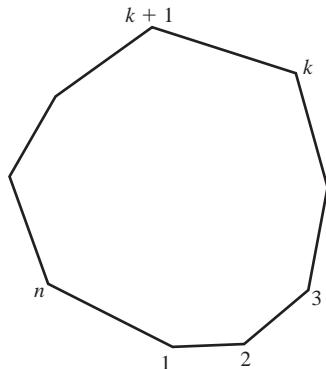
$$\iiint_{\text{vol}} (f_1 \nabla^2 f_2 + \nabla f_1 \cdot \nabla f_2) dV = \iint_{\text{surperf}} f_1 \frac{\partial f_2}{\partial n} dA \quad (A.111)$$

$$\iiint_{\text{vol}} (f_1 \nabla^2 f_2 - f_2 \nabla^2 f_1) dV = \iint_{\text{surperf}} \left( f_1 \frac{\partial f_2}{\partial n} - f_2 \frac{\partial f_1}{\partial n} \right) dA \quad (A.112)$$

En estas ecuaciones,  $f_1$  y  $f_2$  son funciones escalares continuas tridimensionales, y  $\partial f_1 / \partial n$  y  $\partial f_2 / \partial n$  son sus derivadas direccionales en la dirección de la normal a la superficie.

## A.12 ÁREA Y CENTROIDE DE UN POLÍGONO

Podemos utilizar las transformaciones integrales de la Sección A.11 para calcular diversas propiedades de los objetos para aplicaciones infográficas. Para los polígonos, a menudo utilizamos el área y las coordenadas del



**FIGURA A.25.** Un polígono definido con  $n$  vértices en el plano  $xy$ .

centroide en aquellos programas que implican transformaciones geométricas, simulaciones, diseño de sistemas y animaciones.

### Área de un polígono

A partir de la Ecuación A.103, podemos calcular el área de un polígono expresando las coordenadas cartesianas de forma paramétrica y evaluando la integral de línea alrededor del perímetro del polígono. Las ecuaciones paramétricas para los  $n$  lados de un polígono con  $n$  vértices en el plano  $xy$  (Figura A.25) pueden expresarse en la forma:

$$\begin{aligned} x &= x_k + (x_{k+1} - x_k)u & 0 \leq u \leq 1, \quad k = 1, 2, \dots, n \\ y &= y_k + (y_{k+1} - y_k)u \end{aligned} \quad (A.113)$$

donde  $x_{n+1} = x_1$  y  $y_{n+1} = y_1$ .

Sustituyendo la expresión diferencial  $dy = (y_{k+1} - y_k)du$  y la expresión paramétrica de  $x$  en la Ecuación A.103, tenemos:

$$\begin{aligned} A &= \oint_C x \, dy \\ &= \sum_{k=1}^n \int_0^1 [x_k + (x_{k+1} - x_k)u] du \\ &= \sum_{k=1}^n (y_{k+1} - y_k)[x_k + (x_{k+1} - x_k)/2] \\ &= \frac{1}{2} \sum_{k=1}^n (x_k y_{k+1} - x_k y_k + x_{k+1} y_{k+1} + x_{k+1} y_k) \end{aligned} \quad (A.114)$$

Para cada segmento de línea, el segundo y tercer términos de esta suma se cancelan con los términos similares de signos opuestos que podemos encontrar para valores sucesivos de  $k$ . Por tanto, el área del polígono se calcula mediante la fórmula:

$$A = \frac{1}{2} \sum_{k=1}^n (x_k y_{k+1} - x_{k+1} y_k) \quad (A.115)$$

### Centroide de un polígono

Por definición, el centroide es la posición del centro de masas para un objeto de densidad constante (todos los puntos del objeto que tienen la misma masa). Por tanto, las coordenadas del centroide son simplemente los

valores medios de las coordenadas, calculados para todos los puntos contenidos dentro de las fronteras del objeto.

Para algunas formas poligonales simples, podemos obtener el centroide promediando las posiciones de los vértices. Pero, en general, el promediado de vértices no permite localizar correctamente el centroide, porque no se tienen en cuenta las posiciones de los restantes puntos del polígono. Como se ilustra en la Figura A.26, el punto que se obtiene al promediar las coordenadas de los vértices está situado cerca de la zona con mayor concentración de vértices, mientras que el centroide se encuentra en la posición central relativa a toda el área poligonal.

Podemos calcular la posición del centroide  $(\bar{x}, \bar{y})$  de un polígono en el plano  $xy$  promediando las coordenadas de todos los puntos situados dentro de las fronteras del polígono:

$$\begin{aligned}\bar{x} &= \frac{1}{A} \iint_{\text{area}} x dxdy = \frac{\mu_x}{A} \\ \bar{y} &= \frac{1}{A} \iint_{\text{area}} y dxdy = \frac{\mu_y}{A}\end{aligned}\quad (A.116)$$

En estas expresiones,  $\mu_x$  y  $\mu_y$  se denominan momentos del área con respecto a los ejes  $x$  e  $y$ , respectivamente, donde se asume que el área tiene una unidad de masa por unidad de área.

Podemos evaluar cada uno de los momentos del polígono utilizando los mismos procedimientos que ya hemos empleado para calcular el área del polígono. A partir del teorema de Green para un plano, obtenemos una integral de línea equivalente a la integral de área y evaluamos la integral de línea utilizando las representaciones paramétricas de las coordenadas cartesianas que describen los puntos situados en las aristas del polígono. El teorema de Green para una superficie plana (Ecuación A.101) indica que:

$$\iint_{\text{area}} \left( \frac{\partial f_2}{\partial x} - \frac{\partial f_1}{\partial y} \right) dx dy = \oint_C (f_1 dx + f_2 dy) \quad (A.117)$$

Para la evaluación de  $\mu_x$ , podemos hacer  $f_2 = \frac{1}{2}x^2$  y  $f_1 = 0$  en la transformación anterior, de modo que:

$$\mu_x = \iint_{\text{area}} x dxdy = \frac{1}{2} \oint_C x^2 dy \quad (A.118)$$

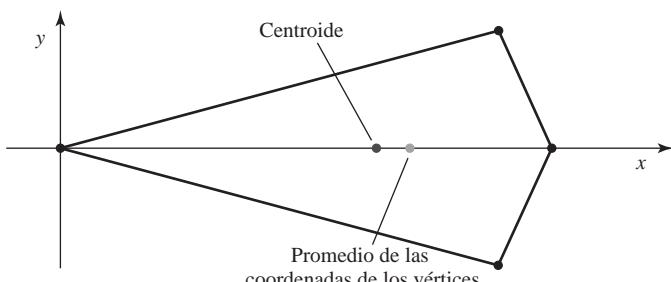
A partir de las representaciones paramétricas A.113 para los lados del polígono, tenemos:

$$x^2 = x_k^2 + 2x_k(x_{k+1} - x_k)u + (x_{k+1} - x_k)^2 u^2$$

y

$$dy = (y_{k+1} - y_k)du$$

para cada uno de los  $n$  lados, etiquetados como  $k = 1, 2, \dots, n$ . Por tanto,



**FIGURA A.26.** Coordenadas del centroide de un polígono y coordenadas obtenidas al promediar las coordenadas de los vértices.

$$\begin{aligned}\mu_x &= \sum_{k=1}^n \frac{y_{k+1} - y_k}{2} \int_0^1 [x_k^2 + 2x_k(x_{k+1} - x_k)u + (x_{k+1} - x_k)^2 u^2] du \\ &= \frac{1}{6} \sum_{k=1}^n (x_{k+1} + x_k)(x_k y_{k+1} - x_{k+1} y_k)\end{aligned}\quad (A.119)$$

Para la evaluación de  $\mu_y$ , hacemos las sustituciones  $f_2 = -\frac{1}{2}y^2$  y  $f_2 = 0$  en el teorema de Green y obtenemos:

$$\mu_y = \iint_{\text{area}} y dx dy = \frac{1}{2} \oint_C y^2 dx \quad (A.120)$$

Utilizando las representaciones paramétricas A.113 para evaluar la integral de línea, tenemos

$$\mu_y = \frac{1}{6} \sum_{k=1}^n (y_{k+1} + y_k)(x_k y_{k+1} - x_{k+1} y_k) \quad (A.121)$$

Dado cualquier conjunto de vértices de un polígono, podemos entonces usar las expresiones correspondientes  $A$ ,  $\mu_x$  y  $\mu_y$  en las Ecuaciones A.116 para calcular las coordenadas del centroide del polígono. Puesto que la expresión  $(x_k y_{k+1} - x_{k+1} y_k)$  aparece en los cálculos de las tres magnitudes,  $A$ ,  $\mu_x$  y  $\mu_y$ , calcularemos dicha expresión una sola vez para cada segmento de línea.

## A.13 CÁLCULO DE LAS PROPIEDADES DE LOS POLIEDROS

---

Para obtener las propiedades de los poliedros se utilizan métodos similares a los de los polígonos. La única diferencia es que ahora calcularemos el volumen espacial, en lugar de un área, y obtendremos el centroide promediando las coordenadas de todos los puntos comprendidos dentro del volumen del poliedro.

El volumen de cualquier región espacial se define en coordenadas cartesianas como:

$$V = \iiint_{\text{vol}} dx dy dz \quad (A.122)$$

Esta integral puede convertirse en una integral de superficie utilizando alguna de las ecuaciones de transformación A.110. Para un poliedro, la integral de superficie puede entonces evaluarse utilizando una representación paramétrica de los puntos situados en cada cara del sólido.

Calculamos las coordenadas del centroide de un poliedro utilizando métodos similares a los que se emplean para los polígonos. Por definición, la posición del centroide en coordenadas cartesianas para una región del espacio (con unidad de masa por unidad de volumen) es el promedio de todos los puntos comprendidos dentro de dicha región:

$$\begin{aligned}\bar{x} &= \frac{1}{V} \iiint_{\text{vol}} x dx dy dz = \frac{\mu_x}{V} \\ \bar{y} &= \frac{1}{V} \iiint_{\text{vol}} y dx dy dz = \frac{\mu_y}{V} \\ \bar{z} &= \frac{1}{V} \iiint_{\text{vol}} z dx dy dz = \frac{\mu_z}{V}\end{aligned}\quad (A.123)$$

De nuevo, podemos convertir las integrales de volumen en integrales de superficie, sustituir las representaciones paramétricas de las coordenadas cartesianas y evaluar las integrales de superficie para las caras del poliedro.

## A.14 MÉTODOS NUMÉRICOS

---

En los algoritmos infográficos, a menudo es necesario resolver sistemas de ecuaciones lineales, de ecuaciones no lineales, de ecuaciones integrales y de otras formas funcionales. Asimismo, para visualizar un conjunto discreto de puntos de datos, puede que resulte útil mostrar una curva continua o una función de superficie continua que se aproximen a los puntos que componen el conjunto de datos. En esta sección, vamos a resumir brevemente algunos algoritmos comunes para la resolución de diversos problemas numéricos.

### Resolución de sistemas de ecuaciones lineales

Para una serie de variables  $x_k$ , con  $k = 1, 2, \dots, n$ , podemos escribir un sistema de  $n$  ecuaciones lineales de la forma siguiente:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n \end{aligned} \quad (A.124)$$

donde los valores de los parámetros  $a_{jk}$  y  $b_j$  son conocidos. Este conjunto de ecuaciones puede expresarse en la forma matricial:

$$\mathbf{A} \mathbf{X} = \mathbf{B} \quad (A.125)$$

donde  $\mathbf{A}$  es una matriz cuadrada  $n$  por  $n$  cuyos elementos son los coeficientes  $a_{jk}$ ,  $\mathbf{X}$  es la matriz columna de valores  $x_j$  y  $\mathbf{B}$  es la matriz columna de valores  $b_j$ . Resolviendo la ecuación matricial para despejar  $\mathbf{X}$ , obtenemos:

$$\mathbf{X} = \mathbf{A}^{-1}\mathbf{B} \quad (A.126)$$

Este sistema de ecuaciones puede resolverse si y sólo si  $\mathbf{A}$  es una matriz no singular, es decir, si su determinante es distinto de cero. En caso contrario, la inversa de la matriz  $\mathbf{A}$  no existe.

Un método para resolver el sistema de ecuaciones es la **regla de Cramer**:

$$x_k = \frac{\det \mathbf{A}_k}{\det \mathbf{A}} \quad (A.127)$$

donde  $\mathbf{A}_k$  es la matriz que se obtiene al sustituir la columna  $k$ -ésima de la matriz  $\mathbf{A}$  por los elementos de  $\mathbf{B}$ . Este método es adecuado para problemas que tengan unas pocas variables. Sin embargo, para más de tres o cuatro variables, el método es extremadamente inefficiente, debido al gran número de multiplicaciones necesarias para calcular cada determinante. La evaluación de un único determinante  $n$  por  $n$  requiere más de  $n!$  multiplicaciones.

Podemos resolver el sistema de ecuaciones de manera más eficiente utilizando variantes de la **eliminación gausiana**. Las ideas básicas de la eliminación gausiana pueden ilustrarse con el siguiente sistema de dos ecuaciones:

$$x_1 + 2x_2 = -4 \quad (A.128)$$

$$3x_1 + 4x_2 = 1$$

Para resolver este sistema de ecuaciones, podemos multiplicar la primera ecuación por  $-3$  y luego sumar las dos ecuaciones para eliminar el término  $x_1$ , lo que nos da la ecuación:

$$-2x_2 = 13$$

que tiene la solución  $x_2 = -13/2$ . Entonces, podemos sustituir este valor en cualquiera de las ecuaciones originales para obtener la solución correspondiente a  $x_1$ , que es 9. Podemos emplear esta técnica básica para

resolver cualquier sistema de ecuaciones lineales, aunque se han desarrollado algoritmos para realizar de forma más eficiente las etapas de eliminación y sustitución.

Una modificación del método de eliminación gausiana es la **descomposición LU** (o **factorización LU**) para la resolución de sistemas de ecuaciones lineales. En este algoritmo, primero factorizamos la matriz **A** en dos matrices, denominadas matriz diagonal inferior **L** y matriz diagonal superior **U** de modo que

$$\mathbf{A} = \mathbf{L} \mathbf{U} \quad (A.129)$$

Todos los elementos de la matriz **L** situados por encima de su diagonal tienen el valor 0, y todos los elementos de la diagonal tienen el valor 1. Todos los elementos de la matriz **U** situados por debajo de la diagonal tienen el valor 0. Entonces, podemos escribir la Ecuación A.125 como:

$$\mathbf{L} \mathbf{U} \mathbf{X} = \mathbf{B} \quad (A.130)$$

Esto nos permite resolver los dos siguientes sistemas de ecuaciones, que son mucho más simples.

$$\mathbf{L} \mathbf{Y} = \mathbf{B}, \quad \mathbf{U} \mathbf{X} = \mathbf{Y} \quad (A.131)$$

Una vez obtenidos los valores de los elementos de la matriz **Y** en la Ecuación A.131, podemos usarlos en el segundo sistema de ecuaciones para calcular los elementos de la matriz **X**. Como ejemplo, la siguiente ecuación ilustra la factorización para una matriz de coeficiente de tamaño 2 por 2:

$$\mathbf{A} = \begin{bmatrix} 2 & 3 \\ 8 & 5 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 4 & 1 \end{bmatrix} \begin{bmatrix} 2 & 3 \\ 0 & -7 \end{bmatrix}$$

Un método para calcular los elementos de las matrices de factorización está dado por el siguiente sistema de ecuaciones, donde  $u_{ij}$  son los elementos de la matriz triangular superior **U** y  $l_{ij}$  son los elementos de la matriz triangular inferior **L**:

$$\begin{aligned} u_{1j} &= a_{1j}, \quad j = 1, 2, \dots, n \\ l_{ii} &= \frac{a_{ii}}{u_{11}}, \quad i = 2, 3, \dots, n \\ u_{ij} &= a_{1j} - \sum_{k=1}^{i-1} l_{ik} u_{kj} \quad j = i, i+1, \dots, n; \quad i \geq 2 \\ l_{ij} &= \frac{1}{u_{jj}} \left( a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \right) \quad i = j+1, j+2, \dots, n; \quad j \geq 2 \end{aligned} \quad (A.132)$$

La eliminación gausiana es susceptible, en ocasiones, a graves errores de redondeo, y otros métodos pueden que no produzcan una solución precisa. En tales casos, puede que podamos obtener una solución utilizando el *método de Gauss-Seidel*. Este método también constituye una forma eficiente de resolver el sistema de ecuaciones lineales cuando conocemos los valores aproximados de la solución. En la técnica de Gauss-Seidel, comenzamos con una «estimación» inicial de los valores de las variables  $x_k$  y luego calculamos repetidamente una serie de aproximaciones sucesivas, hasta que la diferencia entre dos valores sucesivos para cada  $x_k$  sea pequeña. En cada paso, calculamos los valores aproximados de las variables mediante las fórmulas:

$$\begin{aligned} x_1 &= \frac{b_1 - a_{12}x_2 - a_{13}x_3 - \cdots - a_{1n}x_n}{a_{11}} \\ x_2 &= \frac{b_2 - a_{21}x_1 - a_{23}x_3 - \cdots - a_{2n}x_n}{a_{11}} \\ &\vdots \end{aligned} \quad (A.133)$$

Si podemos reordenar la matriz  $\mathbf{A}$  de modo que cada elemento diagonal tenga una magnitud superior a la suma de las magnitudes de los otros elementos situados en esa fila, entonces está garantizado que el método Gauss-Seidel converge hacia una solución.

## Determinación de raíces de ecuaciones no lineales

Una raíz de una función  $f(x)$  es un valor de  $x$  que satisface la ecuación  $f(x) = 0$ . En general, la función  $f(x)$  puede ser una expresión algebraica, como por ejemplo un polinomio, o puede incluir funciones trascendentes. Una expresión algebraica es aquella que sólo contiene los operadores aritméticos, exponentes, raíces y potencias. Las funciones trascendentes, como por ejemplo las funciones trigonométricas, logarítmicas y exponenciales, están representadas por series infinitas de potencias.

Las raíces de una ecuación no lineal pueden ser números reales, números complejos o una combinación de números reales y complejos. En ocasiones, podemos obtener soluciones exactas para todas las raíces, dependiendo de la complejidad de la ecuación. Por ejemplo, sabemos cómo hallar una solución exacta para cualquier polinomio de grado menor o igual que 4, y las raíces de una ecuación trascendente simple tal como  $\sin x = 0$  son  $x = k\pi$  para cualquier valor entero de  $k$ . Pero en la mayoría de los casos de interés práctico, necesitamos aplicar procedimientos numéricos para obtener las raíces de una ecuación no lineal.

Uno de los métodos más populares para hallar las raíces de ecuaciones no lineales es el **algoritmo de Newton-Raphson**. Se trata de un procedimiento iterativo que aproxima  $f(x)$  mediante una función lineal en cada paso de la iteración, como se muestra en la Figura A.27. Comenzamos con una «estimación» inicial  $x_0$  para el valor de la raíz, y luego calculamos la siguiente aproximación a la raíz,  $x_1$ , determinando dónde corta con el eje  $x$  la línea tangente en  $x_0$ . En  $x_0$ , la pendiente (primera derivada) de la curva es:

$$\frac{df}{dx} = \frac{f(x_0)}{x_0 - x_1} \quad (A.134)$$

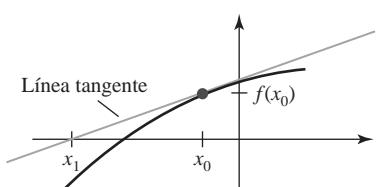
Por tanto, la siguiente aproximación a la raíz es:

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} \quad (A.135)$$

donde  $f'(x_0)$  designa la derivada de  $f(x)$  evaluada en  $x = x_0$ . Repetimos este proceso para cada aproximación que calculemos hasta que la diferencia entre las aproximaciones sucesivas sea “lo suficientemente pequeña”.

Además de resolver los problemas que impliquen variables reales, el algoritmo de Newton-Raphson puede aplicarse a una función de una variable compleja  $f(z)$ , a una función de varias variables y a sistemas de funciones no lineales, reales o complejas. Asimismo, si el algoritmo de Newton-Raphson converge hacia una raíz, convergerá de forma más rápida que cualquier otro método de determinación de raíces. Pero el hecho es que este algoritmo no siempre converge. Por ejemplo, el método falla si la derivada  $f'(x)$  toma el valor 0 en algún punto de la iteración. Asimismo, dependiendo de las oscilaciones de la curva, las aproximaciones sucesivas pueden diverger con respecto a la posición de la raíz.

Otro método, que es más lento pero cuya convergencia está garantizada, es el **método de la biseción**. En este algoritmo, debemos identificar un intervalo  $x$  que contenga una raíz. Entonces, aplicamos un procedimiento de búsqueda binaria dentro de dicho intervalo para ir cercando la raíz en un intervalo cada vez más



**FIGURA A.27.** Aproximación de una curva en un valor inicial  $x_0$  mediante una línea recta que sea tangente a la curva en dicho punto.

pequeño. Primero examinamos el punto medio del intervalo para determinar si la raíz se encuentra en la mitad superior o inferior del mismo. Este procedimiento se repite para cada subintervalo sucesivo hasta que la diferencia entre las posiciones centrales sucesivas sea más pequeño que un cierto valor preestablecido. Puede acelerarse el método interpolando posiciones  $x$  sucesivas en lugar de dividir por la mitad cada subintervalo (*método de la falsa posición*).

## Evaluación de integrales

La integral es un proceso de suma. Para una función de una única variable  $x$ , la integral de  $f(x)$  es igual al área «bajo la curva», como se ilustra en la Figura A.28. Para integrandos simples, podemos determinar a menudo una forma funcional de la integral, pero en general es necesario evaluar las integrales utilizando métodos numéricos.

A partir de la definición de la integral, podemos formar la siguiente aproximación numérica:

$$\int_a^b f(x) dx \approx \sum_{k=1}^n f_k(x) \Delta x_k \quad (A.136)$$

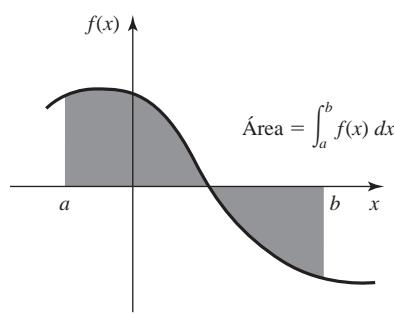
La función  $f_k(x)$  es una aproximación de la  $f(x)$  en el intervalo  $x_k$ . Por ejemplo, podemos aproximar la curva mediante un valor constante en cada subintervalo y sumarle las áreas de los rectángulos resultantes (Figura A.29). Esta aproximación mejora, hasta cierto punto, a medida que reducimos el tamaño de subdivisiones del intervalo que va desde  $a$  hasta  $b$ . Si las subdivisiones son demasiado pequeñas, puede que los valores de las sucesivas áreas rectangulares se pierdan dentro de los errores de redondeo.

Si se utilizan aproximaciones polinómicas para la función en cada subintervalo, generalmente se obtienen mejores resultados que con los rectángulos. Utilizando una aproximación lineal, las subáreas resultantes son trapezoides y el método de aproximación se denomina entonces **regla del trapezoide**. Si utilizamos un polinomio cuadrático (parábola) para aproximar la ecuación en cada subintervalo, el método se denomina **regla de Simpson** y la aproximación de la integral será:

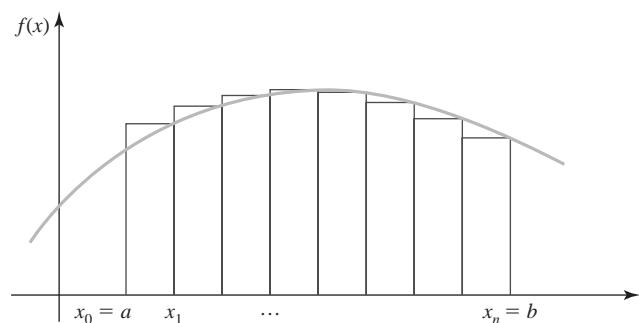
$$\int_a^b f(x) dx \approx \frac{\Delta x}{3} \left[ f(a) + f(b) + 4 \sum_{(impar)k=1}^{n-1} f(x_k) + 2 \sum_{(par)k=2}^{n-2} f(x_k) \right] \quad (A.137)$$

En esta expresión, el intervalo que va desde  $a$  hasta  $b$  se divide en  $n$  intervalos de igual anchura:

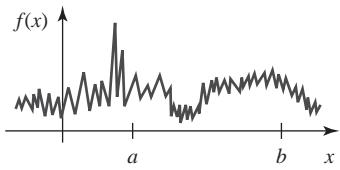
$$\Delta x = \frac{b-a}{n} \quad (A.138)$$



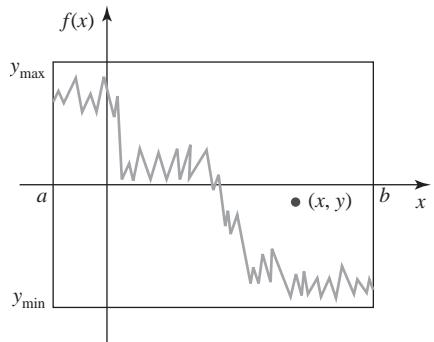
**FIGURA A.28.** La integral de  $f(x)$  es igual al área comprendida entre la función y el eje  $x$  en el intervalo que va desde  $a$  hasta  $b$ .



**FIGURA A.29.** Aproximación de una integral mediante la suma de las áreas de pequeños rectángulos.



**FIGURA A.30.** Una función con oscilaciones de alta frecuencia.



**FIGURA A.31.** Una posición aleatoria  $(x, y)$  generada dentro de un área rectangular que encierra la función  $f(x)$  en el intervalo que va de  $a$  hasta  $b$ .

donde  $n$  es un múltiplo de 2 y donde

$$x_0 = a, \quad x_k = x_{k-1} + \Delta x, \quad k = 1, 2, \dots, n$$

Para una función con una amplitud que varíe de forma muy rápida, como el ejemplo de la Figura A.30, puede que sea difícil aproximar de manera precisa la función en los distintos subintervalos. Asimismo, las integrales múltiples (las que incluyen varias variables de integración) no son fáciles de evaluar mediante la regla de Simpson ni mediante los otros métodos de aproximación. En estos casos, podemos aplicar las técnicas de integración de **Monte Carlo**. El término Monte Carlo se utiliza para describir cualquier método que utilice procedimientos basados en números aleatorios para resolver el problema determinista.

Aplicamos el método Monte Carlo para evaluar una integral generando  $n$  posiciones aleatorias dentro de un área rectangular que contenga  $a f(x)$  en el intervalo que va desde  $a$  hasta  $b$  (Figura A.31). Entonces, podemos calcular una aproximación a la integral mediante la fórmula:

$$\int_b^a f(x) dx \approx h(b-a) \frac{n_{\text{count}}}{n} \quad (A.139)$$

donde el parámetro  $h$  es la altura del rectángulo y el parámetro  $n_{\text{count}}$  es el número de puntos aleatorios comprendidos entre  $f(x)$  y el eje  $x$ . Calculamos una posición aleatoria  $(x, y)$  en la posición rectangular generando primero dos números aleatorios,  $r_1$  y  $r_2$ , y luego realizando las operaciones:

$$h = y_{\max} - y_{\min}, \quad x = a + r_1(b-a), \quad y = y_{\min} + r_2 h \quad (A.140)$$

Pueden aplicarse métodos similares a las integrales múltiples.

En los cálculos de  $x$  e  $y$  en A.140, asumimos que los números aleatorios  $r_1$  y  $r_2$  están distribuidos uniformemente en el intervalo  $(0, 1)$ . Podemos obtener  $r_1$  y  $r_2$  a partir de una función generadora de números aleatorios incluida en una biblioteca matemática o estadística, o bien podemos utilizar el siguiente algoritmo, denominado *generador lineal de congruencias*:

$$i_k = a i_{k-1} + c (\bmod m), \quad k = 1, 2, 3, \dots \quad (A.141)$$

$$r_k = \frac{i_k}{m}$$

donde los parámetros  $a$ ,  $c$ ,  $m$  y  $i_0$  son enteros e  $i_0$  es un valor inicial denominado *semilla*. El parámetro  $m$  se elige lo mayor posible para una máquina concreta, eligiéndose los valores de  $a$  y  $c$  de forma que la cadena de números aleatorios sea lo más larga posible antes de que se repita un valor. Por ejemplo, en una máquina que represente los enteros mediante 32 bits, podemos hacer  $m = 2^{32} - 1$ ,  $a = 1664525$  y  $c = 1013904223$ .

## Resolución de ecuaciones diferenciales ordinarias

Cualquier ecuación que tenga operadores diferenciales se denomina ecuación diferencial. Las magnitudes pueden cambiar sus valores de forma continua desde una posición de coordenada a otra. También pueden cambiar a lo largo del tiempo en cada posición fija y pueden cambiar con respecto a muchos otros parámetros, como por ejemplo la temperatura o la aceleración de giro. Una ecuación que incluya las derivadas de una función de una única variable se denomina *ecuación diferencial ordinaria*. Para resolver una ecuación diferencial podemos determinar una forma funcional que satisfaga la ecuación o podemos emplear métodos de aproximación numérica para determinar los valores de la magnitud en intervalos seleccionados.

Para resolver una ecuación diferencial, también necesitamos conocer uno o más valores iniciales. Una ecuación que sólo incluya la primera derivada de una magnitud, que se denominará *ecuación diferencial de primer orden*, requiere un único valor inicial. Una ecuación que contenga tanto primeras como segundas derivadas, que se denominará *ecuación diferencial de segundo orden*, requiere dos valores iniciales. Y de forma similar para las ecuaciones que incluyan derivadas de orden superior. Hay dos clasificaciones básicas para especificar los valores iniciales. Un **problema de valor inicial** es aquel en el que las condiciones conocidas se especifican para un único valor de la variable independiente. Un **problema de valor de contorno** es aquel en el que las condiciones conocidas se especifican en las fronteras correspondientes a la variable dependiente.

Un ejemplo simple de problema de valor inicial sería la ecuación diferencial de primer orden:

$$\frac{dx}{dt} = f(x, t), \quad x(t_0) = x_0 \quad (A.142)$$

donde  $x$  representa alguna variable dependiente que varía con el tiempo  $t$  (la variable independiente),  $f(x, t)$  es la función conocida de variación con el tiempo para la primera derivada de  $x$  y  $x_0$  es el valor dado de  $x$  para el tiempo inicial  $t_0$ . También podemos escribir esta ecuación en la forma:

$$dx = f(x, t) dt$$

Y, a partir de la definición de la derivada, podemos utilizar intervalos finitos para aproximar los diferenciales de la forma siguiente:

$$\Delta x_k \approx f(x_k, t_k) \Delta t_k, \quad k = 0, 1, \dots, n \quad (A.143)$$

donde  $\Delta x_k = x_{k+1} - x_k$  y  $t_k = t_{k+1} - t_k$ , para  $n$  pasos temporales. Normalmente, tomaremos intervalos de tiempo iguales y utilizaremos los siguientes cálculos incrementales para determinar los valores  $x$  en cada paso temporal, dado el valor de  $x_0$  en  $t_0$ :

$$x_{k+1} = x_k + f(x_k, t_k) \Delta t \quad (A.144)$$

Este procedimiento numérico se denomina **método de Euler** y permite aproximar  $x$  con segmentos de línea recta en cada intervalo temporal  $\Delta t$ .

Aunque el método de Euler es un procedimiento simple de implementar, en general no resulta muy preciso. Es por ello que se han desarrollado mejoras a este algoritmo numérico básico a partir de la siguiente expansión en serie de Taylor, incorporando términos de orden superior en la aproximación de la ecuación diferencial:

$$x(t + \Delta t) = x(t) + x'(t) \Delta t + \frac{1}{2} x''(t) \Delta t^2 + \dots \quad (A.145)$$

Puesto que  $x'(t) = f(x, t)$ , tendremos que  $x''(t) = f'(x, t)$  y así sucesivamente para las derivadas de orden superior.

Un método más preciso y más ampliamente utilizado para evaluar una ecuación diferencial de primer orden es el **algoritmo de Runge-Kutta**, también denominado **algoritmo de Runge-Kutta de cuarto orden**. Este procedimiento se basa en una expansión en serie de Taylor de cuarto orden. El algoritmo para el método de Runge-Kutta es

$$\begin{aligned}
a &= f(x_k, t_k) \Delta t \\
b &= f(x_k + a/2, t_k + \Delta t/2) \Delta t \\
c &= f(x_k + b/2, t_k + \Delta t/2) \Delta t \\
d &= f(x_k + c, t_k + \Delta t/2) \Delta t \\
t_{k+1} &= t_k + \Delta t \\
x_{k+1} &= x_k + (a + 2b + 2c + d)/6, \quad k = 0, 1, \dots, n-1
\end{aligned} \tag{A.146}$$

Podemos aplicar métodos similares para obtener la solución de ecuaciones diferenciales ordinarias de orden superior. La técnica general consiste en utilizar la expansión en serie de Taylor de  $x$  para incluir términos en  $x'', x''',$  etc., dependiendo del orden de la ecuación diferencial. Por ejemplo, de la serie de Taylor podemos obtener la siguiente aproximación para la segunda derivada:

$$x''(t_k) \approx \frac{x_{k+1} - 2x_k + x_{k-1}}{\Delta t^2} \tag{A.147}$$

## Resolución de ecuaciones diferenciales parciales

Como cabría esperar, las ecuaciones diferenciales parciales son, por regla general, más difíciles de resolver que las ecuaciones diferenciales ordinarias, pero podemos aplicar métodos similares y sustituir las derivadas parciales por diferencias finitas.

Vamos a considerar primero una función  $f(x, t)$  que dependa sólo de la coordenada  $x$  y del tiempo  $t$ . Podemos reducir una ecuación diferencial parcial que incluya  $\partial f / \partial x$  y  $\partial f / \partial t$  a una ecuación diferencial ordinaria sustituyendo la derivada espacial por diferencias finitas. Esto nos permite sustituir la función de dos variables por una función de un único subíndice y de una única variable, que será el tiempo:

$$f(x, t) \rightarrow f(x_k, t) \rightarrow f_k(t) \tag{A.148}$$

Las derivadas parciales se sustituyen entonces por las siguientes expresiones:

$$\begin{aligned}
\frac{\partial f}{\partial x} &= \frac{f_{k+1} - f_k}{\Delta x} \\
\frac{\partial f}{\partial t} &= \frac{df_k}{dt}
\end{aligned} \tag{A.149}$$

Después, resolvemos las ecuaciones para un número finito de posiciones  $x$  utilizando las condiciones iniciales o de contorno que se hayan especificado.

Para las derivadas de orden superior, podemos utilizar las expansiones en serie de Taylor para obtener aproximaciones mediante diferencias finitas. Como ejemplo, podemos utilizar la siguiente aproximación para la segunda derivada parcial de  $f$  con respecto a  $x$ :

$$\frac{\partial^2 f(x, t)}{\partial x^2} \rightarrow \frac{f_{k+1}(t) - 2f_k(t) + f_{k-1}(t)}{\Delta x^2} \tag{A.150}$$

Y cuando tengamos funciones definidas sobre superficies o volúmenes del espacio, podemos dividir el espacio en una cuadrícula regular y utilizar diferencias finitas para cada punto del espacio.

Otra técnica que se suele aplicar a las ecuaciones diferenciales parciales es el *método de los elementos finitos*. Se especifica una cuadrícula de puntos en el dominio de interés, que podría ser una superficie o un volumen del espacio, y luego se resuelven las ecuaciones acopladas en las posiciones de nodo utilizando la técnica de variaciones. Según este método, se utiliza una solución funcional aproximada en lugar de las ecuaciones en diferencias finitas. Dependiendo del problema, se especifica una integral para alguna magnitud, como la

energía potencial o el error residual. Entonces, podemos aplicar algún procedimiento, como el análisis de números cuadrados, para minimizar la energía potencial o el error residual. Esta minimización nos proporciona valores para los parámetros desconocidos en la función que aproxima a la solución.

## Métodos de ajuste de curvas por mínimos cuadrados para conjuntos de datos

Cuando una simulación informática o un problema de visualización científica produce un conjunto de valores de datos, normalmente resulta conveniente determinar una forma funcional que describa dicho conjunto. El método estándar para obtener una función que se ajuste a los datos proporcionados es el **algoritmo de los mínimos cuadrados**. Para aplicar este método, primero elegimos un tipo general para la función, como por ejemplo una función lineal, polinómica o exponencial. Después, debemos determinar los valores de los parámetros de la forma funcional que hayamos elegido. Por ejemplo, una función que sea una recta bidimensional puede describirse mediante dos parámetros: la pendiente y el punto de corte con el eje  $y$ . Los parámetros de la función se obtienen minimizando la suma de los cuadrados de las diferencias entre los valores teóricos de la función y los valores reales de los datos.

Para ilustrar este método, vamos a considerar primero un conjunto bidimensional de  $n$  puntos de datos, cuyas coordenadas designamos mediante  $(x_k, y_k)$  con  $k = 1, 2, \dots, n$ . Después de seleccionar la forma funcional  $f(x)$  que podemos utilizar para describir la distribución de datos, establecemos una expresión para la función del error  $E$ , que será la suma de los cuadrados de las diferencias entre  $f(x_k)$  y los valores de datos  $y_k$ :

$$E = \sum_{k=1}^n [y_k - f(x_k)]^2 \quad (A.151)$$

Los parámetros de la función  $f(x)$  se determinan entonces minimizando la expresión de error  $E$ .

Como ejemplo, si queremos describir el conjunto de datos mediante la función lineal:

$$f(x) = a_0 + a_1 x$$

entonces,

$$E = \sum_{k=1}^n [y_k^2 - 2y_k(a_0 + a_1 x_k) + a_0^2 + 2a_0 a_1 x_k + a_1^2 x_k^2] \quad (A.152)$$

Puesto que el error  $E$  es una función de dos variables ( $a_0$  y  $a_1$ ), podremos minimizar  $E$  mediante las siguientes dos ecuaciones acopladas:

$$\begin{aligned} \frac{\partial E}{\partial a_0} &= \sum_{k=1}^n [-2y_k + 2a_0 + 2a_1 x_k] = 0 \\ \frac{\partial E}{\partial a_1} &= \sum_{k=1}^n [-2y_k x_k + 2a_0 x_k + 2a_1 x_k^2] = 0 \end{aligned} \quad (A.153)$$

Entonces, podemos resolver este sistema de dos ecuaciones lineales utilizando la regla de Cramer, que nos da:

$$\begin{aligned} a_0 &= \frac{(\sum_k x_k^2)(\sum_k y_k) - (\sum_k x_k)(\sum_k x_k y_k)}{D} \\ a_1 &= \frac{n \sum_k x_k y_k - (\sum_k x_k)(\sum_k y_k)}{D} \end{aligned} \quad (A.154)$$

donde el denominador en estas dos expresiones es:

$$\begin{aligned}
 D &= \begin{vmatrix} n & \sum_k x_k \\ \sum_k x_k & \sum_k x_k^2 \end{vmatrix} \\
 &= \sum_{k=1}^n x_k^2 - \left( \sum_{k=1}^n x_k \right)^2
 \end{aligned} \tag{A.155}$$

Podemos efectuar cálculos similares para otras funciones. Por ejemplo, para el polinomio:

$$f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

necesitaremos resolver un sistema de  $n$  ecuaciones lineales para determinar los valores de los parámetros  $a_k$ . Y también podemos aplicar el método de ajuste por mínimos cuadrados a funciones de múltiples variables  $f(x_1, x_2, \dots, x_m)$  que pueden ser lineales o no lineales con respecto a cada una de las variables.



# Bibliografía

- AKELEY,K. y T.JERMOLUK(1988). «High-Performance Polygon Rendering», en los artículos de SIGGRAPH'88, *Computer Graphics*, 22(4), págs. 239–246.
- AKELEY, K. (1993). «RealityEngine Graphics», en los artículos de SIGGRAPH'93, *Computer Graphics*, págs. 109–116.
- AKENINE-MÖLLER, T. y E. HAINES (2002). *Real-Time Rendering, Second Edition*, A. K. Peters, Natick, MA.
- AMANATIDES, J. (1984). «Ray Tracing with Cones», en los artículos de SIGGRAPH'84, *Computer Graphics*, 18(3), págs. 129–135.
- ANJKYO, K., Y. USAMI y T. KURIHARA (1992). «A Simple Method for Extracting the Natural Beauty of Hair», en los artículos de SIGGRAPH'92, *Computer Graphics*, 26(2), págs. 111–120.
- APPLE COMPUTER, INC. (1987). *Human Interface Guidelines: The Apple Desktop Interface*, Addison-Wesley, Reading, MA.
- ARVO, J. y D. KIRK (1987). «Fast Ray Tracing by Ray Classification», en los artículos de SIGGRAPH'87, *Computer Graphics*, 21(4), págs. 55–64.
- ARVO, J., ed. (1991). *Graphics Gems II*, Academic Press, San Diego, CA.
- ATHERTON, P. R. (1983). «A Scan-Line Hidden Surface Removal Procedure for Constructive Solid Geometry», en los artículos de SIGGRAPH'83, *Computer Graphics*, 17(3), págs. 73–82.
- BARAFF, D. (1989). «Analytical Methods for Dynamic Simulation of Non-Penetrating Rigid Bodies», en los artículos de SIGGRAPH'89, *Computer Graphics*, 23(3), págs. 223–232.
- BARAFF, D. y A. WITKIN (1992). «Dynamic Simulation of Non-Penetrating Flexible Bodies», en los artículos de SIGGRAPH'92, *Computer Graphics*, 26(2), págs. 303–308.
- BARNESLEY, M. F., A. JACQUIN, F. MALASSENTE, L. REUTER y D. SLOAN (1988). «Harnessing Chaos for Image Synthesis», en los artículos de SIGGRAPH'88, *Computer Graphics*, 22(4), págs. 131–140.
- BARNESLEY,M. F. (1993). *Fractals Everywhere, Second Edition*, Academic Press, San Diego, CA.
- BARNESLEY, M. F. y L. P. HURD, (1993). *Fractal Image Compression*, AK Peters, Wellesly, MA.
- BARR, A. H. (1981). «Superquadrics and Angle-Preserving Transformations», *IEEE Computer Graphics and Applications*, 1(1), págs. 11–23.
- BARSKY, B. A. y J. C. BEATTY (1983). «Local Control of Bias and Tension in Beta-Splines», *ACM Transactions on Graphics*, 2(2), págs. 109–134.
- BARSKY, B. A. (1984). «A Description and Evaluation of Various 3-D Models», *IEEE Computer Graphics and Applications*, 4(1), págs. 38–52.
- BARZEL, R. y A. H. BARR (1988). «A Modeling System Based on Dynamic Constraints», en los artículos de SIGGRAPH'88, *Computer Graphics*, 22(4), págs. 179–188.
- BARZEL, R. (1992). *Physically-Based Modeling for Computer Graphics*, Academic Press, San Diego, CA.
- BAUM, D. R., S. MANN, K. P. SMITH y J. M. WINGET (1991). «Making Radiosity Usable: Automatic Preprocessing and Meshing Techniques for the Generation of Accurate Radiosity Solutions», en los artículos de SIGGRAPH'91, *Computer Graphics*, 25(4), págs. 51–61.
- BERGMAN, L. D., J. S. RICHARDSON, D. C. RICHARDSON y F. P. BROOKS, JR. (1993). «VIEW—an Exploratory Molecular Visualization System with User-Definable Interaction Sequences», en los artículos de

- SIGGRAPH'93, *Computer Graphics*, págs. 117–126.
- BÉZIER, P. (1972). *Numerical Control: Mathematics and Applications*, translated by A. R. Forrest and A. F. Pankhurst, John Wiley & Sons, Londres.
- BIRN, J. (2000). *[digital] Lighting & Rendering*, New Riders Publishing, Indianapolis, IN.
- BISHOP, G. y D. M. WIEMER (1986). «Fast Phong Shading», en los artículos de SIGGRAPH'86, *Computer Graphics*, 20(4), págs. 103–106.
- BLAKE, J. W. (1993). *PHIGS and PHIGS Plus*, Academic Press, Londres.
- BLESER, T. (1988). «TAE Plus Styleguide User Interface Description», NASA Goddard Space Flight Center, Greenbelt, MD.
- BLINN, J. F. y M. E. NEWELL (1976). «Texture and Reflection in Computer-Generated Images», *Communications of the ACM*, 19(10), págs. 542–547.
- BLINN, J. F. (1977). «Models of Light Reflection for Computer-Synthesized Pictures», en los artículos de SIGGRAPH'77, *Computer Graphics*, 11(2), págs. 192–198.
- BLINN, J. F. y M. E. NEWELL (1978). «Clipping Using Homogeneous Coordinates», en los artículos de SIGGRAPH'78, *Computer Graphics*, 12(3), págs. 245–251.
- BLINN, J. F. (1978). «Simulation of Wrinkled Surfaces», en los artículos de SIGGRAPH'78, *Computer Graphics*, 12(3), págs. 286–292.
- BLINN, J. F. (1982). «A Generalization of Algebraic Surface Drawing», *ACM Transactions on Graphics*, 1(3), págs. 235–256.
- BLINN, J. F. (1982). «Light Reflection Functions for Simulation of Clouds and Dusty Surfaces», en los artículos de SIGGRAPH'82, *Computer Graphics*, 16(3), págs. 21–29.
- BLINN, J. F. (1993). «A Trip Down the Graphics Pipeline: The Homogeneous Perspective Transform», *IEEE Computer Graphics and Applications*, 13(3), págs. 75–80.
- BLINN, J. (1996). *Jim Blinn's Corner: A Trip Down the Graphics Pipeline*, Morgan Kauffman, San Francisco, CA.
- BLINN, J. (1998). *Jim Blinn's Corner: Dirty Pixels*, Morgan Kauffman, San Francisco, CA.
- BLOOMENTHAL, J. (1985). «Modeling the Mighty Maple», en los artículos de SIGGRAPH'85, *Computer Graphics*, 19(3), págs. 305–312.
- BONO, P. R., J. L. ENCARNACAO, F. R. A. HOPGOOD, et al. (1982). «GKS: The First Graphics Standard», *IEEE Computer Graphics and Applications*, 2(5), págs. 9–23.
- BOUQUET, D. L. (1978). «An Interactive Graphics Application to Advanced Aircraft Design», en los artículos de SIGGRAPH'78, *Computer Graphics*, 12(3), págs. 330–335.
- BOURG, D. M. (2002). *Physics for Game Developers*, O'Reilly & Associates, Sebastopol, CA.
- BRESENHAM, J. E. (1965). «Algorithm for Computer Control of a Digital Plotter», *IBM Systems Journal*, 4(1), págs. 25–30.
- BRESENHAM, J. E. (1977). «A Linear Algorithm for Incremental Digital Display of Circular Arcs», *Communications of the ACM*, 20(2), págs. 100–106.
- BROOKSJR., F. P. (1986). «Walkthrough: A Dynamic Graphics System for Simulating Virtual Buildings», *Interactive 3D*.
- BROOKS JR., F. P. (1988). «Grasping Reality Through Illusion: Interactive Graphics Serving Science», *CHI '88*, págs. 1–11.
- BROOKS JR., F. P., M. OUH-YOUNG, J. J. BATTER y P. J. KILPATRICK (1990). «Project GROPE—Haptic Display for Scientific Visualization», en los artículos de SIGGRAPH'90, *Computer Graphics*, 24(4), págs. 177–185.
- BROWN, J. R. y S. CUNNINGHAM (1989). *Programming the User Interface*, John Wiley & Sons, Nueva York.
- BROWN, C. W. y B. J. SHEPHERD (1995). *Graphics File Formats*, Manning Publications, Greenwich, CT.
- BRUDERLIN, A. y T. W. CALVERT (1989). «Goal-Directed, Dynamic Animation of Human Walking», en los artículos de SIGGRAPH'89, *Computer Graphics*, 23(3), págs. 233–242.
- BRUNET, P. e I. NAVAZO (1990). «Solid Representation and Operation Using Extended Octrees», *ACM Transactions on Graphics*, 9(2), págs. 170–197.

- BRYSON, S. y C. LEVIT (1992). «The Virtual Wind Tunnel», *IEEE Computer Graphics and Applications*, 12(4), págs. 25–34.
- CALVERT, T., A. BRUDERLIN, J. DILL, T. SCHIPHORST y C. WEILMAN (1993). «Desktop Animation of Multiple Human Figures», *IEEE Computer Graphics and Applications*, 13(3), págs. 18–26.
- CAMBELL, G., T. A. DEFANTI, J. FREDERIKSEN, S. A. JOYCE y L. A. LESKE (1986). «Two Bit/Pixel Full-Color Encoding», en los artículos de SIGGRAPH'86, *Computer Graphics*, 20(4), págs. 215–224.
- CARPENTER, L. (1984). «The A-Buffer: An Antialiased Hidden-Surface Method», en los artículos de SIGGRAPH'84, *Computer Graphics*, 18(3), págs. 103–108.
- CHEN, S. E., H. E. RUSHMEIER, G. MILLER y D. TURNER (1991). «A Progressive Multi-Pass Method for Global Illumination», en los artículos de SIGGRAPH'91, *Computer Graphics*, 25(4), págs. 165–174.
- CHIN, N. y S. FEINER (1989). «Near Real-Time Shadow Generation Using BSP Trees», en los artículos de SIGGRAPH'89, *Computer Graphics*, 23(3), págs. 99–106.
- CHUNG, J. C., et al. (1989). «Exploring Virtual Worlds with Head-Mounted Visual Displays», *Artículos de SPIE (Society of Photo-Optical Instrumentation Engineers, ahora denominada International Society for Optical Engineering) Conference on Three-Dimensional Visualization and Display Technologies*, 1083, enero de 1989, págs. 15–20.
- COHEN, M. F. y D. P. GREENBERG (1985). «The Hemi-Cube: A Radiosity Solution for Complex Environments», en los artículos de SIGGRAPH'85, *Computer Graphics*, 19(3), págs. 31–40.
- COHEN, M. F., S. E. CHEN, J. R. WALLACE y D. P. GREENBERG (1988). «A Progressive Refinement Approach to Fast Radiosity Image Generation», en los artículos de SIGGRAPH'88, *Computer Graphics*, 22(4), págs. 75–84.
- COHEN, M. F. y J. R. WALLACE (1993). *Radiosity and Realistic Image Synthesis*, Academic Press, Boston, MA.
- COOK, R. L. y K. E. TORRANCE (1982). «A Reflectance Model for Computer Graphics», *ACM Transactions on Graphics*, 1(1), págs. 7–24.
- COOK, R. L., T. PORTER y L. CARPENTER (1984). «Distributed Ray Tracing», en los artículos de SIGGRAPH'84, *Computer Graphics*, 18(3), págs. 137–145.
- COOK, R. L. (1984). «Shade Trees», en los artículos de SIGGRAPH'84, *Computer Graphics*, 18(3), págs. 223–231.
- COOK, R. L. (1986). «Stochastic Sampling in Computer Graphics», *ACM Transactions on Graphics*, 6(1), págs. 51–72.
- COOK, R. L., L. CARPENTER y E. CATMULL (1987). «The Reyes Image Rendering Architecture», en los artículos de SIGGRAPH'87, *Computer Graphics*, 21(4), págs. 95–102.
- COQUILLART, S. y P. JANCENE (1991). «Animated Free-Form Deformation: An Interactive Animation Technique», en los artículos de SIGGRAPH'91, *Computer Graphics*, 25(4), págs. 23–26.
- CROW, F. C. (1977a). «The Aliasing Problem in Computer-Synthesized Shaded Images», *Communications of the ACM*, 20(11), págs. 799–805.
- CROW, F. C. (1977b). «Shadow Algorithms for Computer Graphics», en los artículos de SIGGRAPH'77, *Computer Graphics*, 11(2), págs. 242–248.
- CROW, F. C. (1978). «The Use of Grayscale for Improved Raster Display of Vectors and Characters», en los artículos de SIGGRAPH'78, *Computer Graphics*, 12(3), págs. 1–5.
- CROW, F. C. (1981). «A Comparison of Antialiasing Techniques», *IEEE Computer Graphics and Applications*, 1(1), págs. 40–49.
- CROW, F. C. (1982). «A More Flexible Image Generation Environment», en los artículos de SIGGRAPH'82, *Computer Graphics*, 16(3), págs. 9–18.
- CRUZ-NEIRA, C., D. J. SANDIN y T. A. DEFANTI (1993). «Surround-Screen Projection-Based Virtual Reality: The Design and Implementation of the CAVE», en los artículos de SIGGRAPH'93, *Computer Graphics*, págs. 135–142.

- CUNNINGHAM, S., N. K. CRAIGHILL, M. W. FONG, J. BROWN y J. R. BROWN, eds. (1992). *Computer Graphics Using Object-Oriented Programming*, John Wiley & Sons, Nueva York.
- CUTLER, E., D. GILLY y T. O'REILLY, eds. (1992). *The X Window System in a Nutshell*, Second Edition, O'Reilly & Assoc., Inc., Sebastopol, CA.
- CYRUS, M. y J. BECK (1978). «Generalized Two- and Three-Dimensional Clipping», *Computers and Graphics*, 3(1), págs. 23–28.
- DAY, A. M. (1990). «The Implementation of an Algorithm to Find the Convex Hull of a Set of Three-Dimensional Points», *ACM Transactions on Graphics*, 9(1), págs. 105–132.
- DEERING, M. (1992). «High Resolution Virtual Reality», en los artículos de SIGGRAPH'92, *Computer Graphics*, 26(2), págs. 195–202.
- DEERING, M. F. y S. R. NELSON (1993). «Leo: A System for Cost-Effective 3D Shaded Graphics», en los artículos de SIGGRAPH'93, *Computer Graphics*, págs. 101–108.
- DEMERS, O. (2002). *[digital] Texturing & Painting*, New Riders Publishing, Indianapolis, IN.
- DEPP, S. W. y W. E. HOWARD (1993). «Flat-Panel Displays», *Scientific American*, 266(3), págs. 90–97.
- DE REFFYE, P., C. EDELIN, J. FRANÇON, M. JAEGER y C. PUECH (1988). «Plant Models Faithful to Botanical Structure and Development», en los artículos de SIGGRAPH'88, *Computer Graphics*, 22(4), págs. 151–158.
- DEROSE, T. D. (1988). «Geometric Continuity, Shape Parameters y Geometric Constructions for Catmull-Rom Splines», *ACM Transactions on Graphics*, 7(1), págs. 1–41.
- DIGITAL EQUIPMENT CORP. (1989). «Digital Equipment Corporation XUI Style Guide», Maynard, MA.
- DOCTOR, L. J. y J. G. TORBERG (1981). «Display Techniques for Octree-Encoded Objects», *IEEE Computer Graphics and Applications*, 1(3), págs. 29–38.
- DORSEY, J. O., F. X. SILLION y D. P. GREENBERG (1991). «Design and Simulation of Opera Lighting and Projection Effects», en los artículos de SIGGRAPH'91, *Computer Graphics*, 25(4), págs. 41–50.
- DREBIN, R. A., L. CARPENTER y P. HANRAHAN (1988). «Volume Rendering», en los artículos de SIGGRAPH'88, *Computer Graphics*, 22(4), págs. 65–74.
- DURRETT, H. J., ed. (1987). *Color and the Computer*, Academic Press, Boston.
- DUVANENKO, V. (1990). «Improved Line-Segment Clipping», *Dr. Dobb's Journal*, julio de 1990.
- DYER, S. (1990). «A Dataflow Toolkit for Visualization», *IEEE Computer Graphics and Applications*, 10(4), págs. 60–69.
- EARNSHAW, R. A., ed. (1985). *Fundamental Algorithms for Computer Graphics*, Springer-Verlag, Berlin.
- EDELSBRUNNER, H. (1987). *Algorithms in Computational Geometry*, Springer-Verlag, Berlin.
- EDELSBRUNNER, H. y E. P. MUCKE (1990). «Simulation of Simplicity: A Technique to Cope with Degenerate Cases in Geometric Algorithms», *ACM Transactions on Graphics*, 9(1), págs. 66–104.
- ELBER, G. y E. COHEN (1990). «Hidden-Curve Removal for Free-Form Surfaces», en los artículos de SIGGRAPH'90, *Computer Graphics*, 24(4), págs. 95–104.
- ENDERLE, G., K. KANSY y G. PFAFF (1984). *Computer Graphics Programming: GKS—The Graphics Standard*, Springer-Verlag, Berlin.
- FARIN, G. (1988). *Curves and Surfaces for Computer-Aided Geometric Design*, Academic Press, Boston, MA.
- FARIN, G. y D. HANSFORD (1998). *The Geometry Toolbox for Graphics and Modeling*, A. K. Peters, Natick, MA.
- FEDER, J. (1988). *Fractals*, Plenum Press, Nueva York.
- FEYNMAN, R. P., R. B. LEIGHTON y M. L. SANDS (1989). *The Feynman Lectures on Physics*, Addison-Wesley, Reading, MA.
- FISHKIN, K. P. y B. A. BARSKY (1984). «A Family of New Algorithms for Soft Filling», en los artículos de SIGGRAPH'84, *Computer Graphics*, 18(3), págs. 235–244.

- FIUME, E. L. (1989). *The Mathematical Structure of Raster Graphics*, Academic Press, Boston.
- FOLEY, J. D., A. VANDAM, S. K. FEINER y J. F. HUGHES (1990). *Computer Graphics: Principles and Practice, Second Edition*, Addison-Wesley, Reading, MA.
- FOURNIER, A., D. FUSSEL y L. CARPENTER (1982). «Computer Rendering of Stochastic Models», *Communications of the ACM*, 25(6), págs. 371–384.
- FOURNIER, A. y W. T. REEVES (1986). «A Simple Model of Ocean Waves», en los artículos de SIGGRAPH'86, *Computer Graphics*, 20(4), págs. 75–84.
- FOWLER, D. R., H. MEINHARDT y P. PRUSINKIEWICZ (1992). «Modeling Seashells», en los artículos de SIGGRAPH'92, *Computer Graphics*, 26(2), págs. 379–387.
- FRANKLIN, W. R. y M. S. KANKANHALLI (1990). «Parallel Object-Space Hidden Surface Removal», en los artículos de SIGGRAPH'90, *Computer Graphics*, 24(4), págs. 87–94.
- FREEMAN, H. ed. (1980). *Tutorial and Selected Readings in Interactive Computer Graphics*, IEEE Computer Society Press, Silver Springs, MD.
- FRENKEL, K. A. (1989). «Volume Rendering», *Communications of the ACM*, 32(4), págs. 426–435.
- FRIEDER, G., D. GORDON y R. A. REYNOLD (1985). «Back-to-Front Display of Voxel-Based Objects», *IEEE Computer Graphics and Applications*, 5(1), págs. 52–60.
- FRIEDHOFF, R. M. y W. BENZON (1989). *The Second Computer Revolution: Visualization*, Harry N. Abrams, Nueva York.
- FU, K. S. y A. ROSENFIELD (1984). «Pattern Recognition and Computer Vision», *Computer*, 17(10), págs. 274–282.
- FUJIMOTO, A. y K. IWATA (1983). «Jag-Free Images on Raster Displays», *IEEE Computer Graphics and Applications*, 3(9), págs. 26–34.
- FUNKHOUSER, T. A. y C. H. SEQUIN (1993). «Adaptive Display Algorithms for Interactive Frame Rates during Visualization of Complex Virtual Environments», en los artículos de SIGGRAPH'93, *Computer Graphics*, págs. 247–254.
- GARDNER, T. N. y H. R. NELSON (1983). «Interactive Graphics Developments in Energy Exploration», *IEEE Computer Graphics and Applications*, 3(2), págs. 33–34.
- GARDNER, G. Y. (1985). «Visual Simulation of Clouds», en los artículos de SIGGRAPH'85, *Computer Graphics*, 19(3), págs. 297–304.
- GASCUEL, M.-P. (1993). «An Implicit Formulation for Precise Contact Modeling between Flexible Solids», en los artículos de SIGGRAPH'93, *Computer Graphics*, págs. 313–320.
- GASKINS, T. (1992). *PHIGS Programming Manual*, O'Reilly & Associates, Sebastopol, CA.
- GHARACHORLOO, N., S. GUPTA, R. F. SPROULL y I. E. SUTHERLAND (1989). «A Characterization of Ten Rasterization Techniques», en los artículos de SIGGRAPH'89, *Computer Graphics*, 23(3), págs. 355–368.
- GIRARD, M. (1987). «Interactive Design of 3D Computer-Animated Legged Animal Motion», *IEEE Computer Graphics and Applications*, 7(6), págs. 39–51.
- GLASSNER, A. S. (1984). «Space Subdivision for Fast Ray Tracing», *IEEE Computer Graphics and Applications*, 4(10), págs. 15–22.
- GLASSNER, A. S. (1986). «Adaptive Precision in Texture Mapping», en los artículos de SIGGRAPH'86, *Computer Graphics*, 20(4), págs. 297–306.
- GLASSNER, A. S. (1988). «Spacetime Ray Tracing for Animation», *IEEE Computer Graphics and Applications*, 8(2), págs. 60–70.
- GLASSNER, A. S., ed. (1989a). *An Introduction to Ray Tracing*, Academic Press, San Diego, CA.
- GLASSNER, A. S. (1989b). *3D Computer Graphics: A User's Guide for Artists and Designers, Second Edition*, Design Books, Lyons & Bufford Publishers, Nueva York.
- GLASSNER, A. S., ed. (1990). *Graphics Gems*, Academic Press, San Diego, CA.
- GLASSNER, A. S. (1992). «Geometric Substitution: A Tutorial», *IEEE Computer Graphics and Applications*, 12(1), págs. 22–36.
- GLASSNER, A. S. (1995). *Principles of Digital Image Synthesis, Vols. 1–2*, Morgan Kaufmann, San Francisco, CA.

- GLASSNER, A. S. (1999). *Andrew Glassner's Notebook: Recreational Computer Graphics*, Morgan Kaufmann, San Francisco, CA.
- GLASSNER, A. S. (2002). *Andrew Glassner's Other Notebook: Further Recreations in Computer Graphics*, A. K. Peters, Natick, MA.
- GLEICHER, M. y A. WITKIN (1992). «Through-the-Lens Camera Control», en los artículos de SIGGRAPH'92, *Computer Graphics*, 26(2), págs. 331–340.
- GOLDSMITH, J. y J. SALMON (1987). «Automatic Creation of Object Hierarchies for Ray Tracing», *IEEE Computer Graphics and Applications*, 7(5), págs. 14–20.
- GOMES, J., L. DARSA, B. COSTA y L. VELHO (1999). *Warping and Morphing of Graphical Objects*, Morgan Kaufmann, San Francisco, CA.
- GONZALEZ, R. C. y P. WINTZ (1987). *Digital Image Processing*, Addison-Wesley, Reading, MA.
- GOOCH, B. y A. GOOCH (2001). *Non-Photorealistic Rendering*, A. K. Peters, Natick, MA.
- GORAL, C. M., K. E. TORRANCE, D. P. GREENBERG y B. BATTAILLE (1984). «Modeling the Interaction of Light Between Diffuse Surfaces», en los artículos de SIGGRAPH'84, *Computer Graphics*, 18(3), págs. 213–222.
- GORDON, D. y S. CHEN (1991). «Front-to-Back Display of BSP Trees», *IEEE Computer Graphics and Applications*, 11(5), págs. 79–85.
- GORTLER, S. J., P. SCHRÖDER, M. F. COHEN y P. HANRAHAN (1993). «Wavelet Radiosity», en los artículos de SIGGRAPH'93, *Computer Graphics*, págs. 221–230.
- GOURAUD, H. (1971). «Continuous Shading of Curved Surfaces», *IEEE Transactions on Computers*, C-20(6), págs. 623–628.
- GREENE, N., M. KASS y G. MILLER (1993). «Hierarchical Z-Buffer Visibility», en los artículos de SIGGRAPH'93, *Computer Graphics*, págs. 231–238.
- GROTCHE, S. L. (1983). «Three-Dimensional and Stereoscopic Graphics for Scientific Data Display and Analysis», *IEEE Computer Graphics and Applications*, 3(8), págs. 31–43.
- HAECKERLI, P. y K. AKELEY (1990). «The Accumulation Buffer: Hardware Support for High-Quality Rendering», en los artículos de SIGGRAPH'90, *Computer Graphics*, 24(4), págs. 309–318.
- HALL, R. A. y D. P. GREENBERG (1983). «A Testbed for Realistic Image Synthesis», *IEEE Computer Graphics and Applications*, 3(8), págs. 10–20.
- HALL, R. (1989). *Illumination and Color in Computer Generated Imagery*, Springer-Verlag, Nueva York.
- HALLIDAY, D., R. RESNICK y J. WALKER (2000). *Fundamentals of Physics, Sixth Edition*, John Wiley & Sons, Nueva York.
- HANRAHAN, P. y J. LAWSON (1990). «A Language for Shading and Lighting Calculations», en los artículos de SIGGRAPH'90, *Computer Graphics*, 24(4), págs. 289–298.
- HARDY, V. J. (2000). *Java 2D API Graphics*, Sun Microsystems Press, Palo Alto, CA.
- HART, J. C., D. J. SANDIN y L. H. KAUFFMAN (1989). «Ray Tracing Deterministic 3D Fractals», en los artículos de SIGGRAPH'89, *Computer Graphics*, 23(3), págs. 289–296.
- HART, J. C. y T. A. DEFANTI (1991). «Efficient Antialiased Rendering of 3-D Linear Fractals», en los artículos de SIGGRAPH'91, *Computer Graphics*, 25(4), págs. 91–100.
- HAWRYLYSHYN, P. A., R. R. TASKER y L. W. ORGAN (1977). «CASS: Computer-Assisted Stereotaxic Surgery», en los artículos de SIGGRAPH'77, *Computer Graphics*, 11(2), págs. 13–17.
- HE, X. D., P. O. HEYNEN, R. L. PHILLIPS, K. E. TORRANCE, D. H. SALESIN y D. P. GREENBERG (1992). «A Fast and Accurate Light Reflection Model», en los artículos de SIGGRAPH'92, *Computer Graphics*, 26(2), págs. 253–254.
- HEARN, D. y P. BAKER (1991). «Scientific Visualization: An Introduction», *Eurographics '91 Technical Report Series*, Tutorial Lecture 6, Viena, Austria.
- HECKBERT, P. S. (1982). «Color Image Quantization for Frame Buffer Display», en los artículos de SIGGRAPH'82, *Computer Graphics*, 16(3), págs. 297–307.
- HECKBERT, P. S. y P. HANRAHAN (1984). «Beam Tracing Polygonal Objects», en los

- artículos de SIGGRAPH'84, *Computer Graphics*, 18(3), págs. 119–127.
- HECKBERT, P. S., ed. (1994). *Graphics Gems IV*, Academic Press Professional, Cambridge, MA.
- HENDERSON, L. R. y A. M. MUMFORD (1993). *The CGM Handbook*, Academic Press, San Diego, CA.
- HOPGOOD, F. R. A., D. A. DUCE, J. R. GALLOP y D. C. SUTCLIFFE (1983). *Introduction to the Graphical Kernel System (GKS)*, Academic Press, Londres.
- HOPGOOD, F. R. A. y D. A. DUCE (1991). *A Primer for PHIGS*, John Wiley & Sons, Chichester, Inglaterra.
- HORSTMANN, C. S. y G. CORNELL (2001). *Core Java 2, Vols. I-II*, Sun Microsystems Press, Palo Alto, CA.
- HOWARD, T. L. J., W. T. HEWITT, R. J. HUBBOLD y K. M. WYRWAS (1991). *A Practical Introduction to PHIGS and PHIGS Plus*, Addison-Wesley, Wokingham, Inglaterra.
- HUFFMAN, D. A. (1952). «A Method for the Construction of Minimum-Redundancy Codes», *Communications of the ACM*, 40(9), págs. 1098–1101.
- HUGHES, J. F. (1992). «Scheduled Fourier Volume Morphing», en los artículos de SIGGRAPH'92, *Computer Graphics*, 26(2), págs. 43–46.
- HUITRIC, H. y M. NAHAS (1985). «B-Spline Surfaces: A Tool for Computer Painting», *IEEE Computer Graphics and Applications*, 5(3), págs. 39–47.
- IMMEL, D. S., M. F. COHEN y D. P. GREENBERG (1986). «A Radiosity Method for Non-Diffuse Environments», en los artículos de SIGGRAPH'86, *Computer Graphics*, 20(4), págs. 133–142.
- ISAACS, P. M. y M. F. COHEN (1987). «Controlling Dynamic Simulation with Kinematic Constraints, Behavior Functions y Inverse Dynamics», en los artículos de SIGGRAPH'87, *Computer Graphics*, 21(4), págs. 215–224.
- JARVIS, J. F., C. N. JUDICE y W. H. NINKE (1976). «A Survey of Techniques for the Image Display of Continuous Tone Pictures on Bilevel Displays», *Computer Graphics and Image Processing*, 5(1), págs. 13–40.
- JENSEN, H. W. (2001). *Realistic Image Synthesis Using Photon Mapping*, A. K. Peters, Natick, MA.
- JOHNSON, S. A. (1982). «Clinical Varifocal Mirror Display System at the University of Utah», *Proceedings of SPIE*, 367, agosto de 1982, págs. 145–148.
- KAJIYA, J. T. (1983). «New Techniques for Ray Tracing Procedurally Defined Objects», *ACM Transactions on Graphics*, 2(3), págs. 161–181.
- KAJIYA, J. T. (1986). «The Rendering Equation», en los artículos de SIGGRAPH'86, *Computer Graphics*, 20(4), págs. 143–150.
- KAJIYA, J. T. y T. L. KAY (1989). «Rendering Fur with Three-Dimensional Textures», en los artículos de SIGGRAPH'89, *Computer Graphics*, 23(3), págs. 271–280.
- KAPPEL, M. R. (1985). «An Ellipse-Drawing Algorithm for Faster Displays», in *Fundamental Algorithms for Computer Graphics*, Springer-Verlag, Berlín, págs. 257–280.
- KAY, T. L. y J. T. KAJIYA (1986). «Ray Tracing Complex Scenes», en los artículos de SIGGRAPH'86, *Computer Graphics*, 20(4), págs. 269–278.
- KAY, D. C. y J. R. LEVINE (1992). *Graphics File Formats*, Windcrest/McGraw-Hill, Nueva York.
- KELLEY, A. D., M. C. MALIN y G. M. NIELSON (1988). «Terrain Simulation Using a Model of Stream Erosion», en los artículos de SIGGRAPH'88, *Computer Graphics*, 22(4), págs. 263–268.
- KENT, J. R., W. E. CARLSON y R. E. PARENT (1992). «Shape Transformation for Polyhedral Objects», en los artículos de SIGGRAPH'92, *Computer Graphics*, 26(2), págs. 47–54.
- KERLOW, I. V. (2000). *The Art of 3-D Computer Animation and Imaging*, John Wiley & Sons, Nueva York.
- KILGARD, M. J. (1996). *OpenGL Programming for the X Window System*, Addison-Wesley, Reading, MA.
- KIRK, D. y J. ARVO (1991). «Unbiased Sampling Techniques for Image Synthesis», en los artículos de SIGGRAPH'91, *Computer Graphics*, 25(4), págs. 153–156.
- KIRK, D., ed. (1992). *Graphics Gems III*, Academic Press, San Diego, CA.

- KNUDSEN, J. (1999). *Java 2D Programming*, O'Reilly & Associates, Sebastopol, CA.
- KNUTH,D. E. (1987). «Digital Halftones by Dot Diffusion», *ACM Transactions on Graphics*, 6(4), págs. 245–273.
- KOCHANEK, D. H. U. y R. H. BARTELS (1984). «Interpolating Splines with Local Tension, Continuity y Bias Control», en los artículos de SIGGRAPH'84, *Computer Graphics*, 18(3), págs. 33–41.
- KOH, E.-K. y D. HEARN (1992). «Fast Generation and Surface Structuring Methods for Terrain and Other Natural Phenomena», en los artículos de Eurographics'92, *Computer Graphics Forum*, 11(3), págs. C169–180.
- KORIEN, J. U. y N. I. BADLER (1982). «Techniques for Generating the Goal-Directed Motion of Articulated Structures», *IEEE Computer Graphics and Applications*, 2(9), págs. 71–81.
- KORIEN, J. U. y N. I. BADLER (1983). «Temporal antialiasing in Computer-Generated Animation», en los artículos de SIGGRAPH'83, *Computer Graphics*, 17(3), págs. 377–388.
- KREYSZIG, E. (1998). *Advanced Engineering Mathematics, Eighth Edition*, John Wiley & Sons, Nueva York.
- LASSETER, J. (1987). «Principles of Traditional Animation Applied to 3D Computer Animation», en los artículos de SIGGRAPH'87, *Computer Graphics*, 21(4), págs. 35–44.
- LATHROP, O. (1997). *The Way Computer Graphics Works*, John Wiley & Sons, Nueva York.
- LAUREL, B. (1990). *The Art of Human-Computer Interface Design*, Addison-Wesley, Reading, MA.
- LENGYEL, E. (2002). *Mathematics for 3D Game Programming & Computer Graphics*, Charles River Media, Hingham, MA.
- LEVOY, M. (1990). «A Hybrid Ray Tracer for Rendering Polygon and Volume Data», *IEEE Computer Graphics and Applications*, 10(2), págs. 33–40.
- LEWIS, J.-P. (1989). «Algorithms for Solid Noise Synthesis», en los artículos de SIGGRAPH'89, *Computer Graphics*, 23(3), págs. 263–270.
- LIANG, Y.-D. y B. A. BARSKY (1983). «An Analysis and Algorithm for Polygon Clipping.», *Communications of the ACM*, 26(11), págs. 868–877.
- LIANG, Y.-D. y B. A. BARSKY (1984). «A New Concept and Method for Line Clipping», *ACM Transactions on Graphics*, 3(1), págs. 1–22.
- LINDLEY,C.A. (1992). *Practical Ray Tracing in C*, JohnWiley & Sons, Nueva York.
- LISCHINSKI, D., F. TAMPIERI y D. P. GREENBERG (1993). «Combining Hierarchical Radiosity and Discontinuity Meshing», en los artículos de SIGGRAPH'93, *Computer Graphics*, págs. 199–208.
- LITWINOWICZ, P. C. (1991). «Inkwell:A2½-D Animation System», en los artículos de SIGGRAPH'91, *Computer Graphics*, 25(4), págs. 113–122.
- LODDING, K. N. (1983). «Iconic Interfacing», *IEEE Computer Graphics and Applications*, 3(2), págs. 11–20.
- LOKE, T.-S., D. TAN, H.-S. SEAH y M.-H. ER (1992). «Rendering Fireworks Displays», *IEEE Computer Graphics and Applications*, 12(3), págs. 33–43.
- LOOMIS, J., H. POIZNER, U. BELLUGI, A. BLAKEMORE y J. HOLLERBACH (1983). «Computer-Graphics Modeling of American Sign Language», en los artículos de SIGGRAPH'83, *Computer Graphics*, 17(3), págs. 105–114.
- LOPES, A. y K.BRODLIE(2003). «Improving the Robustness and Accuracy of the Marching-Cubes Algorithm for Isosurfaces», *IEEE Transactions on Visualization and Computer Graphics*, 9(1), págs. 16–29.
- LORENSON, W. E. y H. CLINE (1987). «Marching Cubes: A High-Resolution 3D Surface Construction Algorithm», en los artículos de SIGGRAPH'87, *Computer Graphics*, 21(4), págs. 163–169.
- MACKINLAY, J. D., S. K. CARD y G. G. ROBERTSON (1990). «Rapid Controlled Movement Through a Virtual 3D Workspace», en los artículos de SIGGRAPH'90, *Computer Graphics*, págs. 171–176.
- MACKINLAY, J. D., G. G. ROBERTSON y S. K. CARD (1991). «The Perspective Wall: Detail

- and Context Smoothly Integrated», *CHI '91*, págs. 173–179.
- MAESTRI, G. (1999). *[digital] Character Animation 2, Volume 1—Essential Techniques*, New Riders Publishing, Indianapolis, IN.
- MAESTRI, G. (2002). *[digital] Character Animation 2, Volume 2—Advanced Techniques*, New Riders Publishing, Indianapolis, IN.
- MAGNENAT-THALMANN, N. y D. THALMANN (1985). *Computer Animation: Theory and Practice*, Springer-Verlag, Tokyo.
- MAGNENAT-THALMANN, N. y D. THALMANN (1987). *Image Synthesis*, Springer-Verlag, Tokyo.
- MAGNENAT-THALMANN, N. y D. THALMANN (1991). «Complex Models for Animating Synthetic Actors», *IEEE Computer Graphics and Applications*, 11(5), págs. 32–45.
- MANDELBROT, B. B. (1977). *Fractals: Form, Chance, and Dimension*, Freeman Press, San Francisco.
- MANDELBROT, B. B. (1982). *The Fractal Geometry of Nature*, Freeman Press, Nueva York.
- MANTYLA, M. (1988). *An Introduction to Solid Modeling*, Computer Science Press, Rockville, MD.
- MAX, N. L. y D. M. LERNER (1985). «A Two-and-a-Half-D Motion Blur Algorithm», en los artículos de SIGGRAPH'85, *Computer Graphics*, 19(3), págs. 85–94.
- MAX, N. L. (1986). «Atmospheric Illumination and Shadows», en los artículos de SIGGRAPH'86, *Computer Graphics*, 20(4), págs. 117–124.
- MAX, N. L. (1990). «Cone-Spheres», en los artículos de SIGGRAPH'90, *Computer Graphics*, 24(4), págs. 59–62.
- MCCARTHY, M. y A. DESCARTES (1998). *Reality Architecture: Building 3D Worlds with Java and VRML*, Prentice-Hall Europe, Reino Unido.
- MEYER, G. W. y D. P. GREENBERG (1988). «Color-Defective Vision and Computer Graphics Displays», *IEEE Computer Graphics and Applications*, 8(5), págs. 28–40.
- MEYERS, D., S. SKINNER y K. SLOAN (1992). «Surfaces from Contours», *ACM Transactions on Graphics*, 11(3), págs. 228–258.
- MIANO, J. (1999). *Compressed Image File Formats*, Addison-Wesley/ACM Press, Nueva York.
- MILLER, G. S. P. (1988). «The Motion Dynamics of Snakes and Worms», en los artículos de SIGGRAPH'88, *Computer Graphics*, 22(4), págs. 169–178.
- MILLER, J. V., D. E. BREEN, W. E. LORENSON, R. M. O'BARA y M. J. WOZNY (1991). «Geometrically Deformed Models: A Method for Extracting Closed Geometric Models from Volume Data», en los artículos de SIGGRAPH'91, *Computer Graphics*, 25(4), págs. 217–226.
- MITCHELL, D. P. (1991). «Spectrally Optimal Sampling for Distribution Ray Tracing», en los artículos de SIGGRAPH'91, *Computer Graphics*, 25(4), págs. 157–165.
- MITCHELL, D. P. y P. HANRAHAN (1992). «Illumination from Curved Reflectors», en los artículos de SIGGRAPH'92, *Computer Graphics*, 26(2), págs. 283–291.
- MITROO, J. B., N. HERMAN y N. I. BADLER (1979). «Movies from Music: Visualizing Music Compositions», en los artículos de SIGGRAPH'79, *Computer Graphics*, 13(2), págs. 218–225.
- MIYATA, K. (1990). «A Method of Generating Stone Wall Patterns», en los artículos de SIGGRAPH'90, *Computer Graphics*, 24(4), págs. 387–394.
- MOLNAR, S., J. EYLES y J. POULTON (1992). «PixelFlow: High-Speed Rendering Using Image Composition», en los artículos de SIGGRAPH'92, *Computer Graphics*, 26(2), págs. 231–240.
- MOON, F. C. (1992). *Chaotic and Fractal Dynamics*, John Wiley & Sons, Nueva York.
- MOORE, M. y J. WILHELM (1988). «Collision Detection and Response for Computer Animation», en los artículos de SIGGRAPH'88, *Computer Graphics*, 22(4), págs. 289–298.
- MORTENSON, M. E. (1985). *Geometric Modeling*, John Wiley & Sons, Nueva York.
- MURAKI, S. (1991). «Volumetric Shape Description of Range Data Using the 'Blobby Model」, en los artículos de SIGGRAPH'91, *Computer Graphics*, 25(4), págs. 227–235.

- MUSGRAVE, F. K., C. E. KOLB y R. S. MACE (1989). «The Synthesis and Rendering of Eroded Fractal Terrains», en los artículos de SIGGRAPH'89, *Computer Graphics*, 23(3), págs. 41–50.
- MYERS, B. A. y W. BUXTON (1986). «Creating High-Interactive and Graphical User Interfaces by Demonstration», en los artículos de SIGGRAPH'86, *Computer Graphics*, 20(4), págs. 249–258.
- NAYLOR, B., J. AMANATIDES y W. THIBAULT (1990). «Merging BSP Trees Yields Polyhedral Set Operations», en los artículos de SIGGRAPH'90, *Computer Graphics*, 24(4), págs. 115–124.
- NICHOLL, T. M., D. T. LEE y R. A. NICHOLL (1987). «An Efficient New Algorithm for 2D Line Clipping: Its Development and Analysis», en los artículos de SIGGRAPH'87, *Computer Graphics*, 21(4), págs. 253–262.
- NIELSON, G. M., B. SHRIVER y L. ROSENBLUM, ed. (1990). *Visualization in Scientific Computing*, IEEE Computer Society Press, Los Alamitos, CA.
- NIELSON, G. M. (1993). «Scattered Data Modeling», *IEEE Computer Graphics and Applications*, 13(1), págs. 60–70.
- NISHIMURA, H. (1985). «Object Modeling by Distribution Function and a Method of Image Generation», *Journal Electronics Comm. Conf. '85*, J68(4), págs. 718–725.
- NISHITA, T. y E. NAKAMAE (1986). «Continuous-Tone Representation of Three-Dimensional Objects Illuminated by Sky Light», en los artículos de SIGGRAPH'86, *Computer Graphics*, 20(4), págs. 125–132.
- NISHITA, T., T. SIRAI, K. TADAMURA y E. NAKAMAE (1993). «Display of the Earth Taking into Account Atmospheric Scattering», en los artículos de SIGGRAPH'93, *Computer Graphics*, págs. 175–182.
- NORTON, A. (1982). «Generation and Display of Geometric Fractals in 3-D», en los artículos de SIGGRAPH'82, *Computer Graphics*, 16(3), págs. 61–67.
- NSF INVITATIONAL WORKSHOP (1992). «Research Directions in Virtual Environments», en los artículos de SIGGRAPH'92, *Computer Graphics*, 26(3), págs. 153–177.
- OKABE, H., H. IMAOKA, T. TOMIHA y H. NIWAYA (1992). «Three-Dimensional Apparel CAD System», en los artículos de SIGGRAPH'92, *Computer Graphics*, 26(2), págs. 105–110.
- OLANO, M., J. C. HART, W. HEIDRICH y M. MCCOOL (2002). *Real-Time Shading*, A. K. Natick, MA.
- OPPENHEIMER, P. E. (1986). «Real-Time Design and Animation of Fractal Plants and Trees», en los artículos de SIGGRAPH'86, *Computer Graphics*, 20(4), págs. 55–64.
- O'ROURKE, M. (1998). *Principles of Three-Dimensional Computer Animation, Revised Edition*, W. W. Norton, Nueva York.
- OSF/MOTIF (1989). *OSF/Motif Style Guide*, Open Software Foundation, Prentice-Hall, Englewood Cliffs, NJ.
- PAETH, A. W., ed. (1995). *Graphics Gems V*, Morgan Kaufmann, San Diego, CA.
- PAINTER, J. y K. SLOAN (1989). «Antialiased Ray Tracing by Adaptive Progressive Refinement», en los artículos de SIGGRAPH'89, *Computer Graphics*, 23(3), págs. 281–288.
- PALMER, I. (2001). *Essential Java 3D Fast*, Springer-Verlag, Londres.
- PANG, A. T. (1990). «Line-Drawing Algorithms for Parallel Machines», *IEEE Computer Graphics and Applications*, 10(5), págs. 54–59.
- PAO, Y. C. (1984). *Elements of Computer-Aided Design*. John Wiley & Sons, Nueva York.
- PARENT, R. (2002). *Computer Animation: Algorithms and Techniques*, Morgan Kaufmann, San Francisco, CA.
- PAVLIDIS, T. (1982). *Algorithms For Graphics and Image Processing*, Computer Science Press, Rockville, MD.
- PAVLIDIS, T. (1983). «Curve Fitting with Conic Splines», *ACM Transactions on Graphics*, 2(1), págs. 1–31.
- PEACHEY, D. R. (1986). «Modeling Waves and Surf», en los artículos de SIGGRAPH'86, *Computer Graphics*, 20(4), págs. 65–74.

- PEITGEN, H.-O. y P. H. RICHTER (1986). *The Beauty of Fractals*, Springer-Verlag, Berlín.
- PEITGEN, H.-O. y D. SAUPE, eds. (1988). *The Science of Fractal Images*, Springer-Verlag, Berlín.
- PENTLAND, A. y J. WILLIAMS (1989). «Good Vibrations: Modal Dynamics for Graphics and Animation», en los artículos de SIGGRAPH'89, *Computer Graphics*, 23(3), págs. 215–222.
- PERLIN, K. y E. M. HOFFERT (1989). «Hypertexture», en los artículos de SIGGRAPH'89, *Computer Graphics*, 23(3), págs. 253–262.
- PHONG, B. T. (1975). «Illumination for Computer-Generated Images», *Communications of the ACM*, 18(6), págs. 311–317.
- PIEGL, L. y W. TILLER (1997). *The NURBS Book*, Springer-Verlag, Nueva York.
- PINEDA, J. (1988). «A Parallel Algorithm for Polygon Rasterization», en los artículos de SIGGRAPH'88, *Computer Graphics*, 22(4), págs. 17–20.
- PITTEWAY, M. L. V. y D. J. WATKINSON (1980). «Bresenham's Algorithm with Gray Scale», *Communications of the ACM*, 23(11), págs. 625–626.
- PLATT, J. C. y A. H. BARR (1988). «Constraint Methods for Flexible Models», en los artículos de SIGGRAPH'88, *Computer Graphics*, 22(4), págs. 279–288.
- POCOCK, L. y J. ROSEBUSH (2002). *The Computer Animator's Technical Handbook*, Morgan Kaufmann, San Francisco, CA.
- POTMESIL, M. y I. CHAKRAVARTY (1982). «Synthetic Image Generation with a Lens and Aperture Camera Model», *ACM Transactions on Graphics*, 1(2), págs. 85–108.
- POTMESIL, M. y I. CHAKRAVARTY (1983). «Modeling Motion Blur in Computer-Generated Images», en los artículos de SIGGRAPH'83, *Computer Graphics*, 17(3), págs. 389–399.
- POTMESIL, M. y E. M. HOFFERT (1987). «FRAMES: Software Tools for Modeling, Rendering and Animation of 3D Scenes», en los artículos de SIGGRAPH'87, *Computer Graphics*, 21(4), págs. 85–93.
- POTMESIL, M. y E. M. HOFFERT (1989). «The Pixel Machine: A Parallel Image Computer», en los artículos de SIGGRAPH'89, *Computer Graphics*, 23(3), págs. 69–78.
- PRATT, W. K. (1978). *Digital Image Processing*, John Wiley & Sons, Nueva York.
- PREPARATA, F. P. y M. I. SHAMOS (1985). *Computational Geometry*, Springer-Verlag, Nueva York.
- PRESS, W. H., S. A. TEUKOLSKY, W. T. VETTERLING y B. P. FLANNERY (1993). *Numerical Recipes in C: The Art of Scientific Computing, Second Edition*, Cambridge University Press, Cambridge, Inglaterra.
- PRESS, W. H., S. A. TEUKOLSKY, W. T. VETTERLING y B. P. FLANNERY (2002). *Numerical Recipes in C++: The Art of Scientific Computing, Second Edition*, Cambridge University Press, Cambridge, Inglaterra.
- PRESTON, K., FAGAN, HUANG y PRYOR (1984). «Computing in Medicine», *Computer*, 17(10), págs. 294–313.
- PRUSINKIEWICZ, P., M. S. HAMMEL y E. MJOLSNESS (1993). «Animation of Plant Development», en los artículos de SIGGRAPH'93, *Computer Graphics*, págs. 351–360.
- PRUYN, P. W. y D. P. GREENBERG (1993). «Exploring 3D Computer Graphics in Cockpit Avionics», *IEEE Computer Graphics and Applications*, 13(3), págs. 28–35.
- QUEK, L.-H. y D. HEARN (1988). «Efficient Space-Subdivision Methods in Ray-Tracing Algorithms», University of Illinois, Department of Computer Science Report UIUCDCS-R-88-1468.
- RAIBERT, M. H. y J. K. HODGINS (1991). «Animation of Dynamic Legged Locomotion», en los artículos de SIGGRAPH'91, *Computer Graphics*, 25(4), págs. 349–358.
- RAO, K. R. y P. YIP (1990). *Discrete Cosine Transform*, Academic Press, Nueva York.
- REEVES, W. T. (1983a). «Particle Systems: A Technique for Modeling a Class of Fuzzy Objects», *ACM Transactions on Graphics*, 2(2), págs. 91–108.
- REEVES, W. T. (1983b). «Particle Systems—A Technique for Modeling a Class of Fuzzy Objects», en los artículos de SIGGRAPH'83, *Computer Graphics*, 17(3), págs. 359–376.

- REEVES, W. T. y R. BLAU (1985). «Approximate and Probabilistic Algorithms for Shading and Rendering Structured Particle Systems», en los artículos de SIGGRAPH'85, *Computer Graphics*, 19(3), págs. 313–321.
- REEVES, W. T., D. H. SALESIN y R. L. COOK (1987). «Rendering Antialiased Shadows with Depth Maps», en los artículos de SIGGRAPH'87, *Computer Graphics*, 21(4), págs. 283–291.
- REQUICHA, A. A. G. y J. R. ROSSIGNAC (1992). «Solid Modeling and Beyond», *IEEE Computer Graphics and Applications*, 12(5), págs. 31–44.
- REYNOLDS, C. W. (1982). «Computer Animation with Scripts and Actors», en los artículos de SIGGRAPH'82, *Computer Graphics*, 16(3), págs. 289–296.
- REYNOLDS, C. W. (1987). «Flocks, Herds y Schools: A Distributed Behavioral Model», en los artículos de SIGGRAPH'87, *Computer Graphics*, 21(4), págs. 25–34.
- RHODES, M. L., et al. (1983). «Computer Graphics and An Interactive Stereotactic System for CT-Aided Neurosurgery», *IEEE Computer Graphics and Applications*, 3(5), págs. 31–37.
- RIESENFIELD, R. F. (1981). «Homogeneous Coordinates and Projective Planes in Computer Graphics», *IEEE Computer Graphics and Applications*, 1(1), págs. 50–55.
- ROBERTSON, P. K. (1988). «Visualizing Color Gamuts: A User Interface for the Effective Use of Perceptual Color Spaces in Data Displays», *IEEE Computer Graphics and Applications*, 8(5), págs. 50–64.
- ROBERTSON, G. G., J. D. MACKINLAY and S. K. CARD (1991). «Cone Trees: Animated 3D Visualizations of Hierarchical Information», *CHI '91*, págs. 189–194.
- ROGERS, D. F. y R. A. EARNSHAW, eds. (1987). *Techniques for Computer Graphics*, Springer-Verlag, Nueva York.
- ROGERS, D. F. y J. A. ADAMS (1990). *Mathematical Elements for Computer Graphics*, McGraw-Hill, Nueva York.
- ROGERS, D. F. (1998). *Procedural Elements for Computer Graphics*, McGraw-Hill, Nueva York.
- ROSENTHAL, D. S. H., J. C. MICHENER, G. PFAFF, R. KESSEMER y M. SABIN (1982). «The Detailed Semantics of Graphics Input Devices», en los artículos de SIGGRAPH'82, *Computer Graphics*, 16(3), págs. 33–38.
- RUBINE, D. (1991). «Specifying Gestures by Example», en los artículos de SIGGRAPH'91, *Computer Graphics*, 25(4), págs. 329–337.
- RUSHMEIER, H. y K. TORRANCE (1987). «The Zonal Method for Calculating Light Intensities in the Presence of a Participating Medium», en los artículos de SIGGRAPH'87, *Computer Graphics*, 21(4), págs. 293–302.
- RUSHMEIER, H. E. y K. E. TORRANCE (1990). «Extending the Radiosity Method to Include Specularly Reflecting and Translucent Materials», *ACM Transactions on Graphics*, 9(1), págs. 1–27.
- SABELLA, P. (1988). «A Rendering Algorithm for Visualizing 3D Scalar Fields», en los artículos de SIGGRAPH'88, *Computer Graphics*, 22(4), págs. 51–58.
- SABIN, M. A. (1985). «Contouring: The State of the Art», in *Fundamental Algorithms for Computer Graphics*, R. A. Earnshaw, ed., Springer-Verlag, Berlín, págs. 411–482.
- SAKAGUCHI, H., S. L. KENT y T. COX (2001). *The Making of Final Fantasy, The Spirits Within*, Brady Games, Indianapolis, IN.
- SALESIN, D. y R. BARZEL (1993). «Adjustable Tools: An Object-Oriented Interaction Metaphor», *ACM Transactions on Graphics*, 12(1), págs. 103–107.
- SAMET, H. y M. TAMMINEN (1985). «Bintrees, CSG Trees and Time», en los artículos de SIGGRAPH'85, *Computer Graphics*, 19(3), págs. 121–130.
- SAMET, H. y R. E. WEBBER (1985). «Sorting a Collection of Polygons using Quadtrees», *ACM Transactions on Graphics*, 4(3), págs. 182–222.
- SAMET, H. y R. E. WEBBER (1988a). «Hierarchical Data Structures and Algorithms for Computer Graphics: Part 1», *IEEE Computer Graphics and Applications*, 8(4), págs. 59–75.
- SAMET, H. y R. E. WEBBER (1988b). «Hierarchical Data Structures and Algorithms for Computer Graphics: Part 2», *IEEE Computer Graphics and Applications*, 8(3), págs. 48–68.

- SCHACHTER, B. J., ed. (1983). *Computer Image Generation*, John Wiley & Sons, Nueva York.
- SCHEIFLER, R. W. y J. GETTYS (1986). «The X Window System», *ACM Transactions on Graphics*, 5(2), págs. 79–109.
- SCHOENEMAN,C., J.DORSEY, B. SMITS, J.ARVO y D. GREENBERG (1993). «Painting with Light», en los artículos de SIGGRAPH'93, *Computer Graphics*, págs. 143–146.
- SCHRODER, P. y P. HANRAHAN (1993). «On the Form Factor Between Two Polygons», en los artículos de SIGGRAPH'93, *Computer Graphics*, págs. 163–164.
- SCHWARTZ, M. W., W. B. COWAN y J. C. BEATTY (1987). «An Experimental Comparison of RGB, YIQ, LAB, HSV y Opponent Color Models», *ACM Transactions on Graphics*, 6(2), págs. 123–158.
- SEDERBERG, T.W. y E.GREENWOOD (1992). «A physically Based Approached to 2-D Shape Bending», en los artículos de SIGGRAPH'92, *Computer Graphics*, 26(2), págs. 25–34.
- SEDERBERG, T.W., P. GAO, G.WANG y H.MU (1993). «2D Shape Blending: An Intrinsic Solution to the Vertex Path Problem», en los artículos de SIGGRAPH'93, *Computer Graphics*, págs. 15–18.
- SEGAL, M. (1990). «Using Tolerances to Guarantee Valid Polyhedral Modeling Results», en los artículos de SIGGRAPH'90, *Computer Graphics*, 24(4), págs. 105–114.
- SEGAL, M., C. KOROBKIN, R. VAN WIDENFELT, J. FORAN y P. HAEBERLI (1992). «Fast Shadows and Lighting Effects Using Texture Mapping», en los artículos de SIGGRAPH'92, *Computer Graphics*, 26(2), págs. 249–252.
- SELMAN, D. (2002). *Java 3D Programming*, Manning Publications, Greenwich, CT.
- SEQUIN, C. H. y E. K. SMYRL (1989). «Parameterized Ray-Tracing», en los artículos de SIGGRAPH'89, *Computer Graphics*, 23(3), págs. 307–314.
- SHERR, S. (1993). *Electronic Displays*, John Wiley & Sons, Nueva York.
- SHILLING, A. y W. STRASSER (1993). «EXACT: Algorithm and Hardware Architecture for an Improved A-Buffer», en los artículos de SIGGRAPH'93, *Computer Graphics*, págs. 85–92.
- SHIRLEY, P. (1990). «A Ray Tracing Method for Illumination Calculation in Diffuse-Specular Scenes», *Graphics Interface '90*, págs. 205–212.
- SHIRLEY, P. (2000). *Realistic Ray Tracing*, A. K. Peters, Natick, MA.
- SHNEIDERMAN, B. (1986). *Designing the User Interface*, Addison-Wesley, Reading, MA.
- SHOEMAKE, K. (1985). «Animating Rotation with Quaternion Curves», en los artículos de SIGGRAPH'85, *Computer Graphics*, 19(3), págs. 245–254.
- SHEREINER, D., ed. (2000). *OpenGL Reference Manual, Third Edition*, Addison-Wesley, Reading, MA.
- SIBERT, J. L.,W. D. HURLEY y T.W. BLESER (1986). «An Object-Oriented User Interface Management System», en los artículos de SIGGRAPH'86, *Computer Graphics*, 20(4), págs. 259–268.
- SILLION, F. X. y C. PUECH (1989). «A General Two-Pass Method Integrating Specular and Diffuse Reflection», en los artículos de SIGGRAPH'89, *Computer Graphics*, 23(3), págs. 335–344.
- SILLION, F. X., J. R. ARVO, S. H. WESTIN y D. P. GREENBERG (1991). «A Global Illumination Solution for General Reflectance Distributions», en los artículos de SIGGRAPH'91, *Computer Graphics*, 25(4), págs. 187–196.
- SILLION, F. X. y C. PUECH (1994). *Radiosity and Global Illumination*, Morgan Kaufmann, San Francisco, CA.
- SIMS, K. (1990). «Particle Animation and Rendering Using Data Parallel Computation», en los artículos de SIGGRAPH'90, *Computer Graphics*, 24(4), págs. 405–413.
- SIMS, K. (1991). «Artificial Evolution for Computer Graphics», en los artículos de SIGGRAPH'91, *Computer Graphics*, 25(4), págs. 319–328.
- SMITH, A. R. (1978). «Color Gamut Transform Pairs», en los artículos de SIGGRAPH'78, *Computer Graphics*, 12(3), págs. 12–19.
- SMITH, A. R. (1979). «Tint Fill», en los artículos de SIGGRAPH'79, *Computer Graphics*, 13(2), págs. 276–283.

- SMITH, A. R. (1984). «Plants, Fractals y Formal Languages», en los artículos de SIGGRAPH'84, *Computer Graphics*, 18(3), págs. 1–10.
- SMITH, A. R. (1987). «Planar 2-Pass Texture Mapping and Warping», en los artículos de SIGGRAPH'87, *Computer Graphics*, 21(4), págs. 263–272.
- SMITS, B. E., J. R. ARVO y D. H. SALESIN (1992). «An Importance-Driven Radiosity Algorithm», en los artículos de SIGGRAPH'92, *Computer Graphics*, 26(2), págs. 273–282.
- SNYDER, J. M. y J. T. KAJIYA (1992). «Generative Modeling: A Symbolic System for Geometric Modeling», en los artículos de SIGGRAPH'92, *Computer Graphics*, 26(2), págs. 369–378.
- SNYDER, J. M., A. R. WOODBURY, K. FLEISCHER, B. CURRIN y A. H. BARR (1993). «Interval Methods for Multi-Point Collisions between Time-Dependent Curved Surfaces», en los artículos de SIGGRAPH'93, *Computer Graphics*, págs. 321–334.
- SOWIZRAL, H., K. RUSHFORTH y M. DEERING (2000). *The Java 3D API Specification, Second Edition*, Addison-Wesley, Reading, MA.
- SPROULL, R. F. e I. E. SUTHERLAND (1968). «A Clipping Divider», AFIPS Fall Joint Computer Conference. STAM, J. y E. FIUME (1993). «Turbulent Wind Fields for Gaseous Phenomena», en los artículos de SIGGRAPH'93, *Computer Graphics*, págs. 369–376.
- STETTNER, A. y D. P. GREENBERG (1989). «Computer Graphics Visualization for Acoustic Simulation», en los artículos de SIGGRAPH'89, *Computer Graphics*, 23(3), págs. 195–206.
- STRASSMANN, S. (1986). «Hairy Brushes», en los artículos de SIGGRAPH'86, *Computer Graphics*, 20(4), págs. 225–232.
- STRAUSS, P. S. y R. CAREY (1992). «An Object-Oriented 3D Graphics Toolkit», en los artículos de SIGGRAPH'92, *Computer Graphics*, 26(2), págs. 341–349.
- STROTHOTTE, T. y S. SCHLECHTWEG (2002). *Non-Photorealistic Computer Graphics: Modeling, Rendering, and Animation*, Morgan Kaufmann, San Francisco, CA.
- SUNG, H. C. K., G. ROGERS y W. J. KUBITZ (1990). «A Critical Evaluation of PEX», *IEEE Computer Graphics and Applications*, 10(6), págs. 65–75.
- SUTHERLAND, I. E. (1963). «Sketchpad: A Man-Machine Graphical Communication System», *AFIPS Spring Joint Computer Conference*, 23, págs. 329–346.
- SUTHERLAND, I. E. y G. W. HODGMAN (1974). «Reentrant Polygon Clipping», *Communications of the ACM*, 17(1), págs. 32–42.
- SUTHERLAND, I. E., R. F. SPROULL y R. SCHUMACKER (1974). «A Characterization of Ten Hidden Surface Algorithms», *ACM Computing Surveys*, 6(1), págs. 1–55.
- SWEZEY, R. W. y E. G. DAVIS (1983). «A Case Study of Human Factors Guidelines in Computer Graphics», *IEEE Computer Graphics and Applications*, 3(8), págs. 21–30.
- TAKALA, T. y J. HAHN (1992). «Sound Rendering», en los artículos de SIGGRAPH'92, *Computer Graphics*, 26(2), págs. 211–220.
- TANNAS, JR., L. E., ed. (1985). *Flat-Panel Displays and CRTs*, Van Nostrand Reinhold, Nueva York.
- TAUBMAN, D. y M. MARCELLIN (2001). *JPEG 2000: Image-Compression Fundamentals, Standards, and Practice*, Kluwer Academic Publishers, Norwell, MA.
- TELLER, S. y P. HANRAHAN (1993). «Global Visibility Algorithms for Illumination Computations», en los artículos de SIGGRAPH'93, *Computer Graphics*, págs. 239–246.
- TERZOPOULOS, D., J. PLATT, A. H. BARR, et al. (1987). «Elastically Deformable Models», en los artículos de SIGGRAPH'87, *Computer Graphics*, 21(4), págs. 205–214.
- THALMANN, N. M. and D. THALMANN (1985). *Computer Animation: Theory and Practice*, Springer-Verlag, Tokyo.
- THALMANN, D., ed. (1990). *Scientific Visualization and Graphics Simulation*, John Wiley & Sons, Chichester, Inglaterra.
- THIBAULT, W. C. y B. F. NAYLOR (1987). «Set Operations on Polyhedra using Binary Space Partitioning Trees», en los artículos de SIGGRAPH'87, *Computer Graphics*, 21(4), págs. 153–162.

- THOMAS, B. y W. LEFKON (1997). *Disney's Art of Animation from Mickey Mouse to Hercules*, Hyperion Press, Nueva York.
- THOMAS, F., O. JOHNSON y C. JOHNSTON (1995). *The Illusion of Life: Disney Animation*, Hyperion Press, Nueva York.
- TORBERG, J. G. (1987). «A Parallel Processor Architecture for Graphics Arithmetic Operations», en los artículos de SIGGRAPH'87, *Computer Graphics*, 21(4), págs. 197–204.
- TORRANCE, K. E. y E. M. SPARROW (1967). «Theory for Off-Specular Reflection from Roughened Surfaces», *Journal of the Optical Society of America*, 57(9), págs. 1105–1114.
- TRAVIS, D. (1991). *Effective Color Displays*, Academic Press, Londres.
- TUFTÉ, E. R. (1990). *Envisioning Information*, Graphics Press, Cheshire, CN.
- TUFTÉ, E. R. (1997). *Visual Explanations: Images and Quantities, Evidence and Narrative*, Graphics Press, Cheshire, CN.
- TUFTÉ, E. R. (2001). *The Visual Display of Quantitative Information, Second Edition*, Graphics Press, Cheshire, CN.
- TURKOWSKI, K. (1982). «Antialiasing Through the Use of Coordinate Transformations», *ACM Transactions on Graphics*, 1(3), págs. 215–234.
- UPSON, C. y M. KEELER (1988). «VBUFFER: Visible Volume Rendering», en los artículos de SIGGRAPH'88, *Computer Graphics*, 22(4), págs. 59–64.
- UPSON, C., et al. (1989). «The Application Visualization System: A Computational Environment for Scientific Visualization», *IEEE Computer Graphics and Applications*, 9(4), págs. 30–42.
- UPSTILL, S. (1989). *The RenderMan Companion*, Addison-Wesley, Reading, MA.
- VAN WIJK, J. J. (1991). «Spot Noise-Texture Synthesis for Data Visualization», en los artículos de SIGGRAPH'91, *Computer Graphics*, 25(4), págs. 309–318.
- VEENSTRA, J. y N. AHUJA (1988). «Line Drawings of Octree-Represented Objects», *ACM Transactions on Graphics*, 7(1), págs. 61–75.
- VELHO, L. y J. D. M. GOMES (1991). «Digital Halftoning with Space-Filling Curves», en los artículos de SIGGRAPH'91, *Computer Graphics*, 25(4), págs. 81–90.
- VON HERZEN, B., A. H. BARR y H. R. ZATZ (1990). «Geometric Collisions for Time-Dependent Parametric Surfaces», en los artículos de SIGGRAPH'90, *Computer Graphics*, 24(4), págs. 39–48.
- WALLACE, V. L. (1976). «The Semantics of Graphic Input Devices», en los artículos de SIGGRAPH'76, *Computer Graphics*, 10(1), págs. 61–65.
- WALLACE, J. R., K. A. ELMQUIST y E. A. HAINES (1989). «A Ray-Tracing Algorithm for Progressive Radiosity», en los artículos de SIGGRAPH'89, *Computer Graphics*, 23(3), págs. 315–324.
- WALSH, A. E. y D. GEHRINGER (2002). *Java 3D*, Prentice-Hall, Upper Saddle River, NJ.
- WANGER, L. R., J. A. FERWERDA y D. P. GREENBERG (1992). «Perceiving Spatial Relationships in Computer-Generated Images», *IEEE Computer Graphics and Applications*, 12(3), págs. 44–58.
- WARN, D. R. (1983). «Lighting Controls for Synthetic Images», en los artículos de SIGGRAPH'83, *Computer Graphics*, 17(3), págs. 13–21.
- WATT, A. (1989). *Fundamentals of Three-Dimensional Computer Graphics*, Addison-Wesley, Wokingham, Inglaterra.
- WATT, M. (1990). «Light-Water Interaction Using Backward Beam Tracing», en los artículos de SIGGRAPH'90, *Computer Graphics*, 24(4), págs. 377–386.
- WATT, A. y M. WATT (1992). *Advanced Animation and Rendering Techniques*, Addison-Wesley, Wokingham, Inglaterra.
- WEGHORST, H., G. HOOPER y D. P. GREENBERG (1984). «Improved Computational Methods for Ray Tracing», *ACM Transactions on Graphics*, 3(1), págs. 52–69.
- WEIL, J. (1986). «The Synthesis of Cloth Objects», en los artículos de SIGGRAPH'86, *Computer Graphics*, 20(4), págs. 49–54.
- WEILER, K. y P. ATHERTON (1977). «Hidden-Surface Removal Using Polygon Area Sorting», en los artículos de SIGGRAPH'77, *Computer Graphics*, 11(2), págs. 214–222.
- WEILER, K. (1980). «Polygon Comparison Using a Graph Representation», en los artículos de

- SIGGRAPH'80, *Computer Graphics*, 14(3), págs. 10–18.
- WEINBERG, R. (1978) «Computer Graphics in Support of Space-Shuttle Simulation», en los artículos de SIGGRAPH'78, *Computer Graphics*, 12(3), págs. 82–86.
- WELCH, T. (1984). «A Technique for High-Performance Data Compression», *IEEE Computer*, 17(6), págs. 8–19.
- WERNECKE, J. (1994). *The Inventor Mentor*, Addison-Wesley, Reading, MA.
- WESTIN, S. H., J. R. ARVO y K. E. TORRANCE (1992). «Predicting Reflectance Functions from Complex Surfaces», en los artículos de SIGGRAPH'92, *Computer Graphics*, 26(2), págs. 255–264.
- WESTOVER, L. (1990). «Footprint Evaluation for Volume Rendering», en los artículos de SIGGRAPH'90, *Computer Graphics*, 24(4), págs. 367–376.
- WHITTED, T. (1980). «An Improved Illumination Model for Shaded Display», *Communications of the ACM*, 23(6), págs. 343–349.
- WHITTED, T. y D. M. WEIMER (1982). «A Software Testbed for the Development of 3D Raster Graphics Systems», *ACM Transactions on Graphics*, 1(1), págs. 43–58.
- WHITTED, T. (1983). «Antialiased Line Drawing Using Brush Extrusion», en los artículos de SIGGRAPH'83, *Computer Graphics*, 17(3), págs. 151–156.
- WILHELM, J. (1987). «Toward Automatic Motion Control», *IEEE Computer Graphics and Applications*, 7(4), págs. 11–22.
- WILHELM, J. y A. VAN GELDER (1991). «A Coherent Projection Approach for Direct Volume Rendering», en los artículos de SIGGRAPH'91, *Computer Graphics*, 25(4), págs. 275–284.
- WILHELM, J. y A. VAN GELDER (1992). «Octrees for Faster Isosurface Generation», *ACM Transactions on Graphics*, 11(3), págs. 201–227.
- WILLIAMS, L. (1983). «Pyramidal Parametrics», en los artículos de SIGGRAPH'83, *Computer Graphics*, 17(3), págs. 1–11.
- WILLIAMS, L. (1990). «Performance-Driven Facial Animation», en los artículos de SIGGRAPH'90, *Computer Graphics*, 24(4), págs. 235–242.
- WITKIN, A. y W. WELCH (1990). «Fast Animation and Control of Nonrigid Structures», en los artículos de SIGGRAPH'90, *Computer Graphics*, 24(4), págs. 243–252.
- WITKIN, A. y M. KASS (1991). «Reaction-Diffusion Textures», en los artículos de SIGGRAPH'91, *Computer Graphics*, 25(4), págs. 299–308.
- WOLFRAM, S. (1984) «Computer Software in Science and Mathematics», *Scientific American*, 251(3), 188–203.
- WOLFRAM, S. (1991). *Mathematica*, Addison-Wesley, Reading, MA.
- WOO, A., P. POULIN y A. FOURNIER (1990). «A Survey of Shadow Algorithms», *IEEE Computer Graphics and Applications*, 10(6), págs. 13–32.
- WOO M., J. NEIDER, T. DAVIS y D. SHREINER (1999). *OpenGL Programming Guide, Third Edition*, Addison-Wesley, Reading, MA.
- WRIGHT, W. E. (1990). «Parallelization of Bresenham's Line and Circle Algorithms», *IEEE Computer Graphics and Applications*, 10(5), págs. 60–67.
- WU, X. (1991). «An Efficient Antialiasing Technique», en los artículos de SIGGRAPH'91, *Computer Graphics*, 25(4), págs. 143–152.
- WYSZECKI, G. y W. S. STILES (1982). *Color Science*, John Wiley & Sons, Nueva York.
- WYVILL, G. B. WYVILL y C. MCPHEETERS (1987). «Solid Texturing of Soft Objects», *IEEE Computer Graphics and Applications*, 7(12), págs. 20–26.
- YAEGER, L., C. UPSON y R. MYERS (1986). «Combining Physical and Visual Simulation: Creation of the Planet Jupiter for the Film '2010'», en los artículos de SIGGRAPH'86, *Computer Graphics*, 20(4), págs. 85–94.
- YAGEL, R., D. COHEN y A. KAUFMAN (1992). «Discrete Ray Tracing», *IEEE Computer Graphics and Applications*, 12(5), págs. 19–28.
- YAMAGUCHI, K., T. L. KUNII y F. FUJIMURA (1984). «Octree-Related Data Structures and Algorithms», *IEEE Computer Graphics and Applications*, 4(1), págs. 53–59.

- YESSIOS, C. I. (1979). «Computer Drafting of Stones, Wood, Plant y Ground Materials», en los artículos de SIGGRAPH'79, *Computer Graphics*, 13(2), págs. 190–198.
- YOUNG, D. A. (1990). *The X Window System-Programming and Applications with Xt, OSF/Motif Edition*, Prentice- Hall, Englewood Cliffs, NJ.
- ZELEZNICK, R. C., et al. (1991). «An Object-Oriented Framework for the Integration of Interactive Animation Techniques», en los artículos de SIGGRAPH'91, *Computer Graphics*, 25(4), págs. 105–112.
- ZELTZER, D. (1982). «Motor Control Techniques for Figure Animation», *IEEE Computer Graphics and Applications*, 2(9), págs. 53–60.
- ZHANG,Y. y R. E.WEBBER(1993). «Space Diffusion:An Improved Parallel Halftoning Technique Using Space-Filling Curves», en los artículos de SIGGRAPH'93, *Computer Graphics*, págs. 305–312.
- ZIV, J. y A. LEMPEL (1977). «A Universal Algorithm for Sequential Data Compression», *IEEE Transactions on Information Theory*, 23(3), págs. 337–343.
- ZIV, J. y A. LEMPEL (1978). «Compression of Individual Sequences via Variable-Rate Coding», *IEEE Transactions on Information Theory*, 24(5), págs. 530–536.



# Índice

4 conexiones, método de relleno, 208  
8 conexiones, método de relleno, 208

## A

Absolutas, coordenadas, 87  
Absoluto, valor (número imaginario), 826  
Aceleración gravitatoria, 121–2  
Acelerador:  
  ánodo (TRC), 36  
  tensión (TRC), 36  
Adobe Photoshop, formato, 804  
AGL (Apple GL), 75  
Alámbrico:  
  algoritmos de visibilidad, 566–8  
  imágenes, 127–8  
  métodos de variación de la intensidad con la profundidad, 356–7  
Algoritmo basado en diccionario, 795  
Algoritmo de Bresenham:  
  círculo, 105–6  
  línea, 96–100  
Algoritmo de búfer de profundidad (detección de visibilidad), 549–52  
Algoritmo de difusión  
  de errores, 612–3  
  de puntos, 613  
Algoritmo de inundación-relleno, 211  
Algoritmo de los cubos en marcha (*véase* Isosuperficies)  
Algoritmo de ordenación de profundidad, 556–8  
Algoritmo de sustitución, 795  
Algoritmo de visibilidad por subdivisión de área, 559–62  
Algoritmo del pintor (ordenación de profundidad), 556  
Algoritmo lineal de relleno suave, 200–2

Algoritmos de dibujo de líneas, 92–102  
  Bresenham, 96–100  
  DDA, 94–95  
  especificación de valores del búfer de imagen, 102–3  
  métodos paralelos, 100–102  
  polilíneas, 100  
Algoritmos de línea de barrido:  
  detección de superficies visibles, 554–6  
  generación de elipse, 113–20  
  generación de un círculo, 106–11  
  métodos de relleno de contornos curvos, 207  
  propiedades de coherencia, 203  
  relleno de polígonos, 202–6  
  relleno de polígonos convexos, 206  
  segmentos lineales, 92–103  
Algoritmos de recorte, 322  
  bidimensional, 323–48  
  código exterior, 324  
  códigos de región, 324, 402–3  
  coordenadas homogéneas, 402–3  
  curvas, 346, 407–8  
  de líneas de Cyrus-Beck, 330  
  de líneas de Liang-Barsky, 330–3  
  de líneas de Nichol-Lee-Nichol, 333–5  
  de polígonos de Liang-Barsky, 345  
  de polígonos de Sutherland-Hodgman, 338–43  
  de polígonos de Weiler-Atherton, 343–5  
  métodos paralelos, 338–9  
  métodos paramétricos, 330–1  
  poliedros, 406–8  
  polígonos, 336–46, 406–7

polígonos cóncavos, 343–5  
puntos, 323, 403  
recorte de líneas de Cohen-Sutherland, 324–9  
segmentos lineales, 323–36, 404–7  
texto, 347–8  
tridimensional, 401–9  
Algoritmos de relleno de área:  
  antialiasing, 226–8  
  áreas de contorno curvo, 207  
  áreas de contorno irregular, 207–11  
  método de inundación-relleno, 211  
  método de las 4 conexiones, 208  
  método de las 8 conexiones, 208  
  método de línea de barrido, 202–6  
  método de relleno de contorno, 207–10  
  métodos paralelos, 204–5  
  mezcla de color, 200–2  
  mosaico, 200  
  patrones de relleno, 199–200  
  polígonos, 202–6  
  polígonos convexos, 206  
  relleno cuadriculado, 200  
  relleno suave, 200  
  relleno tintado, 200  
Algoritmos de relleno de contorno:  
  región con 4 conexiones, 208  
  región con 8 conexiones, 208  
Algoritmos del punto medio:  
  generación de círculos, 106–11  
  generación de elipses, 113–20  
Algoritmos generadores de círculos, 104–11, 120–3  
  Bresenham, 105–6  
  del punto medio, 106–11  
Algoritmos paralelos:  
  generación de curvas, 124

- Algoritmos paralelos (*cont.*)  
 relleno de áreas, 204–5  
 visualización de líneas, 100–102
- Aliasing**, 221
- Alineación (texto), 220
- Alta definición, monitor de vídeo, 38
- American National Standards Institute (ANSI), 73, 803
- Amplitud de onda dominante, 737
- Analizador diferencial digital (DDA), algoritmo de líneas, 94–95
- Angstrom, 736
- Ángulo:**  
 campo de visión, 387–9  
 de incidencia (luz), 585–6  
 dirección, 815  
 fase, 767–8  
 reflexión especular, 588  
 refracción, 598, 622  
 rotación, 240
- Ángulo de excentricidad, 113
- Ángulo interior (polígono), 129
- Ángulo sólido, 814
- Animación, 252–3, 755  
 aceleraciones, 764–7  
 anticipación, 759  
 aplicaciones, 6, 7, 26, 28–31  
 cinemática, 761, 768–9, 770–1  
 cinemática inversa, 768–9, 770–1  
 compresión y expansión, 759  
 definiciones de objetos, 758  
 descripción de escenas, 760  
 dinámica, 768–9  
 dinámica inversa, 768–9  
 dirigida por objetivos, 768  
 diseño, 757–9  
 doble búfer, 756–7  
 efectos panorámicos, 365  
 en tiempo real, 52–53, 755  
 errores de redondeo y de aproximación, 252–3, 772  
 especificación de acciones, 760  
 especificación de movimiento, 767–9  
 especificación directa de movimiento, 767–8  
 física, 761, 768–9  
 fotograma a fotograma, 755  
 fotograma clave, 758  
 fotogramas intermedios, 758  
 funciones, 760  
 grados de libertad de un cuerpo rígido, 760–1  
 guión, 758  
 lenguajes, 760–1  
 mediante tabla de colores, 757  
 métodos de barrido, 757  
 morfismo, 27–28, 31, 762–4  
 movimiento periódico submuestreado, 770–1  
 movimientos periódicos, 771–2  
 personajes articuladas, 769–71  
 rotacional, 252–3, 771–2  
 seguimiento, 760  
 según la ley del movimiento de Newton, 768–9  
 sistema basado en fotogramas clave, 760, 761–7  
 sistema de scripts, 761  
 sistema parametrizado, 761  
*splines* Kochanek-Bartels, 444  
 submuestreada, 770–1  
 tasas de muestreo, 771–2  
 técnicas tradicionales de dibujos animados, 759–60  
 temporización, 759  
 transformaciones de tabla de color, 757  
 variación del centro de atención, 760  
 velocidad de imagen, 756–7  
 velocidad de imagen irregular, 757
- Animación por computadora, 755–75 (*véase también Animación*)
- ANSI (American National Standards Institute), 73, 803
- Antialiasing**, 221 (*véase también Muestreo*)  
 fases de píxel, 221, 225  
 filtrado, 224  
 frecuencia de muestreo de Nyquist, 221  
 fronteras de un área, 226–8  
 fronteras de una superficie, 226–8  
 función de filtro gausiana, 225–6  
 intervalo de Nyquist muestreo, 221  
 mapa de texturas, 652–3  
 máscara de ponderación, 223  
 método de Pitteway-Watkinson, 227–8  
 muestreo de un área, 221, 224  
 muestreo estocástico, 634–5  
 postfiltrado, 221  
 prefiltrado, 221  
 segmentos de líneas, 221–5  
 superficie de ponderación, 224–5  
 supermuestreo, 221  
 trazado de rayos, 632–3
- API (Application programming interface), 70
- Aplicaciones, 3–33 (*véase también Aplicaciones gráficas*)  
 de entretenimiento, 28–31  
 de formación, 19–23
- Aplicaciones gráficas:  
 agricultura, 14, 18  
 animaciones, 6, 7, 27, 28–31  
 anuncios, 7, 25–28  
 arquitectura, 9–10  
 arte, 23–28 (*véase también Arte por computadora*)  
 astronomía, 12, 31  
 CAD, 5–10  
 CAM, 7  
 ciencias físicas, 12–19, 31  
 diseño, 5–10  
 diseño de oficinas, 9–10  
 edición, 23, 25, 27  
 educación, 19–23  
 entrenamiento, 19–23  
 entretenimiento, 28–31  
 fabricación, 7  
 geología, 31  
 gráficas y diagramas, 3–5  
 ingeniería, 5–8  
 interfaces gráficas de usuario, 32–33  
 matemáticas, 12–13, 25–27  
 medicina, 31–32  
 modelado, 5–10, 12–19  
 negocios, 3, 5, 12, 18, 27–28  
 procesamiento de imágenes, 31–32  
 realidad virtual, 10–11  
 simulaciones, 6–7, 10–11, 12–23

- simuladores, 19–23
- simuladores de vuelo, 19–22
- visualización científica, 12–19
- Apple GL (AGL), 75
- Aproximación por tramos (*spline*), 123, 432
- Árbol binario de particionamiento en el espacio, 492, 558–9 (véase también BSP, árbol)
- Árbol cuaternario, 489
- Árbol octal, 489–92
  - elemento de volumen, 491
  - generación, 489–92
  - métodos de detección de visibilidad, 492, 562–3
  - operaciones CSG, 492
  - voxel, 491
- Archivo de barrido en bruto, 791
- Archivo de imágenes, 791 (véase también Formatos de archivo gráfico)
  - configuraciones, 791–2
  - métodos de compresión, 793–801
  - métodos de reducción de color, 792–3
  - principales formatos, 801–5
- Archivo vectorial, 42
- Archivos de cabecera para OpenGL, 76
- Área rellena, 127
  - atributos, 199–202
  - entorno curvo, 207–11
  - interior, 132–4
  - operaciones lógicas, 200–1
  - polígono, 128–39
  - regla del número de vueltas distintas de cero, 132–4
  - regla par-impar, 132
- Arista (polígono), 128
  - efectos de costura, 214
  - indicador (OpenGL), 214–6
  - vector, 129–30
- Aristas activas, lista de, 205
- Arrastre, 696
- Arte por computadora, 23–28, 62–3, 191
- Asistido por computadora:
  - cirugía, 31–32
  - diseño (CAD), 5
- educación y formación, 19–23
- fabricación (CAM), 7–9
- proyecto y diseño (CADD), 5
- Atenuación angular de la intensidad (fuente luminosa), 581–3
- Atenuación radial de la intensidad (fuente luminosa), 579–80
- Atributos de marcador, 220
- B**
- Bandas de mach, 617
- Barrido:
  - animaciones, 757
  - archivo (búfer), 39
  - fuente, 153
  - operaciones, 38–41, 50–54
- Barrido TC (Tomografía computerizada), 31
- Barrido de textura, 652–3
- Barrido en orden de imagen (mapeado de textura), 652–3
- Barrido en orden de píxel (textura), 652–3
- Base:
  - normal, 819–20
  - ortogonal, 819–20
  - ortonormal, 819–20
  - vectores de coordenadas, 818–9
- Base normal, 819–20
- Beta-*spline*, 465–6
  - condiciones de continuidad, 465–7
  - periódica cúbica, 466
- Bézier:
  - conversiones de B-*splines*, 469–70
  - curva cerrada, 450–1
  - curva cúbica, 451–3
  - curvas, 445–54
  - funciones de mezcla, 445–6
  - matriz, 453
  - propiedades, 450
  - superficies, 454
  - técnicas de diseño, 450–1
- Big endian, 791
- Bitblt (transferencia de bloques de bits), 265
- Bitmap, 40, 148 (véase también Búfer de imagen)
- BMP, 805
- Brillo (luz), 737
- BSP:
  - algoritmo de visibilidad, 559–60
  - árbol, 492
- B-spline:
  - abierto, 461–4
  - control local, 454–6
  - conversiones de Bézier, 469–70
  - cuadrática, 458–9
  - cúbica, 460–1
  - curvas, 454–64
  - fórmulas de recursión de Cox-deBoor, 456
  - funciones de mezcla, 456, 461
  - matriz, 461
  - no uniforme, 464
  - no uniforme racional (NURB), 467–9
  - parámetro de grado, 455
  - parámetro de tensión, 461
  - periódica, 457–64
  - propiedades, 454–5, 456–7
  - racional, 467–9
  - superficies, 464–5
  - uniforme, 457–64
  - vector de nudo, 456
- Búfer:
  - A, algoritmo, 553–4
  - auxiliar, 151–2
  - color, 39, 150–1 (véase también Búfer de refresco)
  - derecho, 150–1
  - de profundidad, 150–1, 549
  - de refresco, 39
  - frontal, 150–1, 772
  - izquierdo, 150–1
  - luminoso (trazado de rayos), 628–30
  - patrón, 150–1, 187
  - trasero, 150–1, 772
  - z (búfer de profundidad), 549
- Búfer de imagen, 39, 86
  - cálculos de dirección, 102–3
  - comando getPixel, 87
  - comando setPixel, 87
  - operaciones de barrido, 38–41
  - planos de bits, 40

Búfer de imagen, 39, 86 (*cont.*)  
 profundidad, 40  
 resolución, 39  
 tabla de búsqueda, 181–2, 604  
 transferencias de bloques de bits, 265

**C**

Cabeceras (archivos de imagen), 791  
 CAD (computer-aided design), 5  
 CADD (computer-aided drafting and design), 5  
 Caja de botones, 58, 59  
 Cálculos de masa (CSG), 488–9  
 Cálculos de volumen (CSG), 488–9  
 CAM (computer-aided manufacturing), 7–9  
 Cámara:  
   círculo de confusión, 632  
   ecuación de la lente fina, 631  
   efectos de enfoque, 630–2  
   efectos de objetivo, 634–5  
   modelo de visualización, 355–6,  
     360–1  
   número f, 631  
   parámetros (modelo de iluminación), 602  
   profundidad de campo, 632  
 Cambio de escala:  
   bidimensional, 242–4, 246  
   composición, 248  
   diferencial, 242  
   direcciones generales de cambio  
     de escala, 250–1  
   factores, 242, 284–5  
   inverso, 246–7, 287  
   métodos de barrido, 265  
   no uniforme (diferencial), 242  
   parámetros (factores), 242, 284–5  
   posición fija, 242, 284–5  
   representación matricial, 246,  
     284–5  
   tridimensional, 284–7  
   uniforme, 242  
 Cambio del centro de atención (animación), 760  
 Campo de gravedad, 698–9  
 Candela (fuente luminosa estándar), 739–40

Cañón de electrones, 36 (*véase también* Tubo de rayos catódicos)  
 Cara frontal (polígono), 137–8  
 Cara posterior (polígono), 137–8  
 Caracola, 165–68  
 Carácter  
   altura, 217  
   atributos, 217–20  
   caché de fuentes, 153–4  
   color, 220  
   con kern, 217  
   cuadrícula, 53, 153  
   cuerpo, 217  
   espaciado proporcional, 153, 217  
   fuente, 153  
   fuente de caracteres, 153  
   fuente de mapa de bits, 53, 153  
   fuente de trazo, 153  
   fuente digitalizada, 153  
   fuentes de contorno, 153  
   legible, 153  
   línea base, 217  
   línea inferior, 217  
   línea superior, 217  
   línea tope, 217  
   monoespaciado, 153  
   polimarcador, 153–4  
   precisión del texto, 220  
   primitivas, 153–4  
   recorte, 347–8  
   sans-serif, 153  
   serif, 153  
   símbolo marcador, 153–4  
   tamaño en puntos, 217  
   trazo descendente, 217  
   vector vertical, 218  
 Características psicológicas del color, 737  
 Caras poligonales de adición de detalle, 650  
 Cardioide, 165–68  
 Cartesianas, coordenadas, 809, 811  
 CAVE, 16, 688  
 Celda:  
   codificación, 54  
   recorrido (trazado de rayos), 627–8  
 Cels, 761  
 Centro de proyección, 380

Centroide (polígono), 836–7  
 CG API (computer-graphics application programming interface), 70  
 CGM (computer-graphics metafile), 803  
 CIE (International Commission on Illumination), 739  
 Cierre de superficie (radiosidad), 640–1  
 Cilíndricas, coordenadas, 812–3, 820–1  
 Cinemática, 761, 768–9, 770–1  
   (*véase también* Animación)  
 Círculo:  
   de confusión, 632  
   ecuaciones, 104–6  
   propiedades, 104  
   simetría, 104  
 CMY, modelo de color, 745–7  
 CMYK, modelo de color, 746  
 Codificación aritmética, 798–9  
 Codificación de longitud de recorrido, 54, 794  
 Codificación mediante coseno discreto, 799–801  
 Código de salida, 324  
 Códigos (trazado de rayos), 634–5  
 Códigos de región (recorte):  
   bidimensional, 324  
   tridimensional, 402–3  
 Coeficiente:  
   reflexión ambiente, 587  
   reflexión difusa, 584–5  
   reflexión specular, 588–9, 590–1  
   transparencia, 600  
 Coeficientes binomiales, 445–6  
 Cohen-Sutherland, recorte de líneas de, 324–9  
 Color: (*véase también* Luz; Fuentes luminosas)  
   brillo, 737  
   características psicológicas, 737  
   codificación (visualización de datos), 529  
   complementario, 738  
   conceptos intuitivos, 738–9  
   consideraciones de selección, 751  
   cromaticidad, 737  
   espectral, 735–6, 740–1

- espectro (electromagnético), 735–6  
 gama, 738  
 iluminante C, 740  
 luminancia, 596–7, 740, 744  
 mezcla, 78–79, 185–6  
 modelos de iluminación, 594–6  
 percepción, 737  
 primario, 738  
 pureza, 737, 742  
 puro, 737, 739  
 real, sistema, 44  
 saturación, 737  
 sombras, 739  
 tintas, 739  
 tonalidades, 739  
 tono, 737
- Columna:  
 número, 86  
 vector (matriz), 821–2
- Commission Internationale de l'Éclairage (CIE), 739
- Componentes nemáticos de cristal líquido, 45–46
- Compresión  
 con pérdidas, 794  
 de archivos, técnicas, 793–801  
 sin pérdidas, 794  
 y expansión (animación), 759
- Computadora cliente, 69
- Concatenación (matriz), 247
- Condiciones de continuidad (*spline*):  
 geométrica, 434–5  
 paramétrica, 434
- Condiciones de contorno (*spline*), 435, 438, 441–2
- Conjugado (número complejo), 826
- Conjunto de fractales invariantes, 494
- Cono:  
 de visión (visualización en perspectiva), 383  
 filtro, 225–6  
 hexagonal (HSV), 747  
 receptores, 742  
 trazado, 633 (véase también Trazado de rayos)
- Constante de amortiguamiento, 768
- Constante de fuerza, 526–7
- Continuidad de orden cero:  
 geométrica, 434–5  
 paramétrica, 434
- Continuidad de primer orden:  
 geométrica, 435  
 paramétrica, 434
- Continuidad de segundo orden:  
 geométrica, 435  
 paramétrica, 434
- Continuidad paramétrica (*spline*), 434
- Contorno:  
 rectángulo, 86  
 recuadro, 86  
 volumen, 626
- Contorno (borde de intensidad), 605–7
- Contornos de línea, 530–2
- Contracción (tensor), 536
- Control:  
 grafo, 433  
 ícono, 724–5  
 operaciones, 73  
 polígono, 433  
 punto (*spline*), 432  
 rejilla (TRC), 37  
 superficie (terreno fractal), 506–7
- Control local (*spline*), 454–5, 456
- Controlado por objetivos, movimiento, 768
- Controlador  
 de vídeo, 50–53  
 gráfico, 53
- Convexo:  
 ángulo, 131–2  
 armazón, 432, 457–8  
 división de polígonos, 131–2  
 polígono, 129, 206
- Coordenadas:  
 absolutas, 87  
 cartesianas, 51, 71–72, 809, 811  
 cilíndricas, 812–3, 820–1  
 curvilíneas, 812–3  
 de dispositivo, 71  
 de pantalla, 71, 86–87, 395, 809, 812  
 de superficie, 812–3  
 de textura, 650  
 de un pixel (cuadrícula de pantalla), 86–87, 124–5
- ejes, 812–3  
 esféricas, 813, 821  
 homogéneas, 89, 245  
 locales (mapeado de imagen), 658–9  
 locales (modelado), 71, 784–5  
 maestras, 71  
 modelado, 71–72, 785  
 normalizadas, 71–72  
 ortogonales, 812–3  
 polares, 788–89, 813–4  
 posición actual, 87  
 proyección normalizada, 361, 370–2, 379, 392–5  
 regla de la mano derecha, 71, 811  
 regla de la mano izquierda, 812–3  
 relativas, 87  
 selección, 692  
 transformación, 267–9, 291–2  
 universales, 71–72  
 visualización, 71–72
- Coordenadas de visualización:  
 bidimensionales, 307, 308, 809  
 regla de la mano derecha, 812–3  
 tridimensionales, 355–6, 811  
 uvn, 363–4
- Copo de nieve (fractal), 496–7
- Corrección gamma, 604–5
- Correspondencia de lenguaje, 74
- Cosenos directores, 815
- Costura (huecos en las aristas de los polígonos), 214
- Cox-deBoor, fórmulas de recursión, 456
- Cramer, regla de, 839
- CSG, métodos, 486–9 (véase también Geometría sólida constructiva)
- Cuadrícula:  
 carácter, 53, 153  
 en construcción interactiva de imágenes, 696
- Cuaternio, 827–8  
 construcciones fractales, 519  
 inverso, 827–8  
 módulo, 827–8  
 multiplicación, 827–8  
 multiplicación por un escalar, 827–8

Cuaternio, 827–8 (*cont.*)  
 parte escalar, 827–8  
 parte vectorial, 827–8  
 representación mediante par ordenado, 827–8  
 rotaciones, 280–4  
 suma, 827–8  
 términos imaginarios, 826–7  
 Cubo unitario (volumen de visualización), 370–2, 378–9, 392–5  
 Cuerpo:  
 no rígido, 526–9  
 personaje, 217  
 Cuerpo rígido:  
 grados de libertad, 760–1  
 transformación, 238–9, 241, 253–4  
 Cursor de mano (digitalizador), 62  
 Curva:  
 algoritmos paralelos, 124  
 aproximación mediante líneas rectas, 121  
 atributos, 193–5  
*B-spline*, 454–64  
 caracola, 165–68  
 cardioide, 165–68  
 círculo, 104–11, 120–3  
 consideraciones de simetría, 104, 113, 121  
 construcción por tramos, 123, 432, 435  
 cuatro hojas, 165–68  
 de ajuste, 437  
 de respuesta del monitor, 604–5  
 elipse, 111–20, 120–3  
 espiral, 165–68  
 fractal, 492–3, 497–9, 507–21  
 hipérbola, 121–2  
 Koch (fractal), 497–8  
 parábola, 121–2  
 polinómica, 123 (*véase también Curva spline*)  
 recorte, 346  
 representaciones paramétricas, 121, 123  
 secciones cónicas, 120–3  
 supercuádrica, 422–4  
 tres hojas, 165–68  
 Curva *spline*, 123, 431–69

aproximación, 432  
 armazón convexo, 432, 457–8  
*Beta-spline*, 465–7  
 Bézier, 445–54  
*B-spline*, 454–64  
 cardinal, 441–4  
 Catmull-Rom, 441–2  
 condiciones de continuidad, 434  
 condiciones de contorno, 435  
 continuidad geométrica, 434–5  
 continuidad paramétrica, 434  
 control local, 454–5, 456  
 conversiones, 469–70  
 de visualización, 470–4  
 funciones base, 435, 436  
 funciones de mezcla, 435, 436  
 grafo de control, 433  
 hermética, 438–41  
 interpolación, 432  
 interpolación cúbica, 437–44  
 Kochanek-Bartels, 444  
 matriz base, 436  
 natural, 438  
 NURB, 467  
 Overhauser, 441–2  
 parámetro de continuidad, 444–5  
 parámetro de desplazamiento, 444–5, 454, 466  
 parámetro de tensión, 444–5  
 polígono característico, 433  
 polígono de control, 433  
 puntos de control, 432  
 racional, 467–9  
 representación matricial, 435  
 vector de nudo, 456  
 Curvas cuádricas, 420–2  
 Curvilíneas, coordenadas, 812–3  
 Cyrus-Beck, recorte de líneas de, 330

**D**

Datos de entrada gráfica, 689  
 Datos en bruto, 791  
 DDA, algoritmo de líneas, 94–95  
 Deflexión: (*véase también Tubo de rayos catódicos*)  
 bobinas, 36, 37  
 corriente, 37  
 magnética (TRC), 37

placas, 37, 38  
 Del, 830  
 al cuadrado, 832  
 Delta-delta, TRC de máscara de sombra, 42  
 Dentado, 92 (*véase también Antialiasing*)  
 Derivada parcial, 829–30  
 Descarga de gas, pantallas, 44  
 Descomposición LU, 840  
 Desenfoque de movimiento, 635–7  
 Despiece, 359  
 Desplazamiento del punto medio (fractal), 504–6  
 Detalles de superficies, 647–59  
 caras poligonales, 650  
 mapeado de entorno, 646  
 mapeado de relieve, 656–8  
 mapeado de texturas, 650–6  
 mapeado del sistema de referencia, 658–9  
 texturado procedimental, 655–6  
 Detección de líneas visibles, 566–8  
 (*véase también Variación de la intensidad con la profundidad*)  
 Detección de superficies visibles, 358, 547–68  
 algoritmo del pintor (ordenación de profundidad), 556  
 clasificación de algoritmos, 547–8  
 comparación de algoritmos, 564–5  
 eliminación de caras ocultas, 548–9  
 gráficas de contorno de superficie, 566  
 método de líneas de barrido, 554–6  
 método de ordenación de la profundidad, 556–8  
 método de proyección de rayos, 563–4  
 método de subdivisión del área, 559–62  
 método del árbol BSP, 559–60  
 método del búfer A, 553–4  
 método del búfer de profundidad, 549–52  
 método del espacio de imagen, 547  
 métodos alámbricos, 566–8

métodos de árbol octal, 492, 562–3  
 métodos del espacio de objetos, 547  
 superficies curvas, 565–6  
 variación de la intensidad con la profundidad, 567–8  
 Determinación de raíces mediante falsa posición, 842  
 Determinación de raíces por biseción, 841  
 Determinante, 823–4  
 Diagrama cromático, 740–1  
 colores complementarios, 741  
 gamas de colores, 741  
 iluminante C, 740  
 línea púrpura, 740  
 longitud de onda dominante, 741–2  
 pureza, 742  
 Diagramas de contorno, 530–2  
 Diagrama de tiempos, 5  
 Dibujo de gráficas, 3–5, 160–65  
 (*véase también* Gráfica)  
 Diferencias finitas, 845–6  
 Diferencias hacia adelante, 471–2  
 Difusa:  
 coeficiente de reflexión, 584–96  
 reflexión, 583–8  
 transmisión, 598  
 Digitalización, 53–54, 86  
 Digitalizador, 62–64  
 acústico, 63  
 cursor de mano, 62–63  
 electromagnético, 63  
 pluma, 61–63  
 precisión, 63  
 resolución, 63  
 sónico, 63  
 tridimensional, 63  
 Dimensión, 818–9  
 de similaridad (fractal), 495–6  
 euclídea, 492–3, 496  
 fraccional, 492–3, 495–6  
 fractal, 492–3, 494–7  
 recuadro, 495–6  
 topológica, 495–6  
 Dinámica, 768–9 (*véase también* Animación)

Diodo emisor de luz (LED), 44  
 Direccional:  
 derivada, 831  
 fuente luminosa, 580  
 Dispositivo, coordenadas de, 71  
 Dispositivos de entrada:  
 caja de botones, 58, 59  
 clasificación lógica, 689–94  
 conmutadores, 58  
 de trazo, 690  
 diales, 58, 59  
 digitalizador, 62–64  
 digitalizadores sónicos tridimensionales, 63  
 dispositivo de cadena, 690  
 dispositivo de elección, 690–2  
 dispositivo de selección, 690, 692–4  
 dispositivo de trazo, 690  
 dispositivo evaluador, 690–1  
 dispositivo localizador, 690  
 electroguante, 61  
 escáner, 64  
 joystick, 61  
 lápiz óptico, 66  
 lógicos, 689–94  
 panel táctil, 64–66  
 ratón Z, 60  
 ratón, 59–60  
 sistemas de voz, 67  
*spaceball*, 60–61  
 tableta gráfica, 61–63  
 teclado, 58  
*trackball*, 60–61  
 Dispositivos de impresión, 67–69  
 Distancia:  
 punto a línea, 692–4  
 trayectoria de trazado de rayos, 622–3  
 Divergencia:  
 operador, 832  
 teorema, 835  
 División de polígonos cóncavos  
 método vectorial, 130–1  
 métodos rotaciones, 130–1  
 División de polígonos convexos, 131–2  
 Doble búfer, 52–53, 756–7  
 Doble refracción, 598

## E

Ecuación de una lente fina, 631  
 Ecuación punto-pendiente de una línea, 92–93  
 Ecuaciones diferenciales  
 ordinarias, 844–5  
 parciales, 845–6  
 Educación y formación, aplicaciones, 19–23  
 Efectos atmosféricos (modelo de iluminación), 601  
 Efectos de ampliación o reducción, 306  
 Efectos de iluminación de superficie, 583–4 (*véase también* Modelo de iluminación)  
 Efectos de pantalla dividida, 313, 320–2  
 Efectos globales de iluminación, 583–4, 618–619, 638–44  
 Efectos panorámicos, 306  
 Ejes:  
 inclinación, 263–4, 290–1  
 principales, 369, 383  
 reflexión, 260, 290  
 rotación, 240  
 Electroguante, 61–62 (*véase también* Realidad virtual)  
 Electromagnético:  
 espectro, 735–6  
 radiación, 182, 735  
 Electrostático:  
 deflexión (TRC), 37, 38  
 enfoque (TRC), 37  
 impresora, 67–68  
 Elemento de imagen (píxel), 39  
 Elemento de volumen, 491  
 Eliminación de caras posteriores, 548–9  
 Elipse:  
 algoritmo del punto medio, 113–20  
 ángulo de excentricidad, 113  
 ecuaciones, 111–13, 121–2  
 propiedades, 111–20  
 punto de enfoque, 111–12  
 simetría, 113  
 Elipsoide, 420–1

- Energía:
- del fotón, 638
  - distribución (fuente luminosa), 737
  - función (modelado de tejidos), 528
  - niveles cuánticos, 37
  - radiante, 638–9
- Entrelazado de líneas de barrido, 40
- Errores de redondeo y de aproximación (animación), 252–3, 772
- Escala de grises, 182
- Escala diferencial, 242
- Escaneado (patrones de textura), 652
- Escáner de imágenes, 64
- Esfera:
- cálculos de intersección con rayos, 623–5
  - ecuaciones, 420, 829–30
- Esféricas, coordenadas, 813–4, 821
- Espacio riemanniano, 820–1
- Especificación de movimiento, 767
- Especro de frecuencias (electromagnético), 735–6
- Espejo varifocal, 47
- Espejo vibratorio, 47
- Espiral, 165–68
- Estación de trabajo gráfica, 54–58
- Estado:
- máquina, 179
  - parámetros, 180
  - sistema, 179
  - variables, 180
- Estándares software, 73–74
- Estereoradian, 814
- Estilos de relleno, 199–200
- Estructura (subsección de imagen), 156
- Euclídea:
- dimensión, 492–3, 496
  - métodos geométricos, 492
- Euler:
- fórmula, 826–7 (*véase también Número complejo*)
  - método (resolución de ecuaciones diferenciales), 844
- Extensión de coordenadas, 86
- F**
- Factor de opacidad, 600
- Factor de reflectividad (radiosidad), 640–1
- Factores de forma (radiosidad), 640–1
- Fakespace Pinch, electroguantes, 50
- Filamento (TRC), 36
- Filtro: (*véase también Antialiasing*)
- cono, 225–6
  - función, 224
  - gausiano, 225–6
  - recuadro, 225–6
- Físico:
- animaciones, 761, 768–9
  - modelado, 526–9
- Fluctuación, 634–5
- Focos, 580
- Forma funcional
- explícita, 828–9
  - implícita, 828–9
- Forma polar (número complejo), 826–7
- Formatos de archivo gráfico:
- Adobe Photoshop, 804
  - algoritmo de diccionario, 795
  - algoritmo de sustitución, 795
  - archivo de barrido, 791
  - archivo de barrido en bruto, 791
  - big endian, 791
  - BMP, 805
  - cabecera, 791
  - CGM, 803
  - codificación aritmética, 798–9
  - codificación Huffman, 795–8
  - codificación LZW, 794–5
  - codificación por coseno discreto, 799–801
  - codificación por longitud de recorrido, 794
  - compresión con pérdidas, 794
  - compresión sin pérdidas, 794
  - configuraciones, 791–2
  - datos en bruto, 791
  - formato híbrido, 792
  - GIF, 805
  - JFIF, 802–3
  - JPEG, 801–3
  - little endian, 791
  - MacPaint, 804
  - metaarchivo, 792
- métodos fractales, 795
- PCX, 805
- PICT, 804
- PNG, 803–4
- reducción de color de corte medio, 793
- reducción de color por popularidad, 792–3
- reducción de color uniforme, 792
- representación geométrica, 792
- SPIFF, 802–3
- Targa, 805
- técnicas de compresión de archivos, 793–801
- TGA, 805
- TIFF, 803
- transformación de coseno inverso, 799–800
- XBM, 804
- XPM, 804
- Formatos híbridos, 792
- Fósforo, 36
- Fotograma:
- animación, 755
  - área de pantalla, 39
  - velocidad, 756–7
- Fotogramas clave (animación), 758
- Fotogramas intermedios (animación), 758
- Fotón, 37, 638–9
- Fractal:
- autoafines, 494
  - auto-elevación al cuadrado, 494, 507–21
  - auto-inversos, 494, 521
  - auto-similares, 494, 497–9
  - características, 492–3
  - características de auto-similitud, 492–3
  - características del terreno, 505–7
  - clasificación, 494
  - conjunto de Julia, 508
  - conjunto de Mandelbrot, 513–9
  - conjunto invariante, 494
  - construcciones afines, 501–4
  - construcciones geométricas, 497–501
  - copo de nieve, 496–7
  - curva de Koch, 496–7

- curva de Peano, 496–9  
 desplazamiento del punto medio, 504–6  
 dimensión, 492–7  
 dimensión de Hausdorff–Besicovitch, 495–6  
 dimensión de recuadro, 495–6  
 dimensión de similaridad, 495–6  
 dimensión fraccional, 492–3, 495–6  
 espacio de Peano, 498–9  
 estadísticamente auto-similar, 494, 499–501  
 generador, 496–7  
 iniciador, 496–7  
 métodos de desplazamiento aleatorio del punto medio, 500, 504–6  
 métodos de inversión, 519–21  
 métodos de recubrimiento de recuadros, 495–6  
 métodos de recubrimiento topológico, 496  
 métodos de subdivisión, 494–6  
 movimiento browniano, 501–4  
 movimiento browniano fraccional, 501–4  
 procedimientos de generación, 494  
 representación mediante cuaternios, 519  
 superficies de control, 506–7  
 Frecuencia dominante, 737  
 Fresnel, leyes de la refracción, 588  
 Frustum, 384–5, 386–92  
 f-stop, 631  
 ftp (file-transfer protocol), 70  
 Fuente, 153 (*véase también* Tipo de letra)  
 caché, 150–1  
 de barrido, 153  
 de contorno, 153  
 de trazos, 153  
 legible, 153  
 mapa de bits, 53, 153  
 monoespaciada, 153  
 proporcional, 153, 217  
 sans-serif, 153  
 serif, 153  
 Fuente luminosa, 578–83  
 atenuación angular de la intensidad, 581–3  
 atenuación radial de la intensidad, 579–80  
 brillo, 737  
 candela, 740  
 compleja, 582–3, 635  
 direccional, 580  
 distribución de energía, 737  
 energía radiante, 638–9  
 flujo radiante, 638–9  
 foco, 580  
 frecuencia dominante, 737  
 infinitamente distante, 579  
 intensidad, 182, 638–9  
 longitud de onda dominante, 737  
 luminancia, 183, 596–7, 740, 744  
 parámetros de pantalla, 582–3  
 potencia radiante, 638–9  
 puntual, 578  
 radiancia espectral, 638–9  
 radiosidad, 638–9, 640–1  
 vector de dirección, 586–7  
 Warn, modelo de, 582–3  
 Función de densidad (objeto sin forma), 429–31  
 Función de perturbación (mapeado de relieve), 656  
 Función de relieve, 656  
 Funciones base, 435 (*véase también* Funciones de mezcla)  
 Funciones de ajuste de color, 738  
 Funciones de atenuación (luz), 579–80, 581–3  
 Funciones de entrada, 73, 694–5  
 realimentación de eco, 695  
 retrollamada, 81, 695  
 Funciones de mezcla (*spline*), 435  
 Bézier, 445–6  
*B-spline*, 456, 461  
 cardinal, 443  
 hermética, 440  
 Funciones de retrollamada, 81, 695  
 Funciones diferenciales, 844–5  
**G**  
 Gama (color), 738  
 Gauss:  
 distribución, 38  
 eliminación, 839  
 función de densidad (relieve), 429–31  
 función de filtro, 225–6  
 teorema, 834–5  
 Gauss-Seidel, método, 840  
 Generación de *splines*:  
 método de diferencias finitas, 471–2  
 método de Horner, 470–1  
 métodos de subdivisión, 472–4  
 Generador (fractal), 496–7  
 Generador de congruencia lineal, 843–4  
 Generador de números aleatorios, 843–4  
 Geometría de objetos, 125–7  
 Geometría sólida constructiva (CSG), 486–9  
 cálculos de masa, 488–9  
 cálculos de volumen, 488–9  
 métodos de árbol octal, 492  
 métodos de proyección de rayos, 486–9  
 plano de partida, 488  
 Geométrico:  
 continuidad (*spline*), 434–5  
 modelo, 779  
 primitivas, 85 (*véase también* Primitivas de salida gráfica)  
 reglas de producción, 521–4  
 representaciones (archivos de imagen), 792  
 tablas (polígono), 135–6  
 transformaciones, 73, 237 (*véase también* Transformaciones)  
 Gestor de ventana, 32  
 getPixel, procedimiento, 87  
 GIFF, 805  
 GKS (Graphical Kernel System), 73  
 GL (Graphics Library), 73  
 Glifo, 536  
 GLU (OpenGL Utility), 75  
 función de dirección de visualización, 396–7  
 función de ventana de recorte bidimensional, 78–79  
 función de ventana de selección, 712–3

GLU (OpenGL Utility) (*cont.*)  
 funciones de ajuste, 484–5  
 funciones de curvas *B-spline*,  
 480–2  
 funciones de superficie *B-spline*,  
 482–4  
 funciones para superficies cuádricas, 425–8  
 ventana de visualización, 78–79  
 GLUT (OpenGL Utility Toolkit), 76  
 creación de submenús, 720–3  
 función de cambio de tamaño de la ventana de visualización,  
 158–61, 317  
 función de creación de menús,  
 717–9  
 función de cajas de botones, 711  
 función de cursor de pantalla,  
 318–9  
 función de superficie cúbica (tetera), 425  
 función de tetera, 425  
 función inactiva, 319, 772  
 funciones de atributo de carácter,  
 220  
 funciones de caracteres, 155  
 funciones de consulta, 316–7,  
 320, 720–1  
 funciones de dial, 711  
 funciones de modificación de menús, 723–4  
 funciones de ratón, 700–5, 705–9  
 funciones de spaceball, 710  
 funciones de tableta, 710  
 funciones de teclado, 705–9  
 funciones para poliedros regulares, 416–9  
 funciones para superficies cuádricas, 424–5  
 gestión de múltiples menús, 720  
 gestión de ventanas de visualización, 76–78, 78–79, 158–61,  
 315–20  
 menú actual, 720  
 GLX (extensión OpenGL al sistema X window), 75  
 Gouraud, representación superficial, 614–7  
 Grad al cuadrado, 832

Gradiente (grad), operador, 830–1  
 Grados de libertad, 760–1  
 Gráfica:  
 de barras, 3–5, 162–163  
 de sectores, 3–5, 163  
 de tiempos, 5  
 tridimensional, 3–5  
 Gráficas de contorno superficial, 566  
 Gráficas y diagramas, 3–5, 160–65  
 (*véase también* Gráfica)  
 Gráficos empresariales, 12, 529  
 (*véase también* Visualización de datos)  
 Gráficos por Internet:  
 explorador Mosaic, 70  
 HTML (hypertext markup language), 70  
 http (hypertext transfer protocol), 70  
 Netscape Navigator, 70  
 protocolo de transferencia de archivos (ftp), 70  
 sitio ftp, 70  
 TCP/IP, 69–70  
 URL (uniform resource locator), 69–70  
 World Wide Web, 69  
 Graftal, 524  
 Gramática L, 524  
 Gramáticas de formas, 521–4  
 Graphical Kernel System (GKS), 73  
 Green:  
 ecuaciones de transformación, 835  
 identidades, 835  
 teorema en el espacio, 835  
 teorema en el plano, 833–4  
 Guión (animación), 757

## H

*h*, parámetro, 89  
 Hausdorff-Besicovitch, dimensión, 495–6  
 Haz de electrones: (*véase también* Tubo de rayos catódicos)  
 convergencia, 37  
 deflexión electrostática, 37, 38  
 deflexión magnética, 36, 37  
 enfoque, 37  
 intensidad, 37

tamaño de punto, 38  
 Hercio, 40  
 Hipérbola, 121–2  
 HLS, modelo de color, 750–1  
 Homogéneas, coordenadas, 89, 245  
 Hooke, ley de, 526–7  
 Horner, método de factorización de polinomios, 470–1  
 HSV, modelo de color (*véase* HSV)  
 Huffman, codificación, 795–8  
 Hypertext Markup Language (HTML), 70

## I

Iconos, 33  
 de aplicación, 724–5  
 de comando, 724–5  
 Iluminante C, 740  
 Imágenes de tono continuo, 605–6  
 (*véase también* Semitono)  
 Impresoras, 67–68  
 de chorro de tinta, 67  
 de impacto, 67  
 de matriz de puntos, 67  
 de no impacto, 67  
 electrotérmica, 68  
 Inclinación  
 en la dirección *x*, 263–4  
 en la dirección *y*, 264  
 en la dirección *z*, 290–1  
 Índice de refracción, 598  
 Iniciador (fractal), 496–7  
 Instancia, 779 (*véase también* Modelado)  
 Intensidad, 182, 638–9, 639–40  
 (*véase también* Modelos de iluminación; Radiosidad)  
 atenuación angular, 581–3  
 atenuación radial, 579–80  
 corrección gamma, 604–5  
 haz de electrones (TRC), 37  
 interpolación (representación de Gouraud), 614–7  
 matriz (mapeado de entorno), 646  
 niveles (sistema), 603  
 Interfaz gráfica de usuario:  
 coherencia, 726  
 componentes, 724–7

- diálogo con el usuario, 724  
 diseño, 724–7  
 facilidades de ayuda, 725–6  
 iconos, 724–5  
 menús, 724–5  
 minimización de la memorización, 726  
 modelo del usuario, 724  
 múltiples niveles de experiencia, 725–6  
 realimentación, 726–7  
 tratamiento de errores, 726  
 ventanas, 724–5
- Interfaz IBM OS/2 para OpenGL (PGL), 75
- Interfaz Windows para OpenGL (WGL), 75
- International Standards Organization (ISO), 73, 74, 803
- Inversa:
- barrido (mapeado de textura), 652
  - cinemática, 768–9, 770–1
  - cuaterniono, 827–8
  - dinámica, 768–9
  - matriz, 824
  - transformación coseno, 799–800
  - transformaciones geométricas, 246
- ISO (International Standards Organization), 73, 74, 803
- Isolíneas, 530–2
- Isosuperficies, 530–2
- J**
- JFIF, 802–3
- Joystick:
- isométrico, 61
  - móvil, 61
  - sensible a la presión (isométrico), 61
- JPEG, 801–3
- Julia, conjunto, 508
- K**
- Koch, curva (copo de nieve fractal), 496–7
- Kochanek-Bartels, *spline*, 444
- L**
- Lado (polígono), 128
- Lambert, ley del coseno, 583–4
- Lápiz (digitalizador), 61
- Lápiz óptico, 66
- LCD (pantalla de cristal líquido), 45–47
- de matriz activa, 46–47
  - de matriz pasiva, 47
- LED (diodo emisor de luz), 46
- Liang-Barsky, recorte:
- líneas bidimensionales, 330–3
  - polígonos, 345
- Línea:
- antialiasing, 221–5
  - atributos, 188–92
  - ecuación punto-pendiente, 92–93
  - ecuaciones, 93
  - extremos, 189
  - opciones de pincel y pluma, 191–2
  - representación paramétrica, 323, 330, 380–1
  - variaciones de intensidad, 225–6
- Línea base (carácter), 217
- Línea de barrido, 39
- entretejido, 40
- Línea digitalizada en escalera, 92
- Línea escalonada, 92
- Línea inferior (carácter), 217
- Línea púrpura (en diagrama cromático), 740
- Línea superior (carácter), 217
- Líneas de campo, 534
- Líneas de flujo, 534
- Líneas punteadas, generación, 190–1
- Lista de aristas:
- activas, 205–6
  - en tablas de polígonos, 135–6
  - ordenadas, 204–5
- Little endian, 791
- Locales, coordenadas, 71, 784–5
- Longitud de onda (luz), 735
- LU, descomposición, 840
- Luminancia, 183, 596–7, 740, 744
- Luz: (*véase también* Color; Modelos de iluminación; Representación de superficies)
- ambiente (fondo), 584 (*véase también* Modelos de iluminación)
  - ángulo de incidencia, 585–6
- banda de frecuencias, 735
- blanca, 736, 737
- brillo, 737
- color espectral, 735
- cromaticidad, 737
- diagrama cromático, 740–1
- de fondo (ambiente), 583–4
- espectro electromagnético, 735–6
- frecuencia, 38, 735
- iluminante C, 740
- longitud de onda, 735
- luminancia, 183, 596–7, 740, 744
- período, 735
- propiedades, 735–7
- pureza, 737
- tono, 737
- velocidad, 736
- LZW, codificación, 794–5
- M**
- MacPaint, formato, 804
- Maestras, coordenadas, 71, 785
- Malla (polígono), 129, 141–5, 416
- cuadrilátera, 143–5, 416
  - triangular, 141–3, 416
- Mandelbrot, conjunto, 513–9
- Mapeado
- de entorno, 646
  - de fotones, 646–7
  - de patrones, 650 (*véase también* Mapeado de texturas)
  - de reflexión, 646
  - de relieve, 656–8
  - del sistema de referencia, 658–9
  - ventana a visor, 306, 310–3
- Mapeado de texturas:
- barrido en orden de imagen, 652–3
  - barrido en orden de píxel, 652–3
  - barrido inverso, 652–3
  - bidimensional (superficie), 651–5
  - coordenada *r*, 654–5
  - coordenada *s*, 650–1, 654–5
  - coordenada *t*, 651, 654–5
  - coordenadas, 650–1
  - espacio, 650–1
  - funciones, 650–1
  - matriz, 650–1, 651, 654–5
  - mip maps, 655

- Mapeado de texturas: (*cont.*)  
 patrones de reducción, 655  
 procedural, 655–6  
 sólido, 654–5  
 texel, 650–1  
 tridimensional (volumen), 654–5  
 unidimensional (lineal), 650–1
- Marcador (carácter), 153–4
- Máscara, 148, 199  
 de píxeles, 148  
 de ponderación del píxel, 223  
 de sombra, 42–43
- Material  
 elástico (objeto no rígido), 526–9  
 opaco, 597  
 transparente, 597  
 translúcido, 597
- Matriz, 821–4  
 aleatorización, 611  
 base (*spline*), 436  
 Bézier, 453  
 B-spline, 461  
 cambio de escala, 246, 284–5  
 caracterización de *splines* (base), 436  
 cardinal, 441–2  
 columna, 821–2  
 concatenación, 251–2, 823  
 cuadrada, 821–2  
 de entorno, 646  
 determinante, 823–4  
 de vértices (OpenGL), 146–7  
 fila, 821–2  
 hermítica, 439–40  
 identidad, 824  
 inclinación, 263–4, 290–1  
 inversa, 824  
 multiplicación escalar, 822–3  
 multiplicación, 823  
 no singular, 824  
 reflexión, 260–2, 290  
 rotación, 246, 272–4, 279–80, 281–2  
 singular, 824  
 suma, 822  
 transposición, 823  
 traslación, 245–6, 270
- Menús, 717–24
- Metaarchivo, 792
- Método vectorial (división de polígonos), 130–1
- Métodos de banda elástica, 696
- Métodos de compresión (archivos de imagen), 793–801
- Métodos de desplazamiento aleatorio del punto medio, 504–6
- Métodos de dibujo, 695–9
- Métodos de división de polígonos rotacional, 130–1
- Métodos de particionamiento del espacio (trazado de rayos):  
 adaptativo, 627  
 búfer de luz, 628–30  
 haces de rayos, 628–30, 631  
 uniforme, 627
- Métodos de posicionamiento (interactivos), 695
- Métodos de pseudocolor, 529
- Métodos de subdivisión:  
 árbol BSP, 492, 559–60  
 árbol octal, 489–92, 562–3  
 generación de fractales, 494–6  
 generación de *splines*, 472–4  
 trazado de rayos adaptativo, 632–3  
 trazado de rayos uniforme, 627
- Métodos del espacio de imagen (detección de visibilidad), 547
- Métodos interactivos de entrada, 689–94
- Métodos numéricos:  
 ajuste de datos por mínimos cuadrados, 846–7  
 algoritmo de Runge-Kutta, 844  
 descomposición LU, 840  
 determinación de raíz, 841–2  
 diferencias finitas, 845–6  
 ecuaciones diferenciales ordinarias, 844–5  
 ecuaciones diferenciales parciales, 845–6  
 ecuaciones no lineales, 841–2  
 eliminación gausiana, 839  
 evaluaciones de integral, 842–3  
 generador de números aleatorios, 843–4  
 método de bisección, 841  
 método de Euler, 844
- método de Gauss-Seidel, 840  
 método de la falsa posición, 842  
 método de Newton-Raphson, 841  
 métodos de Monte Carlo, 843  
 problemas de valor de contorno, 844  
 problemas de valor inicial, 844  
 regla de Cramer, 839  
 regla de Simpson, 842  
 regla del trapezoide, 842  
 sistemas de ecuaciones lineales, 839–40
- Mínimos cuadrados, ajuste de datos, 846–7
- Mip map (patrón de reducción de texturas), 655
- Modelado:  
 conceptos básicos, 779–82  
 coordenadas, 71, 784–5  
 coordenadas locales, 784–5  
 coordenadas maestras, 785  
 de módulos, 781  
 físico, 526–9  
 geométrico, 779–80  
 instancia, 779  
 jerarquías de símbolos, 781–2  
 jerárquico, 237, 784–7  
 paquetes, 782–4  
 representaciones, 779–81  
 símbolos, 779  
 tejidos, 528  
 transformaciones, 71–72, 237, 785–7
- Modelado sólido:  
 geometría sólida constructiva, 486–9  
 representaciones de barrido, 485–6
- Modelo básico de iluminación, 586–97
- Modelo de color, 738  
 aditivo, 740, 743  
 CMY, 745–7  
 CMYK, 746  
 conversión HSV-RGB, 749–50  
 conversión RGB-CMY, 746  
 conversión RGB-YIQ, 745  
 diagrama cromático, 740–1  
 funciones de ajuste de color, 738

- HLS, 750–1  
 HSB (igual que HSV)  
 HSV, 747–8  
 primarios, 738  
 primarios CIE estándar, 739–40  
 primarios CIE, 739–40  
 RGB, 42–44, 180–1, 742–4  
 sustractivo (CMY), 745–7  
 teoría de los tres estímulos de la visión, 742  
 valores de cromaticidad, 740  
 valores XYZ normalizados, 740  
 XYZ, 739–40  
 $YC_rC_b$ , 745  
 YIQ, 745  
 YUV, 745  
 Modelo de metabolismos, 430–1  
 Modelo de objetos suaves, 430–1  
 Modelos de iluminación, 577  
     ángulo de incidencia, 585–6  
     ángulo de reflexión especular, 588  
     ángulo de refracción, 598  
     atenuación angular de la intensidad, 581–3  
     atenuación radial de la intensidad, 579–80  
     coeficientes de reflexión, 585–88  
     coeficiente de reflexión ambiente, 586–7  
     coeficiente de reflexión difusa, 584–88  
     coeficiente de reflexión especular, 588–9  
     coeficiente de transparencia, 600  
     componentes de iluminación RGB, 594–6  
     consideraciones de color, 594–6  
     efectos atmosféricos, 601  
     efectos de iluminación de superficie, 583–4  
     efectos de refracción, 598–9  
     emisiones superficiales, 592–3  
     exponente de reflexión especular, 588–9  
     factor de opacidad, 600  
     focos, 580  
     Fresnel, leyes de la reflexión, 588–9  
     fuentes luminosas, 578–83  
     índice de refracción, 598  
     Lambert, ley del coseno, 585  
     ley de Snell, 598  
     luminancia, 596–7  
     luz ambiente (de fondo), 584–5  
     luz de fondo, 583–4  
     material opaco, 597  
     material translúcido, 597  
     modelo básico, 563–97  
     modelo de Phong, 588–93  
     múltiples fuentes luminosas, 592  
     parámetros de cámara, 602  
     radiosidad, 638–45  
     reflector ideal difuso, 585  
     reflector lambertiano, 585  
     reflexión difusa, 583–4, 584–8  
     reflexión especular, 583–4, 588–92  
     reflexiones combinadas difusaspecular, 592  
     sombras, 601–2  
     transmisión difusa, 598  
     transparencia, 597, 600  
     vector de la fuente luminosa, 586–7  
     vector de reflexión especular, 588  
     vector de transmisión, 599–600  
     vector del observador, 588  
     vector del punto medio, 590–2  
     Warn, fuente luminosa, 582–3  
 Modos de entrada, 694–5  
     por muestreo, 694  
     por solicitud, 694  
 Módulo, 781  
 Módulo (complejo), 826  
 Monitor, 35 (*véase también* Monitor de vídeo)  
 Monitor compuesto, 43  
 Monitor de vídeo, 35–58 (*véase también* Tubo de rayos catódicos)  
     alta definición, 38  
     barrido, 38–41  
     barrido aleatorio, 41–42  
     caligráfico, 41  
     color completo, 44  
     color real, 44  
     compuesto, 44  
     descarga de gas, 44  
     diseño de un TRC, 35–41  
     electroluminiscente, 44–46  
     electroluminiscente de película fina, 44  
     emisivo, 44  
     estaciones de trabajo, 54  
     estereoscópico, 47–50  
     LCD (pantalla de cristal líquido), 45–46  
     LED (diodo emisor de luz), 44–45  
     matriz activa, 46–47  
     matriz pasiva, 47  
     multi-panel, 54–58  
     no emisivo, 44  
     pantalla de plasma, 44, 45  
     pantalla de gran tamaño, 54–58  
     pantalla plana, 44–47  
     resolución, 38  
     RGB, 42–44  
     servidor gráfico, 69  
     TRC color, 42–44  
     TRC de almacenamiento, 36  
     TRC de refresco, 36–41  
     tridimensional, 47  
     vectorial, 41  
 Monitor en color: (*véase también* Monitor de vídeo)  
     compuesto, 44  
     delta-delta, 44  
     en línea, 43  
     máscara de sombra, 42–44  
     penetración de haz, 42  
     RGB, 42–44  
     sistema de color completo, 44  
     sistema de color real, 44  
 Monitores gráficos, 35–58 (*véase también* Monitor de vídeo)  
 Monte Carlo, métodos, 843  
 Morfismo, 27–28, 31, 762–4  
 Mosaico de superficies, 199–200  
 Movimiento browniano, 501–3  
     fraccional, 501–4  
 Movimiento rígido, 253–4  
 Movimientos de animación periódicos, 771–2  
 Muelle:  
     constante, 526–7  
     red (cuerpo no rígido), 526–7  
 Muestreo: (*véase también* Antialiasing)

- Muestreo (*cont.*)  
 adaptativo, 632  
 de área, 221, 224, 632  
 de líneas, 93, 220–3  
 de un segmento de línea, 93, 220–3  
 estocástico, 634–5  
 frecuencia de Nyquist, 221  
 intervalo de Nyquist, 221  
 máscaras de ponderación, 223  
 ponderado, 223  
 postfiltrado, 221  
 prefiltrado, 221  
 supermuestreo, 221  
 velocidad (animación), 771–2  
 Multum in parvo (mip map), 655
- N**
- Nabla, 829–30  
 al cuadrado, 832
- National Television System Committee (NTSC), 604, 743, 744
- NCSA (National Center for Supercomputing Applications), 70
- NCSA CAVE, 16, 688
- Newton, segunda ley del movimiento, 768–9
- Newton-Raphson, determinación de raíces, 841
- Nicholl-Lee-Nicholl, recorte de líneas de, 333–5
- Niveles cuánticos de energía, 37
- Niveles de intensidad del sistema, 603
- Normalizado:  
 coordenadas, 307  
 coordenadas de proyección, 361, 370–2, 379, 392–5  
 cuadrado, 311–3  
 visor, 310–1  
 volúmenes de visualización, 370–2, 379, 392–4 (*véase también* Recorte)
- NTSC (National Television System Committee), 604, 743, 744
- Número complejo:  
 conjugado, 826  
 fórmula de Euler, 826–7
- imaginario puro, 825  
 longitud de un vector, 826  
 módulo, 826  
 operaciones aritméticas, 825, 826  
 parte imaginaria, 824  
 parte real, 824  
 raíces, 827  
 representación mediante par ordenado, 824  
 representación polar, 826–7  
 valor absoluto, 826
- Número de línea de barrido, 86
- Número de vueltas, 132
- Número f, 631
- Número imaginario, 825  
 puro, 825
- NURB (Nonuniform rational B-spline), 467–9
- Nyquist:  
 frecuencia de muestreo, 221  
 intervalo de muestreo, 221
- O**
- Objeto:  
 componente de imagen, 156  
 no rígido (flexible), 526–9  
 opaco, 597  
 rígido, 768–9
- Objeto gráfico, 128  
 estándar, 128, 416
- Objetos sin forma, 429–31
- Open Inventor, 74
- OpenGL: (*véase también* GLUT; GLU)  
 AGL (interfaz Apple), 75  
 Architecture Review Board, 73–74  
 archivos de cabecera, 76  
 biblioteca básica, 74–75  
 biblioteca de núcleo, 74–75  
 bibliotecas relacionadas (GLUT y GLU), 75  
 bordes de texturas, 668–9, 678–9  
 búfer auxiliar, 151–2  
 búfer de acumulación, 187  
 búfer de color frontal-izquierdo, 150–1  
 búfer de patrones, 150–1, 187  
 búfer de profundidad, 150–1, 187
- búfer de refresco predeterminado, 150–1
- búfer de refresco, 78–79
- búfer de selección, 711
- búfer derecho, 150–1
- búfer frontal, 150–1
- búfer izquierdo, 150–1
- búfer trasero, 150–1
- búferes, 150–1
- búferes de color, 78–79, 150–1, 187
- cálculos de iluminación de caras traseras, 663–4
- coeficiente alpha, 183, 78
- coeficientes de atenuación radial de la intensidad, 661
- color de destino, 185
- color de la fuente, 185
- colores de la fuente luminosa, 660
- constantes simbólicas, 74–75
- coordenadas de textura, 669–70, 672–3, 673–4
- coordenadas de textura homogénea, 679–80
- coordenadas homogéneas, 89
- copia de patrones de textura, 675–6
- curvas de recorte, 484–5
- denominación de patrones de texturas, 676–7
- doble búfer, 150–1, 772–5
- efectos atmosféricos, 665
- espacio de texturas bidimensional (volumen), 672–3
- espacio de texturas tridimensional (volumen), 673–4
- espacio de texturas unidimensional (lineal), 669–72
- estados de textura actual, 676–7
- focos, 661–2
- fuente luminosa infinitamente distante, 659–60
- fuente luminosa local, 659–60
- fuentes de luminosas direccionales (focos), 661–2
- función de cara frontal, 216
- función de mapa de bits, 148–50
- función de mapa de píxeles, 150–1

- función de matriz de búfer de selección, 711–2
- función de proyección en perspectiva general, 398–9
- función de proyección en perspectiva simétrica, 398
- función de proyección ortogonal, 78–79, 397
- función de superficie cúbica (tetera), 425
- función de variación de la intensidad con la profundidad, 571
- función de ventana de recorte, 314–5, 396–7
- función de visor, 315, 399
- funciones de antialiasing, 228
- funciones de aproximación de *splines*, 474–85
- funciones de atributo de áreas de relleno, 211–6
- funciones de atributo de áreas poligonales de relleno, 211–6
- funciones de atributo de caracteres, 220
- funciones de atributo de líneas, 196–8
- funciones de atributo de puntos, 195–6
- funciones de búfer de profundidad, 568–71
- funciones de caracteres, 155
- funciones de color, 183–8
- funciones de composición de imágenes, 185
- funciones de consulta, 229
- funciones de curva B-spline, 480
- funciones de curva de Bézier, 474–7
- funciones de curvas, 103–4
- funciones de detección de visibilidad, 568–71
- funciones de eliminación de caras traseras, 569
- funciones de eliminación de polígonos (eliminación de caras ocultas), 569
- funciones de entrada interactiva, 699–724 (véase también GLUT)
- funciones de fuentes luminosas puntuales, 659–63
- funciones de iluminación, 659–67
- funciones de líneas, 91–92
- funciones de matriz de píxeles, 148–53
- funciones de menú, 717–24 (véase también GLUT)
- funciones de pila de nombres, 712–3
- funciones de proyección en perspectiva, 398–9
- funciones de puntos, 88–90
- funciones de relleno de áreas poligonales, 139–45, 416
- funciones de representación de superficies, 667–8
- funciones de respuesta a la visualización, 81
- funciones de superficie B-spline, 482–4
- funciones de superficie de Bézier, 477–80
- funciones de tetera, 425
- funciones de textura lineal, 669–72
- funciones de textura, 213, 668–80
- funciones de texturas de superficies, 672–3
- funciones de texturas volumétricas, 673–4
- funciones de transformación geométricas, 292–9
- funciones de transparencia, 666
- funciones de visualización bidimensional, 78–79, 314–22
- funciones de visualización tridimensional, 396–401
- funciones para poliedros, 416–9
- funciones para poliedros regulares, 416–9
- funciones para superficies cuádricas, 424–5, 425–8 (véase también GLUT; GLU)
- gestión de ventanas de visualización (véase también GLUT), 76–79, 158–61, 315–20, 399
- GLX (extensión a X Windows), 75
- grupos de atributos, 229–30
- identificadores de selección, 711
- indicadores de aristas de polígonos, 214–6
- interfaz Apple (AGL), 75
- interfaz de sistema XWindow (GLX), 75
- interfaz IBM/OS (PGL), 75
- listas de interpolación de vectores normales, 667–8
- listas de visualización, 156–8, 787
- matrices de color, 186–7
- matrices de coordenadas de textura, 676
- matrices de vértices, 145–8, 667–8
- matriz actual, 294
- matriz entrelazada, 186
- matriz modelview, 293–4
- métodos alámbricos, 214–6
- métodos de relleno por interpolación, 213
- métodos de visibilidad alámbrica, 570–1
- mezcla de color, 78–79, 185, 666
- mip maps, 677–8
- modelado jerárquico, 787
- modelo de iluminación, 665
- modo de color, 293–4
- modo de índice de colores, 184–5
- modo de proyección, 78–79, 293–4, 314
- modo de realimentación, 711–2
- modo de representación predeterminado, 199, 666–7
- modo de selección, 711
- modo de textura, 293–4
- modo de visualización de color, 183
- modos de representación, 667–8
- modos RGB y RGBA, 183–8
- nombres de texturas, 676–7
- opciones de color de texturas, 674
- opciones de mapeado de texturas, 674–5
- opciones de textura, 680
- operaciones de barrido, 151–2
- operaciones de selección, 711–7 (véase también GLU)

OpenGL (*cont.*)  
 operaciones de semitonos, 668  
 parámetros de iluminación,  
   659–67  
 parámetros de iluminación  
   ambiente (fondo), 662–3  
 parámetros de iluminación de  
   fondo, 662–3  
 parámetros de iluminación de  
   superficies, 662–5  
 parámetros de reflexión especular,  
   662–3  
 parámetros globales de ilumina-  
   ción, 662–3  
 patrones de reducción de texturas,  
   677–8  
 patrones de textura reducidos,  
   677–8  
 PGL (interfaz IBM OS/2  
   Presentation Manage), 75  
 pila de atributos, 230  
 pila de atributos de cliente, 230  
 pila de atributos de servidor, 230  
 pila de nombres, 711–2  
 pilas de matrices, 295–6  
 plano de recorte lejano, 396–7  
 plano de recorte próximo, 396–7  
 planos de recorte, 396–7, 409–10  
 planos de recorte opcional, 409  
 posición de barrido actual, 148  
 primitivas, 78–79  
 procedimientos de animación,  
   772–5  
 programa de ejemplo introducto-  
   rio, 80  
 propiedades de la fuente lumino-  
   sa, 659  
 proxy de textura, 679  
 rampa de colores, 228  
 reinicialización de texturas, 675–6  
 representación de superficies de  
   intensidad constante, 667–8  
 representación de superficies por  
   el método de Gouraud, 667–8  
 representación plana de superfi-  
   cies, 667–8  
 sintaxis básica, 74–75  
 sistema de referencia bidimen-  
   sional, 88

subconjunto de procesamiento de  
   imágenes, 185  
 subpatrones de textura, 677  
 texturado automático, 679  
 texturas proxy, 679  
 tipos de datos, 74–75  
 tipos de fuentes luminosas, 660  
 transformaciones de barrido, 266  
 transformaciones de proyección  
   ortogonal, 397  
 Utility (GLU), 75  
 Utility Toolkit (GLUT), 75  
 variables de estado, 180  
 vector de posición del observador,  
   396–7  
 vectores unitarios normales a la  
   superficie, 667–8  
 ventana de selección, 712–3  
 WGL (interfaz Windows), 75  
 Operaciones de selección en  
   OpenGL, 711–7  
 Operaciones lógicas, 200–1  
 Operador laplaciano, 831–2  
 Operadores diferenciales, 829–32  
   direccional, 831  
   divergencia, 832  
   gradiente, 830–1, 831  
   Laplace, 831–2  
   ordinario, 829–30  
   parcial, 830  
   rotacional, 832  
 Orden (continuidad de curvas  
   *spline*), 434–5  
 Ortogonales, coordenadas, 812–3  
 Overhauser, *spline*, 441–2  
**P**  
 Paneles táctiles, 64–66  
   acústico, 66  
   LED, 64  
   óptico, 64  
   resistivo, 64–66  
 Pantalla  
   caligráfica (vectorial), 41  
   de cristal líquido (LCD), 45–47  
   de panel de plasma, 44  
   de trazo (vectorial), 41  
   de vídeo, 35 (*véase también*  
     Monitor de vídeo)

electroluminiscente, 44  
 electroluminiscentes de película  
   fina, 44  
 emisiva (emisor), 44  
 multipanel, 54–58  
 no emisiva (no emisor), 44  
 procesador de, 53–54  
 vectorial, 41  
 Pantalla de barrido (monitor), 38–41  
   aleatorio, 41–42  
   bitmap, 40  
   búfer de color, 39  
   búfer de imagen, 39  
   búfer de refresco, 39  
   color, 40  
   entrelazado, 40  
   escala de grises, 40  
   imagen, 39  
   línea de barrido, 39  
   monocromo, 40  
   píxel, 39  
    pixmap, 40  
   planos de bit, 40  
   profundidad, 40  
   relación de aspecto, 39  
   resolución, 39  
   retorno de haz, 40  
   retrazado horizontal, 40  
   retrazado vertical, 40  
   velocidad de refresco, 40–41  
 Pantalla, coordenadas de  
   bidimensional, 51–53, 86–87, 809  
   tridimensional, 87, 395, 812  
 Pantallas (control de iluminación),  
   582–3  
 Pantallas de panel plano, 44–47  
   de cristal líquido (LCD), 45–47  
   de descarga de gas, 44  
   diodo emisor de luz (LED), 44–45  
   electroluminiscente de película  
    fina, 44  
    emisiva, 44  
   matriz activa, 46–47  
   matriz pasiva, 47  
   no emisiva, 44  
   plasma, 44–45  
 Parábola, 121–2  
 Parámetro  
   beta, 464–5

- de continuidad, 444–5  
 de desplazamiento (*spline*), 444–5, 454, 466  
 de grado (B-*spline*), 454–5  
 de tensión (*spline*), 441–2, 461, 466  
**Parámetros de atributo**, 73, 179  
 áreas de relleno, 199–202  
 caracteres y texto, 217–20  
 color, 180–1  
 escala de grises, 182  
 estilos de pincel y pluma, 191–2, 194–5  
 líneas curvas, 193–5  
 lista de sistemas, 179  
 polígonos, 199–200  
 puntos, 188  
 segmento lineal, 188–92  
 símbolos marcadores, 220  
**Parte real** (número imaginario), 824  
**Particionamiento de imágenes**, 156  
**Paseo aleatorio**, 501–4  
**Patrones de píxel** (semitono), 607  
**Patrones de reducción** (mapeado de textura), 655  
**Patrones de textura volumétricos**, 654–5  
**PCX**, 805  
**Peano**:  
 curva, 496–7, 498–9  
 espacio, 498–9  
**Pel**, 39  
**Penumbra**, 635, 637  
**Período** (onda luminosa monocromática), 735  
**Persistencia**, 38  
**Personajes articulados** (animación), 769–71  
**PGL** (interfaz de Presentation manager con OpenGL), 75  
**PHIGS** (Programmer's Hierarchical Interactive Graphics Standard), 73  
**PHIGS+**, 73  
**Phong**:  
 modelo de reflexión especular, 588–93  
 representación de superficies, 617  
**PICT**, 804  
**Pincel y pluma**, atributos, 191–2  
**Pipeline de visualización**:  
 bidimensional, 305–7  
 tridimensional, 360–1  
**Pirámide de visión** (proyección en perspectiva), 383  
**Pitteway-Watkins**, antialiasing, 227–8  
**Pixblt** (transferencia de bloques de píxeles), 265  
**Píxel**, 39  
**Píxel**, coordenadas del, 86–87, 124–5  
**Pixmap**, 40, 148  
**Planck**, constante, 638–9  
**Planificación de tareas**, 5  
**Plano**:  
 de partida (trazado de rayos), 488  
 cara anterior, 137–8  
 cara posterior, 137–8  
 coeficientes, 136–8  
 complejo, 824  
 ecuaciones, 136–8  
 frontal (recorte), 370  
 lejano (recorte), 370–2  
 parámetros, 136–8  
 posterior (recorte), 370  
 próximo (recorte), 370–2  
 recorte, 370–2, 407–9  
 vector normal, 138–9  
**Plano de visualización**, 356, 361, 362  
 posición, 362  
 vector normal, 362  
 ventana de recorte, 370, 377–8  
**Planos de bits** (sistema de barrido), 40  
**Planos de recorte**, 361  
 orientaciones arbitrarias, 407–9  
 próximo y lejano, 370–2  
**PNG**, 803–4  
**Polares**, coordenadas, 788–89, 813–4  
**Poliedros**, 416  
 intersecciones en trazado de rayos, 625–6  
 propiedades, 838  
**Polígono**, 129 (véase también Área de relleno)  
 ángulo convexo, 131–2  
 área, 836  
 arista, 128  
 atributos, 199–200  
 cara frontal, 137–8  
 cara posterior, 137–8  
 característico, 433  
 centroide, 836–7  
 clasificaciones, 129  
 convexo, 129  
 de control, 433  
 degenerado, 129  
 división de polígonos cóncavos, 130–1  
 división de polígonos convexos, 131–2  
 ecuaciones del plano, 136–8  
 efectos de costura, 214  
 estándar, 128  
 indicador de arista (OpenGL), 214–6  
 intersección con rayos, 625–6  
 lado, 128  
 lista de aristas activas, 205–6  
 malla, 128  
 métodos de relleno, 199–206  
 (véase también Algoritmos de relleno de área)  
 parámetros del plano, 136–8  
 simple, 128  
 tabla de aristas, 135–6  
 tabla de caras de la superficie, 135–6  
 tabla de datos geométricos, 135–6  
 tabla de vértices, 135–6  
 tablas, 135–6  
 tablas de aristas ordenada, 204–5  
 test interior-exterior, 132–4 (véase también Plano)  
 vector de la arista, 129–30  
 vector normal, 138–9  
 vértices, 128  
**Polígono cóncavo**, 129  
 división, 130–1  
 identificación, 129–30  
 recorte, 343–5  
**Polilínea**, 91, 160–62  
 algoritmos de visualización, 100  
 unión en bisel, 190

- Polilínea, 91, 160–62 (*cont.*)  
 unión en punta, 190  
 unión redondeada, 190
- Polimarcador, 153–4, 160–62
- Polinomios de Bernstein, 445–6
- Ponderación:  
 máscara (píxel), 223  
 superficie, 224–5
- Posición actual, 87
- Posición actual de barrido  
 (OpenGL), 148
- Postfiltrado, 221 (*véase también Antialiasing*)
- Precisión (atributo de texto), 220
- Prefiltrado, 221 (*véase también Antialiasing*)
- Presentation Manager, interfaz con OpenGL (PGL), 75
- Primitivas de salida gráfica, 72, 85  
 área de relleno, 124–39  
 carácter, 153–4  
 círculo, 104–11  
 elipse, 111–20  
 mapa de bits, 148  
 mapas de píxel, 148  
 marcador, 153–4  
 matrices de píxel, 148  
 polilínea, 91, 160–62  
 polimarcador, 153–4, 160–62  
 polinomio, 123  
 punto, 86–87  
 sección cónica, 120–3  
 segmento lineal, 92–102  
*spline*, 123  
 texto, 153–4
- Primitivas para matrices de píxeles, 148
- Problema del valor de contorno, 844
- Problema del valor inicial, 844
- Procedimiento registrado, 80
- Procesamiento de imágenes, 31–32
- Producto escalar (vectores), 816–7
- Producto vectorial (vectores), 817–8
- Profundidad (búfer de imagen), 40
- Profundidad de campo, 632
- Programas de dibujo, 23–26
- Propiedades de coherencia, 203
- Protocolo de transferencia de archivos (ftp), 70
- Proyección:  
 axonométrica, 369  
 caballera, 376  
 cabinet, 376  
 centro, 380  
 frustum, 384–5, 386–92  
 isométrica, 369  
 plano, 356, 549  
 punto de referencia, 380  
 vector (oblicua), 376–7  
 volumen de visualización, 370–2,  
 377–8, 383–5
- Proyección de rayos  
 detección de superficies visibles,  
 563–4  
 geometría sólida constructiva,  
 486–9
- Proyección en perspectiva, 356, 368,  
 379–80  
 ángulo del campo visual, 387–9  
 casos especiales, 381–3  
 centro, 380  
 cono de visión, 383  
 coordenadas de transformación  
 normalizadas, 393–5  
 coordenadas, 381–3  
 de dos puntos, 383  
 de tres puntos, 383  
 de un punto, 383  
 frustum, 384–5, 386–92  
 frustum oblicuo, 390–2  
 frustum simétrico, 386–90  
 matriz, 385–6, 392–5  
 matriz de inclinación, 391  
 matriz de transformación normali-  
 zada, 394–5  
 matriz de transformación oblicua,  
 392–3  
 pirámide de visión, 383  
 punto de fuga principal, 383  
 punto de referencia, 380  
 puntos de fuga, 383  
 representación en coordenadas  
 homogéneas, 385–6, 393–4  
 volumen de visualización norma-  
 lizada, 392–4  
 volumen de visualización, 383–5
- Proyección en perspectiva oblicua:  
 frustum, 390–2
- matriz, 392–3  
 matriz de inclinación, 391
- Proyección ortogonal:  
 axonométrica, 369  
 coordenadas de proyección, 369  
 elevaciones, 368  
 isométrica, 369  
 matriz de transformación de nor-  
 malización, 373  
 transformación de normalización,  
 370–2  
 volumen de visualización, 370–2
- volumen de visualización norma-  
 lizado, 372
- vista plana, 368
- Proyección paralela, 356, 368  
 axonométrica, 369  
 caballera, 376  
 cabinet, 376  
 ejes principales, 369  
 isométrica, 369  
 oblicua, 374–9  
 ortogonal, 368–73  
 transformación de inclinación,  
 374–6  
 transformación de normalización,  
 372–3, 379  
 vector, 376–7  
 vista de elevación, 368  
 vista superior, 368  
 volumen de visualización, 370–2,  
 377–8
- Proyección paralela oblicua:  
 coordenadas, 374  
 en dibujo y diseño, 374–6  
 matriz de transformación, 377–8  
 proyección caballera, 376  
 proyección cabinet, 376  
 transformación de normalización,  
 379  
 vector, 376–7  
 volumen de visualización, 377–8
- Puesta en fase de los píxeles, 221,  
 225
- Punto:  
 atributos, 188  
 de foco (elipse), 111–2  
 de fuga principal, 383  
 de fuga, 383

- de pivote (rotación), 240  
 de referencia (proyección en perspectiva), 380  
 de referencia de visualización, 380  
 en el espacio de coordenadas, 814  
 fijo (cambio de escala), 242, 284–5  
 observado, 362–3  
 propiedades, 814  
 recorte, 323  
 Pureza (luz), 737
- R**
- Radiancia espectral, 638–9  
 Radiante:  
 energía (radiancia), 638–9  
 exitancia, 638–9  
 flujo, 638–9, 639–40  
 intensidad, 638–9, 639–40  
 potencia, 638–9  
 transferencia de energía, 640–3  
 Radiosidad (exitancia radiante), 638–45  
 cierre superficial, 640–1  
 ecuación, 640–1  
 factor de reflectividad, 640–1  
 factores de forma, 640–1  
 modelo básico, 638–43  
 refinamiento progresivo, 643–5  
 semicubo, 642–3  
 semiesfera, 639–40  
 Raíces:  
 de números complejos, 826–7  
 ecuaciones no lineales, 841–2  
 Ratón, 59–60  
 Z, 60  
 Rayo de sombra, 622  
 Rayo del píxel, 618–619  
 Rayo secundario, 620  
 Rayos catódicos, 36  
 Realidad artificial (*véase* Realidad virtual)  
 Realidad virtual:  
 aplicaciones, 10–11  
 lenguaje de modelado (VRML), 70  
 sistemas, 47–50, 62–63  
 Realimentación de eco, 695  
 Recorte de área, 336–46  
 Recorte de línea:  
 bidimensional, 323–36  
 Cohen-Sutherland, 324–9  
 Cyrus-Beck, 330  
 Liang-Barsky, 330–3  
 Nichol-Lee-Nichol, 333–5  
 tridimensional, 404–7  
 ventana de recorte no lineal, 335  
 ventana de recorte no rectangular, 335  
 Recorte de polígonos:  
 bidimensionales, 336–46  
 métodos paralelos, 338–9  
 Sutherland-Hodgman, 338–43  
 tridimensionales, 406–8  
 Weiler-Atherton, 343–5  
 Recorte de superficies, 406–8  
 Recuadro:  
 de entrada de dial, 58, 59  
 dimensión, 495–6  
 filtro, 225–6  
 recubrimiento (fractal), 495–6  
 Redes gráficas, 69  
 Reducción de color:  
 de corte medio, 793  
 por popularidad, 792–3  
 uniforme, 792  
 Refinamiento progresivo (radiosidad), 643–5  
 Reflectividad, 585  
 Reflector:  
 difuso ideal, 585, 639–40  
 lambertiano, 585, 639–40  
 perfecto, 585, 639–40  
 Reflexión (luz):  
 ángulo de incidencia, 585–6  
 coeficientes, 584–88, 588–9  
 difusa, 584–88  
 especular (modelo de Phong), 588–92  
 lambertiana, 585  
 leyes de Fresnel, 588–9  
 rayo, 620  
 vector del punto medio, 590–2  
 Reflexión especular, 583–4, 588–92  
 ángulo, 588  
 coeficiente, 588–9  
 exponente, 588–9  
 leyes de Fresnel, 588–9  
 modelo de Phong, 588–93  
 vector, 588  
 vector del punto medio, 590–2  
 Refracción:  
 ángulo, 598  
 coeficiente de transparencia, 600  
 difusa, 597  
 doble, 598  
 índice, 598  
 ley de Snell, 598  
 rayo, 598–9, 620  
 vector, 599–600  
 Refresco:  
 archivo de visualización, 42  
 búfer, 39 (*véase también* Búfer de imagen)  
 TRC, 36–41 (*véase también* Tubo de rayos catódicos)  
 velocidad (TRC), 36, 40–41  
 Regla de la mano derecha, 817–8  
 sistema cartesiano, 71, 811  
 Regla de la mano izquierda, coordenadas cartesianas, 812–3  
 Regla de paridad (par-impar), 132  
 Regla del número de vueltas distinto de cero, 132–4  
 Regla del trapezoide, 842  
 Regla par-impar de relleno de polígonos, 132  
 Reglas de producción, 521–2  
 Relación de aspecto, 39  
 Relativas, coordenadas, 87  
 Relleno  
 de área, 202–11 (*véase también* Algoritmos de relleno de área)  
 mediante cuadrícula, 200  
 mediante patrones, 212–3  
 suave, 200  
 tintado, 200  
 Representación, 358 (*véase también* Representación de superficies)  
 de volúmenes, 530–2, 532–4  
 mediante producto cartesiano (*spline* superficial), 436, 454, 464  
 plana de superficies, 614  
 rápida de superficies por el método de Phong, 617–8

Representación de funciones:  
 explícita, 828–9  
 implícita, 828–9

Representación de superficies, 358, 577, 613–45 (*véase también*  
 Detalles de superficie)  
 bandas de mach, 617  
 Gouraud, 614–7  
 intensidad constante, 614  
 interpolación de intensidad, 614–7  
 interpolación de vectores normales, 617  
 mapeado de fotones, 646–7  
 métodos poligonales, 613–8  
 Phong, 617  
 Phong rápida, 617–8  
 plana, 614  
 radiosidad, 638–45  
 trazado de rayos, 618–38

Representaciones de barrido, 485–6

Representaciones de objetos:  
 árboles BSP, 492  
 árboles cuaternarios, 489  
 árboles octales, 489–92  
*B-splines*, 454–64  
 contorno (B-rep), 415  
 curvas y superficies fractales,  
     492–521  
 elipsoide, 420–1  
 esfera, 420  
 funciones de densidad, 429–31  
 graftal, 524  
 gramática L, 524  
 gramáticas de forma, 521–4  
 interpolación de *splines* cúbicos,  
     437–44  
 metabolas, 430–1  
 métodos CSG, 486–9  
 métodos de particionamiento  
     espacial, 415  
 modelado físico, 526–9  
 modelo de objeto suave, 430–1  
 objeto gráfico estándar, 128, 416  
 poliedros, 416  
 polígonos, 128–39  
 relieves gausianos, 430  
 representaciones de barrido, 485  
 sistemas de partículas, 524–5  
*splines*, 431–73

*splines* beta, 465–7  
*splines* de Bézier, 445–54  
*splines* racionales, 467–9  
 supercuádricas, 422–4  
 superficies cuádricas, 420–2  
 superficies sin forma, 429–31  
 toro, 421–2  
 visualización de datos, 529–37

Representaciones de particionamiento del espacio, 415

Representaciones no paramétricas, 828–9

Representaciones paramétricas, 829  
 círculo, 104–11, 829–32  
 curva, 828–9  
 elipse, 113  
 elipsoide, 421  
 esfera, 420, 829–32  
 parábola, 121  
 segmento lineal, 323–4, 330,  
     380–1  
*spline*, 123, 434, 435, 436  
 superficie, 829–32  
 toro, 422

Resaltado (método de selección), 693

Resolución:  
 aproximaciones de semitonos,  
     608–9  
 dispositivo de visualización, 38

Resolución de ecuaciones integrales:  
 aproximaciones mediante rectángulo, 842–3  
 métodos de Monte Carlo, 843  
 regla de Simpson, 842  
 regla del trapezoide, 842

Resolución de ecuaciones lineales:  
 descomposición LU, 840  
 eliminación gausiana, 839  
 Gauss-Seidel, 840  
 regla de Cramer, 839

Resolución de ecuaciones no lineales, 842–3

Resolución de sistemas de ecuaciones, 839–40

Retrazado horizontal, 40  
 Retrazado vertical, 40

Restricciones (dibujo interactivo), 696

Retorno (haz de electrones), 40

REYES, 553

RGB:  
 coordenadas cromáticas, 743  
 modelo de color, 42–44, 180–1,  
     742–4  
 monitor, 44 (*véase también*  
     Monitor de video)

Rotación:  
 ángulo, 240  
 bidimensional, 240–2, 245–6,  
     254–5  
 composición, 248  
 construcción de matrices, 254–5  
 de los ejes de coordenadas, 272–4  
 eje de coordenadas, 272–4  
 eje de, 240  
 eje espacial general, 274–80  
 eje *x*, 273  
 eje *y*, 274  
 eje *z*, 272–3  
 en las animaciones, 252–3, 771–2  
 inversa, 246–7, 274  
 métodos de barrido, 265  
 métodos de cuaternios, 280–4  
 punto de pivote, 240  
 representación matricial, 246  
 tridimensional, 271–84  
 vector del eje, 275–6

Rotacional, operador, 832

Ruido (tramado), 611

Runga-Kutta, algoritmo, 844

**S**

Sans-serif, tipo de letra, 153

Saturación (luz), 737

Secciones cónicas, 120–3

Segmento (subsección de imagen), 156

Segmento de línea dirigido (vector), 814

Seguimiento (animación), 760

Selección:  
 coordenadas, 692  
 dispositivo de entrada, 690, 692–4  
 distancia, 692–4  
 objeto, 692–4  
 resaltado, 693  
 ventana, 693

- Semicubo (radiosidad), 642–3  
 Semiesfera (radiosidad), 639–40  
 Semitonos:  
     aleatorización, 610–3  
     aproximaciones, 607–10  
     métodos de color, 610  
     patrones, 607  
 Serif, tipo de letra, 153  
 Servidor gráfico, 69  
 setPixel, procedimiento, 87  
 SIGGRAPH (Special Interest Group in Graphics), 70  
 Símbolo:  
     instancia, 779  
     jerarquías, 781–2  
     modelado, 779  
 Simetría  
     círculo, 104  
     elipse, 113  
     en los algoritmos de dibujo de curvas, 121  
 Simpson, regla de, 842  
 Simulaciones, 6–7, 10–11, 12–23  
 Simuladores, 19–23  
     de vuelo, 19–22  
 Sistema de color completo, 44  
 Sistema de animación  
     parametrizado, 761  
     por script, 761  
 Sistema de barrido, 50–54  
     codificación de celdas, 53  
     codificación de longitud de recorrido, 54  
     controlador de pantalla, 50–51  
     controlador de vídeo, 51–53  
     controlador gráfica, 53  
     conversión de barrido, 53  
     coprocesador de pantalla, 53  
     procesador de pantalla, 53–54  
 Sistema de enfoque (TRC), 37  
 Sistema de fotogramas clave, 760, 762–7  
 Sistemas de entrada de voz, 67  
 Sistemas de pantalla grande, 54–58  
 Sistemas de partículas, 524–5  
 Snell, ley de, 598  
 Software gráfico: (*véase también OpenGL*)  
     bidimensional, 73  
 CG API, 70  
 correspondencia de lenguaje, 74  
 estándares, 73–74  
 funciones básicas, 72–73  
 GKS, 73–74  
 GL, 70, 73  
 Open Inventor, 74, 75  
 operaciones de control, 73  
 PHIGS, 73  
 PHIGS+, 73  
 pipeline de visualización, 71–72  
 Renderman, 74  
 representaciones de coordenadas, 70–72  
 tridimensional, 73  
 VRML, 70, 74  
 Sombra, 635, 637  
 Sombras (color), 739  
 Sombras:  
     modelado, 601–2  
     penumbra, 635, 637  
     sombra, 635, 637  
 Sombreado (*véase Modelos de iluminación; Representación de superficies*)  
 Spaceball, 11, 60–61  
 SpaceGraph, sistema, 47  
 SPIFF, 802–3  
*Spline* cardinal, 441–4  
*Spline* Catmull-Rom, 441–2  
*Spline* cúbica, 123, 435  
     beta, 466  
     Bézier, 451–3  
     *B-spline*, 460–1  
     interpolación, 437–44  
     natural, 438  
*Spline* de aproximación, 432  
*Spline* de interpolación, 432  
*Spline* hermítica, 438–41  
*Spline* racional, 467–9  
 Stoke, teorema de, 833  
 Subdivisión espacial adaptativa:  
     árbol BSP, 492  
     trazado de rayos, 627  
 Subdivisión espacial uniforme:  
     árbol octal, 489–92  
     trazado de rayos, 627  
 Suceso:  
     cola, 694  
 modo de entrada, 694  
 Supercuádrica, 422–4  
 Superelipse, 422–4  
 Superelipsoide, 423  
 Superficie:  
     circundante, 560  
     cuádrica, 420–2  
     curva de ajuste, 437  
     detalle, 647–49  
     exterior, 560  
     fractal, 498–9, 500–1, 504–6, 519  
     interior, 560  
     plana, 416  
     ponderación, 224–5  
     representación explícita, 828–9  
     representación implícita, 828–9  
     representación no paramétrica, 828–9  
     representación paramétrica, 829  
     sin forma, 429–31  
     solapada, 560  
     supercuádrica, 422–4  
     teselada, 128  
 Superficie curva, 420–2, 436, 530–2  
     *B-spline*, 464–5  
     cuádrica, 420–2  
     curva de ajuste, 468–9  
     elipsoide, 420–1  
     esfera, 420  
     gráficas de contorno, 530–2  
     isosuperficie, 530–2  
     ponderación, 224–5  
     representación (*véase Representación de superficies*)  
     representación no paramétrica, 828–9  
     representación paramétrica, 829  
     representaciones explícitas, 828–9  
     representaciones implícitas, 828–9  
     *spline*, 436 (*véase también Superficie spline*)  
     *spline* de Bézier, 454  
     supercuádrica, 422–4  
     toro, 421–2  
     visibilidad, 565–6 (*véase también Detección de superficies visibles*)  
 Superficie *spline*, 432, 436  
     ajuste, 437

Superficie *spline*, 432, 436 (*cont.*)

- Bézier, 454
  - B-spline, 464–5
  - representación mediante producto cartesiano, 436
  - visualización, 470–4
- Supermuestreo, 221, 632
- Sutherland-Hodgman, recorte de polígonos, 338–43

## T

TAC (Tomografía axial computerizada), 31

Tablas:

- de atributos (polígono), 135–6
- de aristas, 135–6, 204–5, 205–6
- de aristas ordenadas, 204–5
- de caras de la superficie, 135–6
- de consulta de vídeo, 604
- de sustitución, 181–2, 604
- de sustitución de colores, 181–2
- de vértices (polígono), 135–6
- geométrica, 135–6

Tableta, 62–63 (*véase también Digitalizador*)

- de datos, 62
- gráfica, 62–63

Tamaño en puntos (carácter), 217

Targa, 805

TCP/IP, 69–70

Teclado, 58

Técnicas de construcción interactiva

- de imágenes, 695–9
- arrastre, 696
- campo de gravedad, 698–9
- cuadrículas, 696
- dibujo, 699
- métodos de banda elástica, 696
- posicionamiento, 695
- restricciones, 696

Temporización (animación), 759

Tensor, 818

- contracción, 536
- dimensión, 818
- métrico, 821
- propiedades de transformación, 818
- rango, 818
- visualización de datos, 535–6

Teoremas diferenciales, 833–5

Teoría de los tres estímulos de la visión, 742

Terminación

- cuadrada saliente, 190
- de línea redondeada, 189
- plana, 189

Terreno (fractal), 505–7

Teselado de superficies, 128

Test interior-exterior:

- planos espaciales, 137–9
- regla del número de vueltas distintas de cero para polígonos, 132–4
- regla par-impar para polígonos, 132

Tetera, 425

Texel, 650–1

Texto: (*véase también Carácter*)

- alineación, 220
- atributos, 217–20
- precisión, 220
- recorte, 347–8
- trayectoria, 218–9

Textura sólida, 654–5

TGA, 805

TIFF, 803

Tintes (color), 739

Tipo de letra, 153 (*véase también Fuente*)

- legible, 153

Tomografía, 31

- axial computerizada (TAC), 31
- computerizada (TC), 31
- emisiva de posición (TEP), 31

Tonalidades (color), 739

Tono, 737

Topológico

- dimensión, 495–6
- recubrimiento (fractal), 496

Toro, 421–2

Trackball, 60–61

Tramado, 610

- aleatorio (ruido), 611

matriz, 611

- método de difusión de errores, 612–3

- método de difusión de puntos, 613

método de tramado ordenado, 611

ordenado, 611

ruido, 611

Transferencia de bloques, 265

Transformación:

- afin, 292
- cambio de escala, 242–4, 248
- compuestas, 247–59, 287–9
- comutativa, 251–2
- cuadro rígido, 238–9, 253–4
- de barrido, 265–6
- de coordenadas universales a coordenadas de visualización, 71, 307, 308, 361, 366–7
- de la estación de trabajo, 313
- de ventana a visor, 306, 310–3
- de visor, 395
- eficiencia de cálculo, 252–3
- geométrica bidimensional, 238–69
- geométrica tridimensional, 270–1
- geométrica, 73, 237
- geométricas básicas, 238–44
- inversa, 246–7
- métodos de barrido, 265–6
- modelado, 73, 237
- no comutativa, 251–2
- representación matricial, 244–65, 270–91
- rotación, 240–2, 246, 271–84
- sistema de coordenadas, 267–9, 291–2
- traslación, 238–40, 245–6, 270–1

Transformación de inclinación,

- 263–4
- bidimensional, 263–4
- dirección *x*, 263–4
- dirección *y*, 264
- dirección *z*, 290–1
- eje, 263–4, 290–1
- en proyecciones en perspectiva oblicuas, 391–2
- en proyecciones paralelas oblicuas, 374–6
- matriz, 263–4, 290–1
- parámetros, 263–4, 264, 290–1
- tridimensional, 290–1

Transformación de normalización:

- bidimensional, 310–3

- proyección en perspectiva, 393–4

- proyección ortogonal, 370–2  
 Transformación de proyección:  
     axonométrica, 369  
     caballera, 376  
     cabinet, 376  
     ejes principales, 369  
     isométrica, 369  
     ortogonal, 368–73  
     ortográfica, 368  
     paralela, 356, 368, 368–79  
     paralela oblicua, 374–9  
     perspectiva, 356, 368, 379–95  
     perspectiva oblicua, 390–2  
     perspectiva simétrica, 386–90  
 Transformación de reflexión, 260–3  
     bidimensional, 260–3  
     eje, 260, 290  
     métodos de barrido, 266–7  
     plano, 290  
     tridimensional, 290  
 Transformación de visualización, 73  
     (véase también Proyección)  
     algoritmos de recorte, 323–48,  
     401–10  
     bidimensional, 306  
     coordenadas de pantalla, 51–53,  
     86–87, 395, 809  
     coordenadas de proyección nor-  
     malizadas, 361, 372, 379,  
     392–5  
     coordenadas normalizadas, 307  
     coordenadas uvn, 363–4  
     cuadrado normalizado, 311–3  
     efectos de pantalla dividida, 313,  
     320–2  
     frustum, 384–5, 386–92  
     parámetros de la cámara, 355,  
     360–1  
     pipeline, 305–7, 360–1  
     plano de proyección, 356, 361,  
     362  
     plano de recorte lejano, 370–2  
     plano de recorte próximo, 370–2  
     plano de visualización, 356, 361–2  
     planos de recorte, 361  
     proyección en perspectiva,  
     379–95  
     proyección en perspectiva  
     oblicua, 390–2  
     proyección en perspectiva simétr-  
     ica, 386–90  
     proyección ortogonal, 368–73  
     proyección paralela oblicua,  
     374–9  
     proyección paralela, 368–9  
     proyecciones, 356  
     punto de vista, 362  
     transformación de coordenadas  
     universales a coordenadas de  
         visualización, 308, 366–7  
     transformación de estación de tra-  
     bajo, 313  
     transformación de visor, 395  
     tridimensional, 355–60  
     vector de visualización vertical,  
     362  
     ventana de recorte, 305, 307–9  
     ventana de visualización, 305  
     visualización normalizado, 310–1  
     volumen de visualización norma-  
     lizado, 372, 379, 392–4  
     volumen de visualización, 361  
 Transmisión especular (refracción),  
     598  
 Transparencia: (véase también  
     Refracción; Trazado de rayos)  
     coeficiente, 600  
     factor, 200  
     ley de Snell, 598  
     modelo básico, 600  
     vector, 599–600  
 Transpuesta (matriz), 823  
 Traslación:  
     bidimensional, 238–40, 245–6  
     composición, 248  
     distancias, 238  
     inversa, 246–7, 271  
     métodos de barrido, 265  
     representación matricial, 245–6,  
     270  
     tridimensional, 270–1  
     vector, 238  
 Trayectoria (texto), 218–9  
 Trazado de rayos, 619  
     algoritmo básico, 620–2  
     ángulo de refracción, 622  
     *antialiasing*, 632–3  
     árbol, 620  
     cálculos de intersección con  
     superficies, 623–30  
     códigos, 634–5  
     de distribución, 634–5  
     desenfoque de movimiento, 635–7  
     ecuación del rayo, 622  
     efectos de enfoque de cámara,  
     630–2  
     efectos del objetivo de la cámara,  
     634–5  
     en el modelo de radiosidad, 643–5  
     fluctuación, 634–5  
     fuente de luz compleja, 635  
     haces, 628–30, 631  
     intersecciones con la esfera,  
     623–5  
     intersecciones con poliedros,  
     625–6  
     método del búfer de luz, 628–30  
     muestreo adaptativo, 632  
     muestreo estocástico, 634–5  
     rayo de sombra, 622  
     rayo del ojo (rayo del píxel),  
     618–619  
     rayo del píxel (primario),  
     618–619  
     rayo luminoso invertido, 620  
     rayo reflejado, 620  
     rayo refractado, 620, 622  
     rayos distribuidos, 634–8  
     rayos secundarios, 620  
     recorrido de celdas, 627–8  
     subdivisión adaptativa, 627  
     subdivisión del espacio, 627–8  
     subdivisión uniforme, 627  
     supermuestreo, 632  
     trazado de conos, 633  
     vector de la trayectoria, 621–2  
     vector de transmisión, 622  
     volúmenes de contorno, 626  
 Trazadora de plumillas, 68, 69  
 Trazo descendente (carácter), 217  
 Tubo de rayos catódicos (TRC), 35  
     (véase también Monitores de  
     vídeo; Pantalla de barrido)  
     alta definición, 38  
     ánodo de aceleración, 36  
     cañón de electrones, 36  
     cátodo, 36

Tubo de rayos catódicos, (*cont.*)  
 color, 42–44  
 componentes, 36–38  
 deflexión del haz, 37  
 enfoque, 37  
 filamento, 36  
 fósforo, 36  
 intensidad del haz, 37  
 máscara de sombra, 42–43  
 máscara de sombra delta-delta, 43  
 máscara de sombra en línea, 43  
 penetración del haz, 42  
 persistencia, 38  
 rejilla de control, 37  
 relación de aspecto, 39  
 resolución, 38  
 RGB, 42–44  
 tamaño de punto, 38  
 tensión de aceleración, 36  
 velocidad de refresco, 36

**U**

Unidad imaginaria, 825  
 Unión, polilíneas  
 en bisel, 190  
 en punta, 190  
 redondeada, 190  
 Universales, coordenadas, 71–72  
 URL (Uniform Resource Locator), 69–70  
 Usuario:  
 diálogo, 724  
 facilidades de ayuda, 725–6  
 interfaz, 724 (*véase también*  
 Interfaz gráfica de usuario)  
 modelo, 724  
 uvn, sistema de coordenadas, 363–4

**V**

Valores cromáticos, 737, 741, 742  
 Variable:  
 dependiente, 828–9  
 independiente, 828–9  
 Variación de intensidad con la profundidad, 356–7  
 Vector:  
 ángulos directores, 815  
 arista, 129–30  
 columna (matriz), 821–2

cosenos directores, 815  
 de desplazamiento, 238 (*véase*  
*también* Traslación)  
 de eje (rotación), 275–6  
 de nudos, 456  
 de posición, 818–9  
 de transmisión (refracción), 599–  
 600  
 de trayectoria (trazado de rayos),  
 621–2  
 del observador (modelo de iluminación), 588  
 del punto medio, 590–2  
 dimensión, 818  
 en la representación de cuaternios,  
 827–8  
 espacio vectorial, 819–20  
 fila (matriz), 821–2  
 fuente luminosa, 586–7  
 módulo (longitud), 815  
 multiplicación por un escalar, 816  
 normal a la superficie, 138–9,  
 656, 658, 830–1  
 operaciones, 816–7  
 producto, 817–8  
 producto escalar, 816–7  
 producto vectorial, 817–8  
 propiedades, 814–6  
 proyección paralela oblicua,  
 376–7  
 punto medio, 590–2  
 reflexión especular, 588  
 suma, 816  
 transmisión (refracción), 599–600  
 traslación, 238  
 vertical (carácter), 218  
 visualización de datos, 532–5  
 Vector base, 819–20 (*véase también*  
*Base*)  
 ortogonal, 819–20  
 ortonormal, 819–20  
 Vector normal:  
 interpolación (representación  
 superficial de Phong), 617  
 plano de visualización, 362  
 promedio (malla poligonal), 614  
 superficie curva, 656, 658, 830–1  
 superficie de un plano, 138–9  
 Vector vertical de visualización:

bidimensional, 308  
 tridimensional, 363  
 Vectores de los ejes de coordenadas  
 (base), 819–20  
 Velocidad de la luz, 736  
 Ventana, 305 (*véase también*  
 Ventana de recorte; Ventana de visualización; Transformación de visualización; Visor)  
 Ventana de recorte, 305, 370, 377–8  
 efectos de ampliación, 306  
 efectos panorámicos, 306  
 en coordenadas de visualización,  
 308  
 en coordenadas universales,  
 308–9  
 mapeado de visor, 310–1  
 no lineal, 335–6, 346  
 no rectangular, 335, 345  
 Ventana de visualización, gestión,  
 305  
 Vértice (polígono), 128  
 Visiocasco, 11, 47–50, 360 (*véase*  
*también* Realidad virtual)  
 Visión (teoría de los tres estímulos),  
 742  
 Visor, 305 (*véase también* Ventana de recorte)  
 en coordenadas de pantalla, 311–3  
 normalizado, 310–1  
 tridimensional, 361 (*véase tam-*  
*bien* Volumen de visualización)  
 Vista de elevación, 368  
 Vista superior, 368  
 Vistas en sección, 359  
 Visualización:  
 archivo, 42  
 ampliación y reducción, 298–9  
 aplicaciones, 12–19  
 bidimensional, 305–14  
 científica, 12, 529 (*véase también*  
 Visualización de datos)  
 controlador, 50–51  
 coprocesador, 53  
 dispositivos, 35–58 (*véase tam-*  
*bien* Monitores de vídeo;  
 Procesadores de visualización)  
 despiece y secciones transversales, 359

estereoscópica, 47–50, 360  
 funciones de respuesta, 81  
 lista de, 42  
 métodos, 529–37 (*véase también*  
     Visualización de datos)  
 panorámica, 306  
 programa de, 42  
 tridimensional, 71–72, 355–60,  
     362–96  
 ventana, 32, 305  
**Visualización de datos:**  
     aplicaciones, 12–19  
     campos escalares, 529–33  
     campos multivariados, 536  
     campos tensoriales, 535–6  
     campos vectoriales, 532–5  
     escalares, 529–33  
     glifos, 536  
     gráficas de contorno, 530–2  
     isolíneas, 530–2  
     isosuperficies, 530–2  
     líneas de campo, 534  
     líneas de flujo, 534  
     métodos de codificación de color,  
         529  
     multivariados, 536  
     representación de volúmenes,  
         530–2, 532–4  
**Visualización de polígonos:**  
     bandas de mach, 617  
     Gouraud, 614–7  
     intensidad constante, 614  
     interpolación de intensidad, 614–7  
     interpolación de vectores norma-  
         les, 617  
     Phong, 617  
     Phong rápido, 617–8  
     plano, 614  
**Volumen de visualización paralelo-**  
 pipédico, 371, 377–8, 390, 393–4  
 rectangular, 371  
**Volumen de visualización,** 361  
 cubo unitario, 370–2, 378–9,  
     392–5  
 normalizado, 372–3, 379, 392–5  
 proyección en perspectiva,  
     383–92  
 proyección paralela, 370–2, 370–2  
**Vóxel,** 491

VRML (Virtual-reality modeling  
 language), 70

## W

Warn, modelo (fuente luminosa),  
 582–3  
 Weiler-Atherton, recorte de polígo-  
 nos, 343–5  
 WGL (Interfaz Windows a  
 OpenGL), 75  
 World Wide Web, 69–70

## X

XBM, 804  
 XPM, 804  
 XWindow System, 75  
 XYZ, modelo de color, 739–40

## Y

$YC_rC_b$ , modelo de color, 745  
 YIQ, modelo de color, 745  
 YUV, modelo de color, 745



# Índice de funciones OpenGL

## Funciones de la biblioteca básica

- glBegin:  
banda de cuadriláteros, 143–4  
banda de triángulos, 141–3  
cuadriláteros, 143  
polígono, 140–1  
polilínea, 91–92  
polilínea cerrada, 92  
puntos, 88–90  
segmentos de línea, 79, 91  
triángulo «ventilador», 142–3  
triángulos, 141–2
- glBindTexture, 676–7  
glBitmap, 148–50  
glBlendFunc, 185–6, 228–9
- glCallList, 156–7  
glCallLists, 157–8, 787  
glClear, 78–9, 569–70  
glClearColor, 76–7, 187–8, 315–6  
glClearDepth, 569–70  
glClearIndex, 187–8, 315–6  
glClipPlane, 409–10  
glColor, 78–9, 183–4  
glColorPointer, 186–7  
glCopyPixels, 152–3, 266–7  
glCopyTexImage, 675–6  
glCopyTexSubImage, 675–6  
CullFace, 568–9
- glDeleteLists, 158–9  
glDeleteTextures, 676–7  
glDepthFunc, 569–70  
glDepthMask, 570–1, 666–7  
glDepthRange, 569–70  
glDisable, 184–5, 196–7, 212–3,  
  409–10, 474–5, 568–9  
glDisableClientState, 147–8  
glDrawBuffer, 151–2, 266–7
- glDrawElements, 145–7  
glDrawPixels, 151, 266–7
- glEdgeFlag, 215–6  
glEdgeFlagPointer, 215–6
- glEnable:  
*antialiasing*, 228–9  
comprobación de profundidad,  
  569–70  
curvas de Bézier, 474–5  
eliminación de polígonos, 568–9  
estilos de línea, 196–7  
estilos de relleno, 212–3  
fuente luminosa, 659–60  
mezcla de color, 152–3, 184–5  
normalización de vectores, 667–8  
planos de recorte opcionales,  
  409–10  
semitonos (tramado), 668–9  
superficies de Bézier, 477–8  
texturado, 668–9, 671–2, 673–4  
variación de la intensidad con la  
  profundidad, 571
- glEnableClientState:  
indicadores de aristas, 215–6  
matrices de colores, 185–6  
matrices de coordenadas de textu-  
  ra, 675–6  
matrices de vértices, 145–7  
normales a la superficie, 667–8
- glEnd, 88–89  
glEndList, 155–6, 787
- glEvalCoord:  
curva de Bézier, 474–5  
superficie de Bézier, 477–8
- glEvalMesh:  
curva de Bézier, 477  
superficie de Bézier, 478–9
- glFlush, 80–1  
glFog, 571  
glFrontFace, 215–7  
glFrustum, 398–9
- glGenLists, 156–7  
glGenTextures, 677  
glGet\*\*, 229, 295–6, 314–5, 410,  
  772  
glGetTexLevelParameter, 679
- glIndex, 183–4  
glInterleavedArrays, 186–7  
glIndexPointer, 187–8  
glInitNames, 712–3  
glIsList, 156–7  
glIsTexture, 677–8  
glLight, 658–9  
glLightModel, 662–4  
glLineStipple, 196–7  
glLineWidth, 195–6  
glListBase, 157–8  
glLoadIdentity, 88, 293–4, 314  
glLoadMatrix, 293–4  
glLoadName, 712–3  
glLogicOp, 152–3
- glMap:  
  curva de Bézier, 474–5  
  superficie de Bézier, 477–8
- glMapGrid:  
  curva de Bézier, 477  
  superficie de Bézier, 478–9
- glMaterial, 664–5  
glMatrixMode, 78–9, 88, 294–5,  
  314, 396  
glMultMatrix, 294–5
- glNewList, 155–6, 787

glNormal, 666–7  
 glNormalPointer, 667–8  
 glOrtho, 397  
 glPixelStore, 149–50  
 glPixelZoom, 267–8  
 glPointSize, 195–6  
 glPolygonMode, 214–6, 570–1  
 glPolygonOffset, 214–5  
 glPolygonStipple, 212–3  
 glPopAttrib, 229–30  
 glPopMatrix, 295–6  
 glPopName, 712–3  
 glPushAttrib, 229–30

glPushMatrix, 295–6  
 glPushName, 712–3

glRasterPos, 149–50, 155  
 glReadBuffer, 151–2  
 glReadPixels, 151–2, 266–7  
 glRect, 139–40  
 glRenderMode, 711–2  
 glRotate, 293

glScale, 293  
 glSelectBuffer, 711–2  
 glShadeModel, 197–8, 213–4, 666–7

glTexCoord, 669–70, 672–9

glVertex, 79, 88–90  
 glVertexPointer, 145–7, 186–7  
 glViewport, 314–5, 399

## Funciones de la biblioteca GLU

gluBeginCurve, 480–1  
 gluBeginSurface, 481–2  
 gluBeginTrim, 483–4  
 gluBuild\*MipmapLevels, 678–9  
 gluBuild\*Mipmaps, 678–9

gluCylinder, 426–7

gluDeleteNurbsRenderer, 480–1  
 gluDeleteQuadric, 426–7  
 gluDisk, 426–7

gluEndCurve, 480–1  
 gluEndSurface, 481–2  
 gluEndTrim, 483–4

gluGetNurbsProperty, 482–3

gluLoadSamplingMatrices, 483–4  
 gluLookAt, 397

gluNewNurbsRenderer, 480–2  
 gluNewQuadric, 425–6  
 gluNurbsCallback, 483–4  
 gluNurbsCallbackData, 483–4  
 gluNurbsCurve, 480–1  
 gluNurbsProperty, 481–2, 482–3  
 gluNurbsSurface, 481–2

gluOrtho2D, 79, 88, 314–5  
 gluPartialDisk, 426–7

gluPerspective, 398–9  
 gluPickMatrix, 712–3  
 gluPwlCurve, 483–4

gluQuadricCallback, 427–8  
 gluQuadricDrawStyle, 425–6  
 gluQuadricNormals, 427–8  
 gluQuadricOrientation, 426–7  
 gluQuadricTexture, 679  
 gluSphere, 425–6

## Funciones de la biblioteca GLUT

glutAddMenuEntry, 717–8  
 glutAddSubMenu, 720–1  
 glutAttachMenu, 717–8

glutBitmapCharacter, 155  
 glutButtonBoxFunc, 711

glutCreateMenu, 717  
 glutCreateSubWindow, 318–9  
 glutCreateWindow, 76–7, 315–6

glutDestroyMenu, 720

glutDestroyWindow, 316–7  
 glutDetachMenu, 723–4  
 glutDeviceGet, 710  
 glutDialsFunc, 711  
 glutDisplayFunc, 76–7, 319–20

glutFullScreen, 316–7

glutGet, 320  
 glutGetMenu, 720–1  
 glutGetWindow, 316–7

glutHideWindow, 318–9

glutIconifyWindow, 317–8  
 glutIdleFunc, 319–20, 772

glutInit, 76–7, 314–5  
 glutInitDisplayMode, 76–7, 182–3, 315–6, 568–9, 772  
 glutInitWindowPosition, 76–7, 315–6  
 glutInitWindowSize, 76–7, 315–6

glutKeyboardFunc, 705

glutMainLoop, 76–7, 319–20

- glutMotionFunc, 705
- glutMouseFunc, 700–1
- glutPassiveMotionFunc, 705
- glutPopWindow, 317–8
- glutPositionWindow, 316–7
- glutPostRedisplay, 319–20
- glutPushWindow, 317–8
- glutRemoveMenuItem, 723–4
- glutReshapeFunc, 158–9, 317–8
- glutReshapeWindow, 316–7
- glutSetColor, 184–5
- glutSetCursor, 318–9
- glutSetIconTitle, 317–8
- glutSetMenu, 720
- glutSetColor, 184–5
- glutSetCursor, 318–9
- glutSetIconTitle, 317–8
- glutSetMenu, 720
- glutSetWindow, 316–7
- glutSetTitle, 317–8
- glutShowWindow, 318–9
- glutSolidCone, 424–5
- glutSolidCube, 417
- glutSolidDodecahedron, 417
- glutSolidIcosahedron, 417–9
- glutSolidOctahedron, 417
- glutSolidSphere, 424
- glutSolidTeapot, 425–6
- glutSolidTetrahedron, 417
- glutSolidTorus, 424–5
- glutSpaceballButtonFunc, 710
- glutSpaceballMotionFunc, 710
- glutSpaceballRotationFunc, 711
- glutSpecialFunc, 707
- glutStrokeCharacter, 155
- glutSwapBuffers, 666–7, 772
- glutTabletButtonFunc, 710
- glutTabletMotionFunc, 710
- glutWireCone, 424
- glutWireCube, 417
- glutWireDodecahedron, 417
- glutWireIcosahedron, 417
- glutWireOctahedron, 417
- glutWireSphere, 424
- glutWireTeapot, 425–6
- glutWireTetrahedron, 417
- glutWireTorus, 424–5





En esta tercera edición se presentan los principios básicos de diseño, utilización y comprensión de los sistemas y aplicaciones infográficos, junto con numerosos ejemplos de programación en OpenGL. Se analizan en profundidad los componentes tanto hardware como software de los sistemas gráficos, utilizándose un enfoque integrado para relacionar los temas de gráficos bidimensionales y tridimensionales. Sin presuponer ningún conocimiento previo del lector en el tema de gráficos por computadora, los autores presentan los conceptos fundamentales y los utilizan para mostrar cómo crear todo tipo de imágenes, desde simples dibujos lineales hasta escenas fotorrealistas altamente complejas.

### Novedades principales

- Proporciona explicaciones completas y exhaustivas de la biblioteca básica de programación gráfica OpenGL y de las bibliotecas auxiliares GLU y GLUT.
- Incluye un amplio conjunto de más de 100 ejemplos de programación para ilustrar el uso de las funciones OpenGL.
- Presenta ejemplos de programación en C11, con más de 20 programas C11 completos.
- Combina la explicación de los métodos infográficos tridimensionales y bidimensionales.
- Incluye los más recientes avances en las técnicas y aplicaciones infográficas.

### Acerca de los autores

**Donald Hearn** se incorporó a la Facultad de Informática de la Universidad de Illinois en Urbana-Champaign en 1985. El Dr. Hearn ha impartido un amplio rango de cursos sobre gráficos por computadora, visualización científica, ciencias de la computación, matemáticas y ciencia aplicada. Asimismo, ha dirigido numerosos proyectos de investigación y ha publicado diversos artículos técnicos en estas áreas.

**M. Pauline Baker** pertenece al Departamento de Informática y a la Escuela de Informática de la Universidad de Indiana-Universidad Purdue. La Dra. Baker es una eminente científica y dirige el Laboratorio de Tecnología Ubiña para Visualización y Espacios Interactivos, colaborando con diversos grupos de investigación en la utilización de la infografía y de la realidad virtual para la exploración de datos científicos. Anteriormente, la Dra. Baker era Directora asociada de entornos de visualización y entornos virtuales en NCSA (National Center for Supercomputer Applications), Universidad de Illinois.

### Incluye:



LibroSite es una página web asociada al libro, con una gran variedad de recursos y material adicional tanto para los profesores como para estudiantes. Apoyos a la docencia, ejercicios de autocontrol, enlaces relacionados, material de investigación, etc., hacen de LibroSite el complemento académico perfecto para este libro.

