maik_sys
www.wuolah.com/student/maik_sys

Resumen por temas y preguntas de exámenes IG 2018.pdf

Resumen por temas y preguntas de examen IG 2018

3° Informática Gráfica

Grado en Ingeniería Informática

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación
UGR - Universidad de Granada

Resumen por temas y preguntas de exámenes

Informática Grafica 2017/2018

TEMA 1

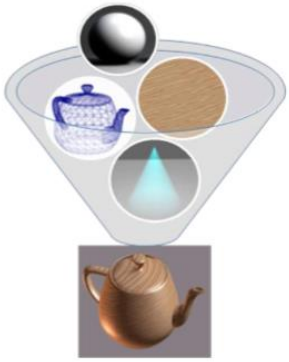
1.1. Proceso de visualización

En aplicaciones interactivas 3D hay tres elementos básicos: el usuario, el modelo digital 3D y la imagen sintética generada. Para generar la imagen sintética necesitamos:

- Objetos o personas en un entorno, con una forma y colores determinados.
- Una cámara para capturar la fotografía.
- Luz (sin luz se ve todo negro).

En informática grafica necesitamos:

- Modelos digitales 3D de lo que queremos representar:
  - Geometría (Triángulos).
  - Propiedades sobre apariencia (textura, color, material).
- Iluminación (posición de luces, modelo de sombreado, etc.)
- Una o varias cámaras, indicando para cada una
  - Angulo de visión
  - Posición y orientación
- Información sobre la resolución de la imagen de salida.



Diversos elementos necesarios para componer una imagen por ordenador.

Estos elementos forman los dos componentes principales: la escena y los parámetros de visualización.

1.2. Ray-Tracing: definición y efectos que puede simular

Primitiva: elementos más pequeños que pueden ser visualizados, en este caso, normalmente son triángulos.

Ray-Traicing: algoritmo trazado de rayos, lo que hace es seguir el camino que sigue un rayo de luz desde la imagen hasta la fuente de luz, pasando por la cámara y rebotando entre todos los objetos de la escena.

Suele ser más lento pero consigue resultados más realistas, se utiliza para la síntesis de imágenes realistas off-line, como la producción de animaciones y efectos especiales en películas.

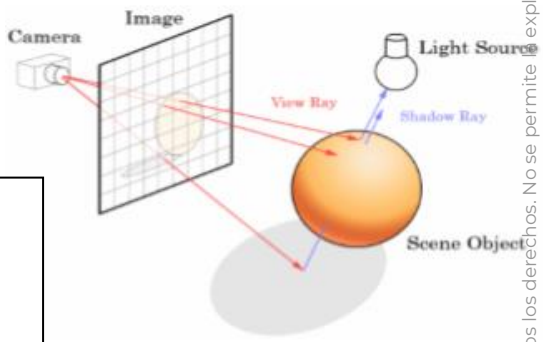


Ilustración 14: Trazado de rayos.

```
Inicializar el color de todos los pixeles
Para cada pixels de I:
    Calcular T el conjunto de primitivas que cubren s
    Para cada primitiva P de T
        Calcular el color de P en s
        Actualizar el color de s
```

1.3. Redibujado: Mecanismo de doble buffer

En OpenGL se utiliza un sistema de doble buffer para el dibujado, de forma que la rasterización se realiza en una imagen que no se ve, está oculta, hasta que no ejecutamos el intercambio de buffers. De esta forma no se ve cómo avanza el barrido del algoritmo raster y en escenas animadas esto es muy importante. Se puede utilizar un único buffer, pero no es recomendable.

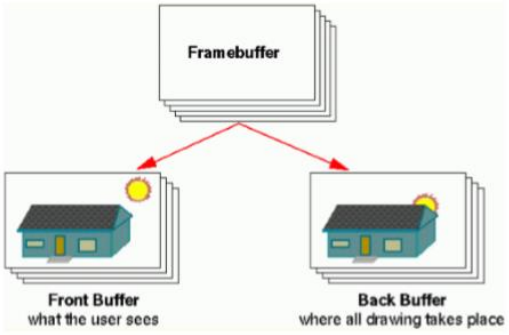
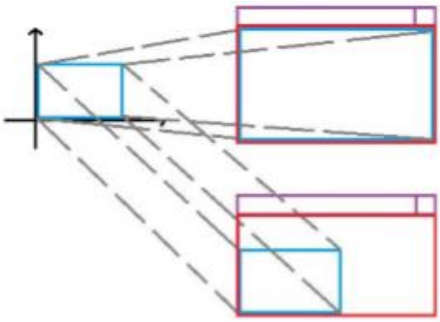


Ilustración 18 Mecanismo de Doble Buffer

1.4. ViewPort

La función glViewport permite establecer que parte de la ventana será usada para visualizar. Es un bloque rectangular de pixeles.



glViewport(izq, abajo, ancho, alto) ;

Cuando se realiza un cambio de tamaño de ventana, es bueno redimensionar el viewport para adaptarlo al nuevo tamaño de ventana, si queremos que siga conservando el 100% del área. Es lo que se llama una transformación de coordenadas de vista a coordenadas de dispositivo.

```
void FGE_CambioTamano(int nuevoAncho, int nuevoAlto) {
        glViewport(0,0,nuevoAncho,nuevoAlto);
}
```

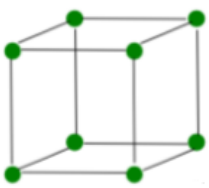
TEMA 2

2.1. Modelos de fronteras o superficie

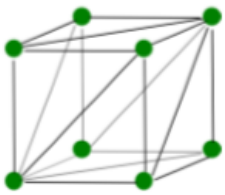
Para representar modelos de fronteras se utilizan polígonos, que vistos todos juntos pueden parecer como un mosaico o una red.

- Geometría: posiciones o coordenadas de puntos que están sobre la superficie
- Topología: organización de la geometría para dar lugar a una superficie.

Geometría



Una posible topología



Conservando Geometría

Cambiando topología

Esta estructura reticular de caras, aristas y vértices se denomina normalmente malla de triángulos, triangle mesh en inglés.

Estructuras de datos para almacenar solidos definidos por fronteras. (Pregunta de Examen)

- **Lista de triángulos aislados (sopa de triángulos)**: definir cada triangulo uno a uno, con las coordenadas de cada uno de sus vértices. Para cada triangulo necesitamos 9 valores float, es muy simple, pero tiene puntos débiles:
  - Los vértices pueden aparecer repetidos.
  - Falta de información cuando dos triángulos comparten aristas.
- **Tiras de triángulos:** permite ahorrar repetir vértices y reutilizarlos sin tener que indicarlos expresamente. No es fácil sacar de una sola tira toda una malla poligonal. Hay algoritmos que se encargan de descomponer las mallas de triángulos en tiras lo más largas posibles, así se consigue menor redundancia en la trasmisión de vértices. Con esta estructura conseguimos mejorar el rendimiento pero no tenemos información de la topología.
- **Tabla de vértices y triángulos:** hay que minimizar la redundancia en la información de los vértices y proporcionar cierto tipo de información tipológica, entonces se puede usar un vector de vértices y vector de triángulos. Tendremos 3*numero_de_vertices y 3*numero_de_triangulos.

```
struct Malla {
        Natural num_ver;
        Natural num_tri;
        Tupla3r *vertices;
        Tupla3n *triangulos;
};
```

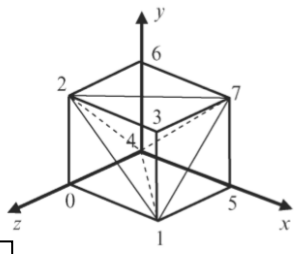


Tabla de vértices		
x	y	z
0.0	0.0	1.0
1.0	0.0	1.0
0.0	1.0	1.0
1.0	1.0	1.0
0.0	0.0	0.0
1.0	0.0	0.0
0.0	1.0	0.0
1.0	1.0	0.0

Tabla de triángulos		
v0	v1	v2
0	1	2
1	3	2
2	3	7
2	7	6
1	7	3
1	5	7
6	7	4
7	5	4
0	4	1
1	4	5
2	6	4
0	2	4

Visualización con glDrawElements

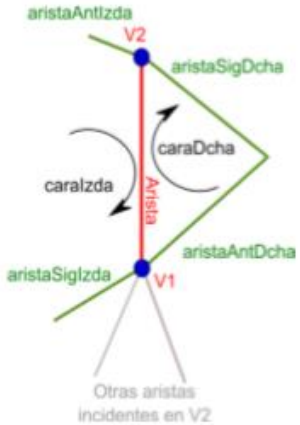
```
void visualizarVA( Malla *mesh ) {
        glEnableClientState( GL_VERTEX_ARRAY ); // habilitar 'vertex arrays'
    // especificar puntero a tabla de coords. de vértices
    glVertexPointer( 3, GL_FLOAT, 0, mesh->vertices );
    // dibujar usando vértices indexados
    glDrawElements( GL_TRIANGLES, 3*mesh->num_tri,
                    GL_UNSIGNED_INT, mesh->triangulos ) ;
} ;
```

- **Aristas Aladas:** cada arista almacena referencias a los vértices situados en los extremos de la arista, ya que podemos entender los triángulos como una ilusión óptica formada por la conectividad entre los vértices (red de aristas).

```
Vértice
Tupla3r p;
Natural arista;
```

```
Cara
Natural arista;
```

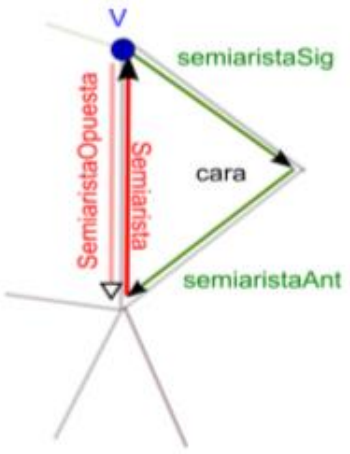
```
Arista
Natural V1, V2;
Natural caraIzda,
caraDcha;
Natural aristaSigDcha;
Natural aristaSigIzda;
Natural aristaAntDcha;
Natural aristaAntIzda;
```



⚠ Las aristas aladas son elegantes y potentes, pero hay que estar constantemente comprobando la orientación de la arista antes de decidir el paso a la siguiente arista. Las **Semiaristas Aladas** -> eliminan esta contra.

- **Semiaristas Aladas:** divide cada arista en dos semiaristas orientadas en sentido antihorario de la cara a la que pertenece cada una. Hoy día la mayoría de las librerías utilizan mallas de triángulos y conservan la información topológica, usan la estructura de datos de semiaristas aladas.

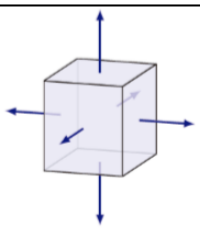
```
Vértice
    Tupla3r    p;
    Natural
    semiarista;

Cara
    Natural
    semiarista;

Semiarista
Natural V;
Natural cara;
Natural semiaristaSig;
Natural semiaristaAnt;
Natural opuesta;
```



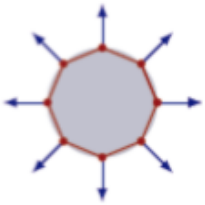
2.2. Atributos de los elementos de modelos de fronteras (Normales vértices y caras)

- Normales: vectores de longitud unidad
  - Normales caras: vector unitario perpendicular a cada cara, apuntando al exterior de la malla. Se pre-calcula a partir de la información de la cara.
  - Normales vértices: vector unitario perpendicular al plano tangente a la superficie en la posición del vértice.

Normales caras



Normales Vértices



- Colores: valores RGB o RGBA
  - Colores caras: útil cuando cada cara representa un trozo de superficie de color homogéneo.
  - Colores vértices: color de la superficie en cada vértice.

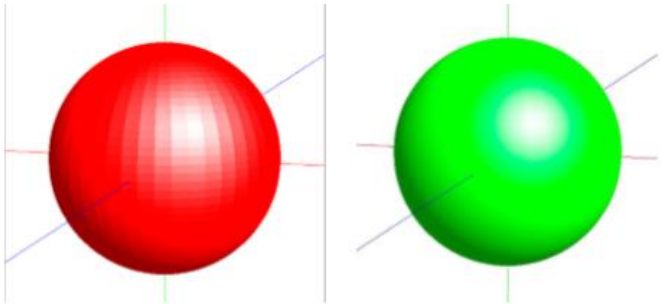


Ilustración 50: Iluminación usando normales de caras (izda) y normales por vértice (dcha.)

- Otro atributo habitual es la coordenada de textura.

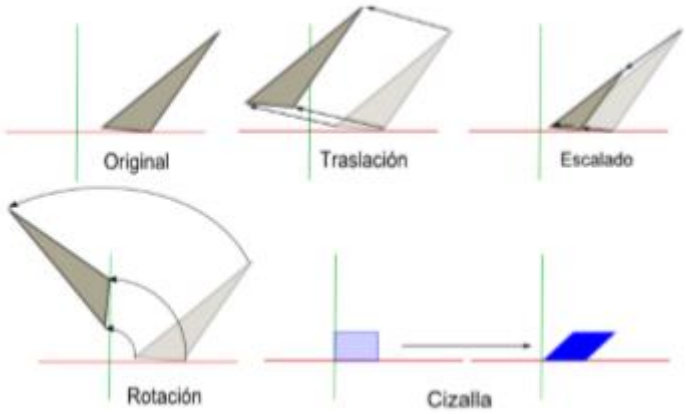
Interpolación de color

Se necesita usar la función **glShadeModel**:

- **glShadeModel(GL_FLAT):** dibuja cada triángulo con el color del último vértice de la cara.
- **glShadeModel(GL_SMOOTH):** hace que cada pixel sea generado por interpolación lineal de los colores de los vértices del triángulo.

2.3. Transformaciones usuales en informática gráfica

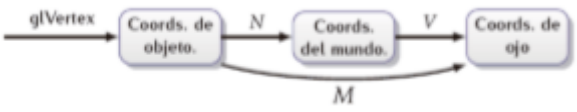
- **Traslación**: consiste en desplazar todos los puntos en la misma dirección y misma distancia.
- **Escalado**: estrechar o alargar las figuras en una o varias direcciones.
- **Rotación**: rotar los puntos de un ángulo dado, en torno a un eje de rotación.
- Cizalla: desplazamiento de todos los puntos en la misma dirección, pero con distintas distancias.



2.4. Transformaciones en OpenGL

OpenGL almacena una matriz 4x4M que codifica una transformación geométrica, que se llama modelview matriz (matriz de modelado y vista). Se puede ver como la composición de dos matrices, V y N:

- N: matriz de modelado, que posiciona los puntos en su lugar de coordenadas del mundo.
- V: matriz de vista, que posiciona los puntos en su lugar de coordenadas relativas a la cámara.



OpenGL va componiendo la matriz modelview con las instrucciones que se le indiquen:

```
glMatrixMode(GL_MODELVIEW); // indicamos que, a partir de aquí,
                            // se modifica la matriz modelview, M.
glLoadIdentity() ; / / hacemos M:= I (matriz identidad)
glMultMatrix( T3 ); / / hacemos M:=M·T3
glMultMatrix( T2 ); / / hacemos M:=M·T2
glMultMatrix( T1 ); / / hacemos M:=M·T1
```

Al final tendrá que ejecutar estas instrucciones:

$$M = T_3 \cdot T_2 \cdot T_1$$

Es decir, el efecto de T1 seguido de T2 seguido de T3.

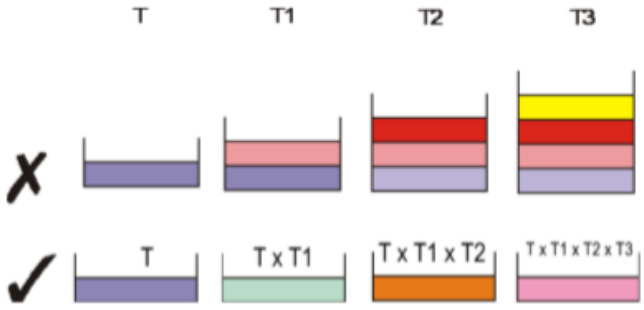
Orden inverso al que se escribe. El motivo es por la multiplicación de matrices, la última que metemos es la primera que se le aplica al gráfico.

2.5. Mecanismo de pilas para las transformaciones

OpenGL tiene un mecanismo de pilas para las transformaciones geométricas, que guarda en la pila el estado actual.

Cuando se realiza glPushMatrix, se duplica el tope de la pila, así se almacenan dos copias de la matriz actual. La que queda en el tope es la única que se consulta y se ve afectada por las operaciones de transformación, mientras que la que hay debajo solo será utilizadas para realizar glPopMatrix.

La pila de transformaciones cuando se realizan multiplicaciones de matrices no incrementa de tamaño.



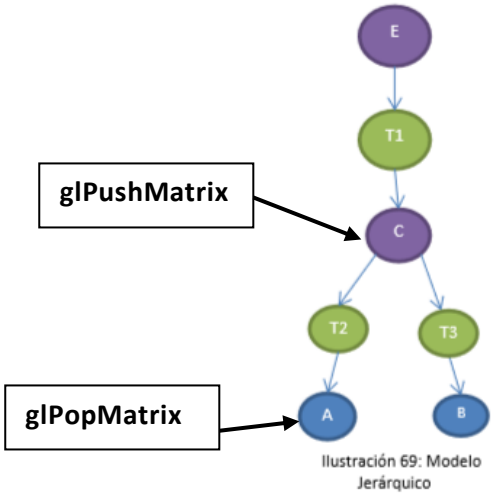


Ilustración 69: Modelo Jerárquico

TEMA 3

Transformación de vista 3D: Etapas para realizarla y los parámetros que intervienen. (Pregunta de Examen).

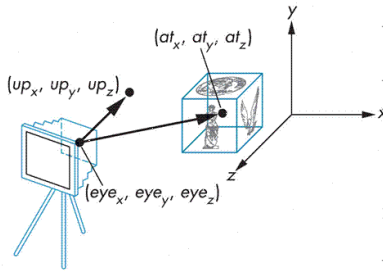
Las etapas son:

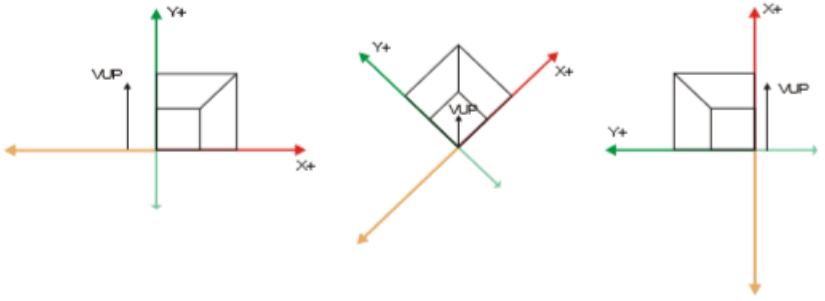
1. **Transformaciones del modelo**
   - ✓ Situarlo en la escena.
   - ✓ Cambiarlo de tamaño.
   - ✓ Crear modelos compuestos de otros más simples.
2. **Transformación de vista**
   - ✓ Poner al observador en la posición deseada.
3. **Transformación de perspectiva**
   - ✓ Pasar de un mundo 3D a una imagen 2D.
4. **Rasterización**
   - ✓ Calcular para cada pixel su color, teniendo en cuenta la primitiva que se muestra, su color, material, texturas, luces, etc…
5. **Transformación de dispositivo**
   - ✓ Adaptar la imagen 2D a la zona de dibujado.

- **La transformación de vista**: una cámara o un observador, tienen un espacio y una orientación. Esto hace que la escena se vea desde un punto de vista u otro y son definidos mediante las transformaciones de vista.

```
void gluLookAt(GLdouble eyeX,  GLdouble eyeY,  GLdouble eyeZ,
               GLdouble atX,  GLdouble atY,  GLdouble atZ,
               GLdouble upX,  GLdouble upY,  GLdouble upZ);
```

- Esta función posiciona al observador de forma que el ojo está en la posición eye, mirando hacia el punto at y con el sentido “hacia arriba” definido por el vector up, que normalmente es (0,1,0).





Visualización de una escena con UP (0,1,0), (1,1,0) y (1,0,0).

Fin de pregunta, lo que viene ahora es ampliación

- **VRP**: punto del espacio donde está el ojo, el observador ficticio que contempla la escena. Está en el plano de proyección.
- **VPN**: vector libre que indica la dirección en la que se está mirando.
- **VUP**: indica la dirección hacia arriba en la imagen.

¿De dónde salen estos ejes?

- **N** es el vector **VPN**, el eje Z de nuestro sistema de coordenadas de vista.
- **U** seria el eje **X** y es perpendicular al plano que forman **VUP** y **VPN**. Se calcula mediante el producto vectorial de los dos vectores.
- **V** seria el eje **Y** del sistema de coordenadas, es ortogonal a **N** y **U**. Se calcula mediante el producto vectorial de los dos ejes.

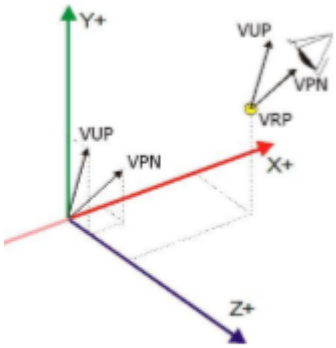


Ilustración 75 VRP, VPN y VUP

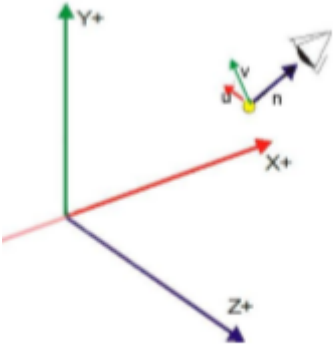
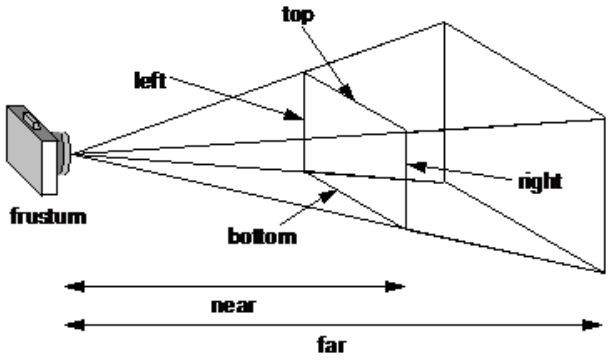


Ilustración 76: Sistema de Referencia de la Cámara

3.1. Volumen de visualización(Frustum)

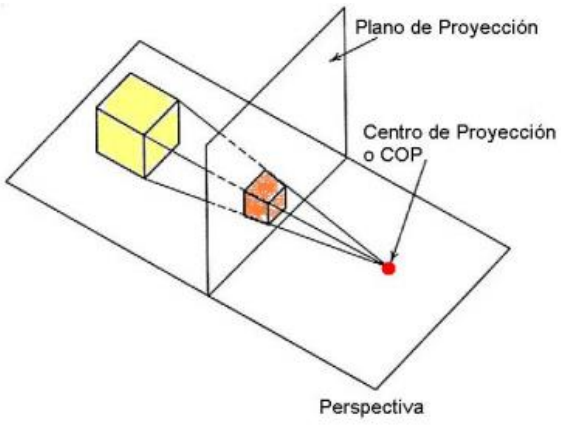
- Frustum es una figura geométrica de aspecto piramidal, que delimita la región del espacio del universo 3D que terminará siendo visible en la pantalla.



- Sus parámetros son:
  - **Near**: distancia del observador al plano más cercano. Todo lo que esté más cerca del plano se ignora.
  - **Far**: distancia del observador al plano más lejano. Lo que esté más alejado se ignora.
  - **Bottom**: distancia desde el centro del plano cercano hasta el borde inferior del frustum.
  - **Top**: distancia desde el centro del plano cercano hasta el borde superior del frustum.
  - **Right**: distancia desde el centro del plano cercano hasta el borde derecho del frustum en su parte más cercana a la cámara.
  - **Left**: distancia desde el centro del plano más cercano hasta el borde izquierdo del frustum en su parte más cercana a la cámara.

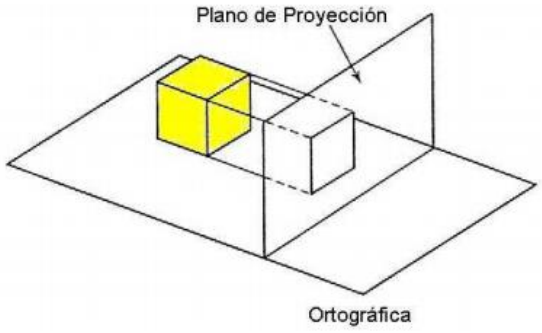
3.2. Transformación de proyección

- **Perspectiva**: se genera un único centro de proyección en un punto concreto y finito del eje z de la cámara.
    - Sus características son:
        - El tamaño del objeto varía necesariamente con la distancia del objeto a la proyección.
        - Objetos parecen más realistas.
        - No es útil para almacenar formas y medidas exactas de los objetos.



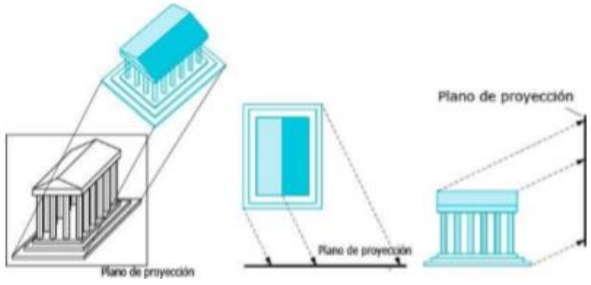
Perspectiva

- **Ortográfica**: proyección paralela, su centro de proyección está ubicado en el infinito del eje de Z de la cámara, de forma que todas las proyecciones son paralelas.



Ortográfica

- **Oblicua**: proyección paralela, se consigue situando el proyector en el infinito pero fuera del eje Z de la cámara.



3.3. Definición de la matriz de proyección en OpenGL (glFrustum, glOrtho, glPerspective)

En la máquina de estados de OpenGL hay una matriz de proyección que puede ser manipulada mediante varias llamadas.

Primero hay que invocar: **glMatrixMode(GL_PROJECTION);** esto indica a OpenGL que todas las operaciones sobre matrices que se apliquen desde ese instante hasta que se cambie el glMatrixMode, se aplicarán sobre la matriz de proyección. Tras esta llamada, inicializamos al valor identidad: **glLoadIdentity();**

- Si queremos usar en nuestra escena, una proyección **perspectiva**, usamos:

```
void glFrustum(GLdouble  left,  GLdouble  right,
               GLdouble  bottom,  GLdouble  top,
               GLdouble  near,  GLdouble  far);
```

  Los valores near y far son siempre positivos, los demás están en el sistema de coordenadas de la cámara (normalmente left y botton son negativos).

- Si queremos usar en nuestra escena, una proyección **ortográfica**, usamos:

```
void glOrtho(GLdouble  left,  GLdouble  right,
             GLdouble  bottom,  GLdouble  top,
             GLdouble  near,  GLdouble  far);
```

- Si queremos usar en nuestra escena, una proyección **perspectiva** con otros parámetros, usamos:

```
void gluPerspective(GLdouble  fovY,
                    GLdouble  aspect,
                    GLdouble  near,  GLdouble  far)
```

  La diferencia respecto a glFrustum es que esta proyección está centrada obligatoriamente con respecto al observador, ya que no hay posibilidad de definir las posiciones de los planos de volumen de visualización.

3.4. Eliminación de partes ocultas (Z-BUFFER)

Definición: para cada primitiva que se dibuje, se almacena en el pixel del Z-Buffer, su valor de profundidad normalizado. Si el valor de Z de la primitiva es menor al actual en esa posición, se sustituye el color del pixel por ese nuevo valor y se actualiza el ZBuffer.

Definición dos: buffer de profundidad, es la parte de la memoria de nuestra tarjeta gráfica encargada de la visibilidad de nuestros gráficos 3D según las coordenadas de sus pixeles. Se puede gestionar qué elementos de una escena renderizada son visibles y cuales permanecen ocultos según sus posiciones en el eje Z (Distancia de la cámara).

Características:

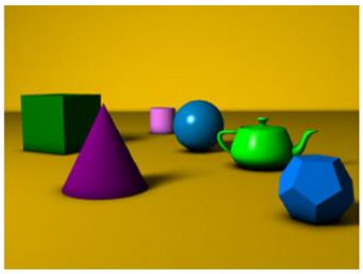
- Es uno de los más simples y más usados.
- Fácilmente implementarlo a nivel de hardware.
- En OpenGL:
  glutInitDisplaymode(GLIT_DOUBLE | GLUT_RGBA | GLUT_DEPTH**);**
  glEnable(GL_DEPTH_TEST **);**
- Almacena la profundidad del objeto más cercano al punto (x, y) de la pantalla.

- Cada vez que se dibuja, al igual que limpiamos el buffer de color, hay que limpiar el ZBuffer:

  glClear(GL_DEPTH_BUFFER **);**

  La principal ventaja de este algoritmo es que en el peor de los casos su complejidad es proporcional al número de primitivas.

Escena y su Z-Buffer



Esquema del Algoritmo Z-Buffer

Procedimiento **Z-Buffer**(lista polígonos L)

```
Inicializar ZBuffer e Imagen

Paracada polígono P en L
   Sea Δz := −nx/nz
   Paracada línea de barrido ocupada por P
            (línea a altura y, entre x0 y x1)
         x := menor entero mayor o igual que x0
         z := f(x,y)(=profundidad en Z de P en (x,y) )
         Mientras que x < x1
              Si z > ZBuffer[x,y] entonces
                   Imagen[x,y] := color de P
                   Zbuffer[x,y] := z
              z := z+Δz
              x := x+1
```

3.5. Iluminación: Reflexión, fuentes de luz y Normales

Un modelo de iluminación simplificado

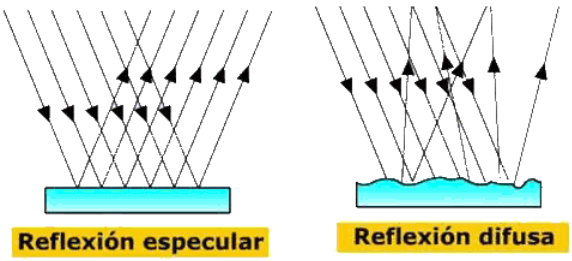
- Las fuentes de luz son puntuales o unidireccionales, hay un número finito de ellas (8 luces).
- No se considera luz incidente en una superficie que no provenga directamente de ellas fuentes de luz.
- Los objetos o polígonos son totalmente opacos.
- El objeto no impide la trayectoria de la luz.
- El espacio entre los objetos no dispersa la luz.
- Se usa el modelo RGB.

La consecuencia de la simplificación se ve en la siguiente imagen:

Iluminación compleja (izda.) vs Iluminación

Tipos de Reflexión:

- **Especular:** Si la luz incide sobre una superficie perfectamente lisa, los rayos que llegan paralelos salen también paralelos después de reflejarse.
- **Difusa:** En superficies rugosas, los rayos incidentes paralelos producen rayos reflejados que no son paralelos entre sí, debido a que la inclinación de la superficie varía de un punto de incidencia a otro.



Fuentes de luz:

- **Posicionales**: ocupan un punto en el espacio y emiten en todas las direcciones.
- **Direccionales**: están en un punto a distancia infinita y tienen un vector director en el que emiten los fotones.

Ilustración 97 Fuente de luz posicional vs direccional

Normales:

La iluminación en un punto p depende la orientación de la superficie en dicho punto. Esta orientación se caracteriza por el vector normal NP asociado a dicho punto. En los modelos de fronteras, se puede calcular de varias formas:

- Considerar que NP es la normal del triángulo que contiene p.
- Aproximar el valor de NP en cada vértice como la media de las normales de los triángulos que lo comparten.

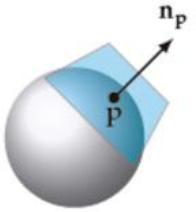
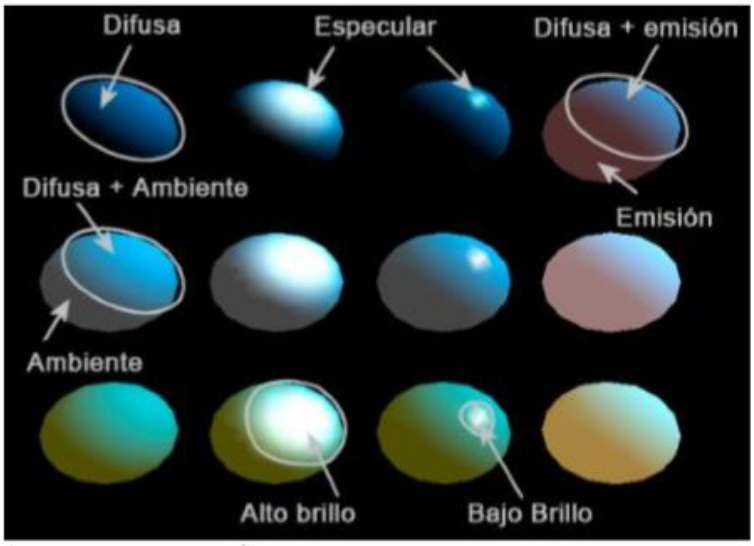


Ilustración 98: Vector normal en un punto p

3.6. Modelo de reflectancia de Phong

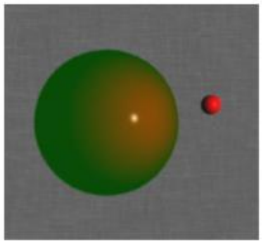
Este modelo propone que un material se puede definir mediante cuatro componentes:

- Color **difuso**, indica el color base del material.
- Color **especular**, indica el color de los brillos.
- Color **ambiente**, comportamiento del objeto cuando no le incide directamente ninguna fuente de luz.
- Color de **emisión**, comportamiento "artificial", que hace que se vea el objeto incluso cuando no hay fuentes de luz activas. El objeto no ilumina a otros ni se convierte en una fuente de luz, pero si se vería a oscuras.
- **Brillo**, cantidad de luz que es rebotada de forma especular.



El color del pixel depende también de los valores de color de las fuentes de luz que intervienen, por ejemplo:

- Un material "blanco" iluminado por una fuente de luz roja se verá rojo.
- Un material "blanco" iluminado por una fuente de luz verde, se verá verde
- Un material "verde" iluminado por una fuente de luz roja se verá naranja (Imagen derecha).

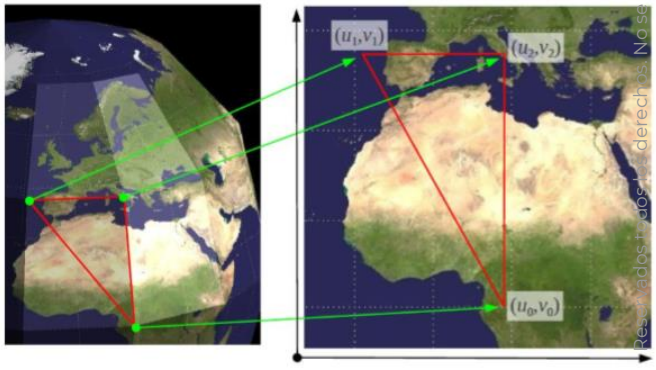


Material ambiental verde iluminado con luz difusa roja

3.7. Texturas

Coordenadas de textura

Para poder aplicar una textura a la superficie de un objeto, es necesario hacer corresponder cada punto p = (x, y, z) de su superficie con un punto sp=(u, v) del dominio de la textura: Debe existir una función f tal que (u, v) = f (x, y, z). Entonces decimos que (u, v) son las coordenadas de textura del punto p.



La función f puede implementarse usando una tabla de coordenadas de textura de los vértices, o bien calcularse proceduralmente con un subprograma.

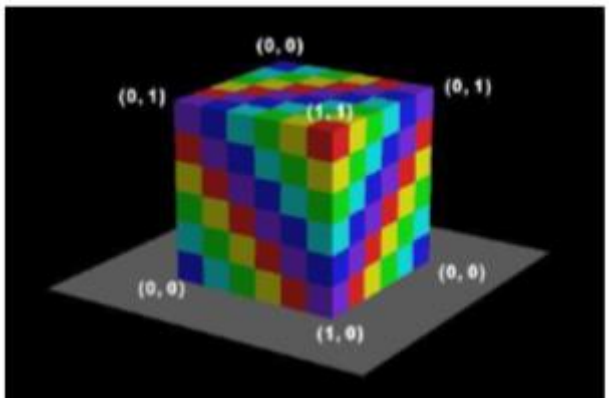
i es el índice del vértice en la tabla de vértices.

La asignación de coordenadas de textura se puede hacer de dos formas:

- **Asignación explícita a vértices**: La coordenadas forman parte de la definición del modelo de escena, y son un dato de entrada al cauce gráfico, en forma de un vector o tabla de coordenadas de textura de vértices (v0, u0), (v1, u1), . . . (un-1, vn-1). Se puede hacer:
    - manualmente en objetos sencillos, o bien
    - de forma asistida usando software para CAD (p.ej. 3D Studio).

Esta modalidad hace necesario realizar una interpolación de coordenadas de textura en el interior de los polígonos de la malla.

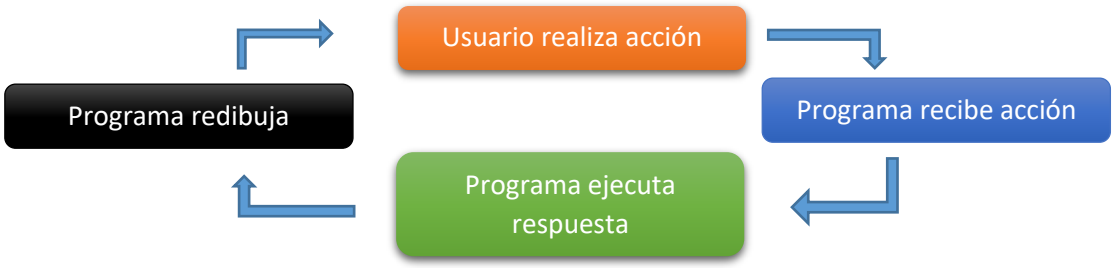
Asignación manual de texturas



- **Asignación procedural:** La función f se implementa como un algoritmo CoordText(p) que se invoca para calcular las coordenadas de textura, de forma que acepta un punto p como parámetro y devuelve el par (u, v)= f(p) con las coordenadas de textura de p. Esta asignación procedural se puede hacer de dos formas:
    - **Asignación procedural a vértices**, de forma que coordText(vi) es invocado para calcular las coordenadas de textura de cada vértice, y las coordenadas obtenidas se almacenan y después se interpolan linealmente en el interior de los polígonos de la malla. Es decir, usamos una función para realizar la asignación por vértices, pero una vez creado el vector de coordenadas de textura, se comporta igual que en el caso de la asignación explícita a vértices.
    - **Asignación procedural a puntos**, en cuyo caso coordText(p) es invocado cada vez que hay que calcular el color de un punto p de la superficie durante los cálculos del modelo de iluminación local. Esto sólo se puede hacer programándolo en el fragment shader.

TEMA 4

4.1. *Proceso de interacción.*



A este proceso es conveniente proporcionarle un mecanismo de realimentación, que permita conocer los pasos seguidos por el sistema. Un sistema sin realimentación es muy poco usable. Algunas técnicas de realimentación:

- Mostrar el estado del sistema
- Indicar la herramienta activa
- Resaltar elementos seleccionables
- Cambios de cursor según acción permitida

Hay dos tipos de dispositivos lógicos principales:

- **Locator:** lee las posiciones (x, y) de la pantalla. Su sistema de coordenadas es el de la pantalla, por lo que hay que realizar una transformación inversa a la visualización para obtener la coordenada del mundo a la que se corresponde.
- **Pick:** lee identificadores de componentes del modelo geométrico. Es un dispositivo lógico que consulta la escena y devuelve el objeto sobre el que está el cursor.

4.2. *Gestión de eventos (Callbacks en GLUT)*

GLUT gestiona entradas en modo evento, tiene un buffer donde se va almacenando las peticiones realizadas por el usuario (mover ratón, pulsar tecla, etc...). El gestor de eventos va leyendo ese buffer, y si el evento tiene asociada una función que lo procesa (un callback), ejecuta ducha función y reacciona al evento recibido. Si no hay un callback para el evento, se ignora.

Callback más usados en glut son:

- **glutDisplayFunc**: Redibujado.
- **glutMouseFunc**: Pulsación de botones del ratón.
- **glutMotionFunc**: Movimiento del ratón mientras se pulsa un botón.
- **glutPassiveMotionFunc**: Movimiento del ratón sin pulsar botones.
- **glutReshapeFunc**: Cambio de tamaño de la ventana.
- **glutKeyFunc**: Pulsación de tecla.
- **glutIdleFunc**: Ausencia de eventos externos.
- **glutTimerFunc**: Temporizador.

Hay que tener una **máquina de estados**, que reaccione de forma distinta si se pulsa botón derecho o izquierdo, que haga unos cálculos si se está moviendo el ratón con el botón principal o secundario, o ninguno.

4.3. Selección en OpenGL

Para distinguir objetos, OpenGL utiliza una pila de enteros como identificadores. Dos primitivas se consideran que corresponden a objetos distintos cuando el contenido de la pila de nombre es distinto.

```
glLoadName(i) / / Sustituye el nombre activo por i
glPushName(i) / / Apila el nombre i
glPopName() / / Desapila un nombre
glInitNames() / / Vacia la pila de nombres
```

Cuando hacemos una selección y no queremos dibujar, solo queremos saber el objeto que hay debajo del cursor, OpenGL funciona en tres modos:

- **RENDER**: modo por omisión. Este modo las primitivas que se envían se dibujan en pantalla.
- **SELECT**: OpenGL devuelve los identificadores de las primitivas que se proyectan en una región.
- **FEEDBACK:** OpenGL devuelve la información geométrica de las primitivas que se dibujarían.

El cambio de modo se realiza con la función glRenderMode:

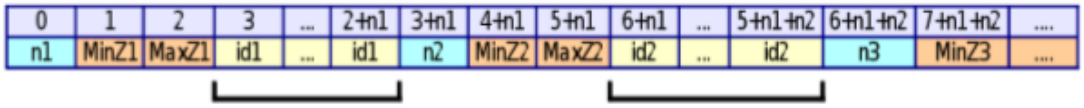
```
glRenderMode( GL_SELECT );
glRenderMode( GL_RENDER );
glRenderMode( GL_FEEDBACK );
```

Buffer de Selección

Al realizar la selección usando el modo GL_SELECT devuelve el número de objetos seleccionados como resultado de selección y la información asociada a estos en un buffer de enteros.

El buffer devuelve:

- El número de primitivas que contiene la pila
- El intervalo de profundidades de cada primitiva: Zmin y Zmax.
- Los nombres asociados a cada primitiva. En un objeto jerárquico, se obtendría el nombre del objeto y de la parte o partes que hayamos querido identificar.

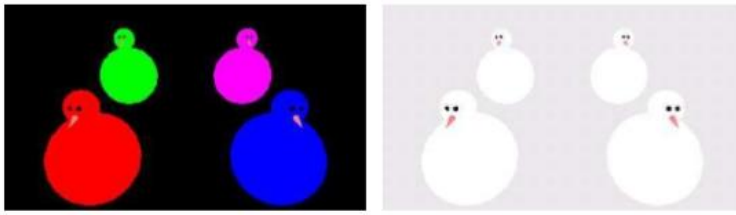


Selección en OpenGL con codificación por colores.

Se trata de crear una función de dibujado distinta para cuando queremos seleccionar, y cuando el usuario hace clic, se pinta la escena "para seleccionar" en el buffer trasero y se lee el color del pixel donde el usuario ha hecho click.

Resumen:

- Llamar a función dibuja_seleccion()
- Leer el pixel (x, y) dado por la función gestora del evento de ratón.
- Averiguar a qué objeto hemos asignado el color de dicho pixel.
- **No intercambiar buffers.**



Escena visible (dcha) y escena para selección (izda).

4.4. Animación

Animación asistida por ordenador y animación por ordenador

- **Animación asistida por ordenador:** el ordenador se usa como una herramienta para generar imágenes intermedias, pero los verdaderos artistas y los que crean a los personajes a mano son los humanos.
- **Animación por ordenador:** la totalidad del proceso la genera el ordenador.

Nos centramos en la animación por ordenador. La gran diferencia frente a la animación clásica y a la asistida por ordenador, es que una vez se han definido los modelos y datos que controlan la animación, es el propio ordenador el que gestiona la obtención de imágenes, sin intervención humana en el proceso.

4.5. Animación por esqueletos

- **Cinemática directa:** Mediante este proceso, se definen en cada paso los ángulos concretos de rotación para cada articulación.
- **Cinemática inversa**: En este sistema se definiría la posición inicial y final del extremo de la jerarquía, en este caso la mano, y el sistema se encarga de calcular los ángulos de cada articulación en cada frame. Esta forma de animar implica establecer las restricciones de movimiento allí donde las hubiera (p.ej. que el codo no pueda ir hacia atrás), pero sin duda es la preferida por los diseñadores gráficos.

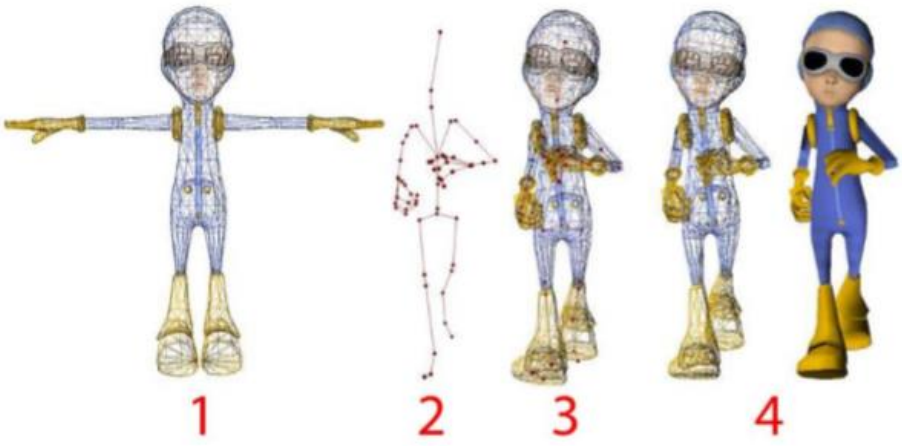


Ilustración 131 1. Malla a animar. 2. Esqueleto de control. 3. Distintas poses. 4. Resultado final.

4.6. Sistemas de partículas

Un sistema de partículas es un conjunto de muchas diminutas partículas que juntas representan un objeto difuso. A lo largo de un intervalo de tiempo, las partículas se generan en el sistema, se transforman y mueren.

Una partícula, en singular, es un elemento con una serie de propiedades comunes a todas las partículas de su clase, pero con distintos valores para cada una de la partícula. Entre estas propiedades podemos encontrar la posición, velocidad, aceleración, color, edad, etc. En función de lo que estemos modelando, necesitaremos controlar unas propiedades u otras en el sistema de partículas.

El ciclo de vida de una partícula:

- **Generación**: Las partículas se generan aleatoriamente en el emisor, para el cual se ha definido una forma y posición determinada que puede variar. Los valores iniciales son definidos de forma aleatoria o siguiendo un patrón.
- **Dinámica de partículas**: Los atributos o propiedades de las partículas cambian en cada frame (p.ej. en una explosión el color de la partícula se oscurece conforme se aleja del origen de la explosión, indicando que se enfría).
- **Extinción**: Cada partícula tiene dos propiedades que controlan su existencia: edad y tiempo máximo de vida. La edad es el número de frames que lleva viva la partícula, y siempre se inicializa a cero. Cuando la partícula llega a una edad igual a su tiempo de vida, ésta muere.

Ilustración 133 Sistema de partículas: fuego

Una partícula es algo diferente:

- No se representa con un conjunto de primitivas (polígonos), pero el conjunto de partículas sí forma un volumen.
- No es una entidad estática.
- Un sistema de partículas se dice que es no determinista, ya que su forma no está completamente definida y se usan procesos estocásticos que cambian la forma y apariencia.