

Informática Gráfica

Juan Carlos Torres

Curso 2024/25

Dpto. Lenguajes y Sistemas Informáticos
Universidad de Granada

Disclaimer

You can edit this page to suit your needs. For instance, here we have a no copyright statement, a colophon and some other information. This page is based on the corresponding page of Ken Arroyo Ohori's thesis, with minimal changes.

CC BY-NC-SA

 This book is released into the public domain using the CC BY-NC-SA. This license enables reusers to distribute, remix, adapt, and build upon the material in any medium or format for noncommercial purposes only, and only so long as attribution is given to the creator. If you remix, adapt, or build upon the material, you must license the modified material under identical terms.

To view a copy of the CC BY-NC-SA code, visit:

<https://creativecommons.org/licenses/by-nc-sa/4.0/>

Colophon

This document was typeset with the help of KOMA-Script and L^AT_EX using the kaobook class.

1

Introducción

La humanidad siempre ha considerado excitante fantasear con la existencia de mundos virtuales y con la posibilidad de que nuestra existencia no sea real. Lewis Carroll, en "A través del espejo" plantea en 1871 la posibilidad de existencia de realidades alternativas, cuando Alicia accede a un mundo virtual cruzando un espejo (figura 1.1).

Mucho antes, en 1635, Calderón de la Barca escribió la obra de teatro "La vida es sueño" en la que plantea la dificultad para saber si lo que percibimos es real o no¹.

Recientemente, el desarrollo de la Informática Gráfica ha abierto las puertas para que la generación de entornos virtuales sea una realidad, planteando un sin fin de cuestiones éticas al posible uso de las aplicaciones potenciales, mucho antes de que estas puedan ser realidad.

Concretamente, en la década de los 90, el desarrollo de nuevas tecnologías parecía augurar un cambio vertiginoso. Como ejemplo la mítica película "Tron" postulaba la existencia de un mundo virtual en el interior de los ordenadores al que las personas podría viajar. Algo más cerca de la realidad, la película Nivel 13 plantea un futuro en el que los sistemas de entretenimiento sean sistemas de Realidad Virtual en los que el usuario pueda crear mundos virtuales para vivir situaciones imposibles en el mundo real (figura 1.2), llegando a plantearse la posibilidad de que los avatares de ese mundo virtual lleguen a tener conciencia, y que puedan materializarse en el mundo real. Sobre esta misma idea se ha especulado con frecuencia, como ejemplos podemos citar la serie de dibujos animados Código Lyoko, que seguramente será más conocida por los más jóvenes (figura 1.3), o el problema de los tres cuerpos.

No obstante, y a pesar de que la tecnología ha avanzado mucho, no se puede decir que la realidad haya superado a la ficción. La realidad está muy lejos de ofrecer tan siquiera el grado de inmersión que se plantea en la película «Nivel 13».

En esta asignatura se aborda el problema de la creación y utilización de entornos virtuales. Es decir, el proceso de crear un escenario, su visualización y la interacción con el mismo. Es decir:

- Que abstracciones, modelos y estructuras de datos usar para representar escenas en un sistema informático.
- Que métodos y técnicas utilizar para interactuar con el escenario.
- Como generar imágenes del mismo.

En la asignatura abordaremos estos tres problemas.

1.1. Informática Gráfica

La Informática Gráfica es una disciplina que se centra en la creación, manipulación y visualización de imágenes y entornos virtuales mediante el uso de tecnología informática. Este campo abarca una amplia gama

| | | |
|-------|---|----|
| 1.1 | Informática Gráfica | 2 |
| 1.2 | OpenGL | 3 |
| 1.2.1 | Creación de la ventana de dibujo con glut | 4 |
| 1.2.2 | Ejemplo de programa | 5 |
| 1.2.3 | Transformaciones geométricas | 8 |
| 1.3 | Eliminación de partes ocultas: ZBuffer | 8 |
| 1.4 | Ejercicios | 10 |

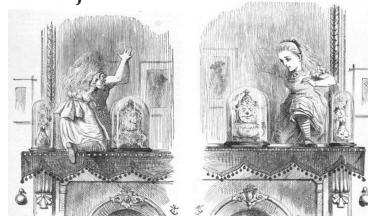


Figura 1.1: Lewis Carroll. Alicia en el país de las maravillas: A través del espejo. Madrid: Ediciones Cátedra, 1992.

1: Pedro Calderón de la Barca. La vida es sueño. (Primera edición 1635). Vicens Vives, 2014.

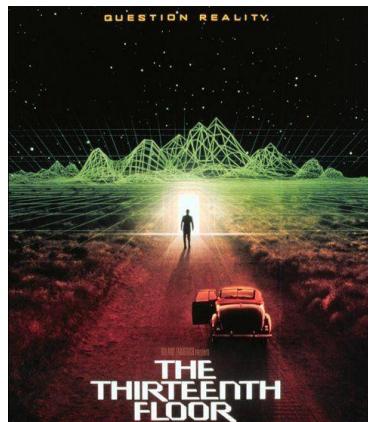


Figura 1.2: Cartel de la película Nivel 13.



Figura 1.3: Cartel de la serie Código Lyoko.

de áreas, desde el diseño de videojuegos y películas de animación hasta la visualización de datos científicos y médicos. A modo de ejemplo citaremos algunos:

Cine. Se utiliza para la creación de efectos visuales en películas.

Videojuegos. Es la herramienta esencial en el desarrollo de videojuegos.

Animación por ordenador. Al igual que en videojuegos, escenarios y personajes se representan como modelos gráficos.

Medicina. Los resultados de pruebas de diagnóstico por imagen (TAC, Resonancia, etc.) se pueden representar, visualizar y explorar interactivamente como modelos 3D. Estos modelos se pueden usar también para la simulación de procedimientos quirúrgicos.

Arquitectura. Se utiliza para la creación de modelos y visualizaciones de edificios, espacios urbanos.

Diseño industrial. Se utiliza para la creación de modelos de los elementos a diseñar. Estos modelos se suelen usar para realizar simulaciones acortando el proceso de prueba y fabricación.

En las aplicaciones gráficas podemos encontrar tres componentes fundamentales: la visualización, el modelado y la interacción.

El modelado se ocupa de la representación de objetos. Incluye los métodos de representación, las estructuras de datos y las operaciones que se realizan con las representaciones. Hay diferentes métodos de modelado, en función de la dimensión del objeto representado y de las operaciones que sea necesario realizar con él.

La visualización se ocupa de la generación de imágenes de los modelos. Dependiendo de la naturaleza del modelo la visualización puede implicar un amplio rango de acabados, desde la creación de gráficos 2D a la renderización de imágenes fotorealistas en sistemas de Realidad Virtual. La visualización juega un papel crucial en la comunicación de información compleja de manera efectiva.

La interacción es el tercer pilar de la Informática Gráfica y se ocupa de las técnicas que permiten que los usuarios interactúan con los modelos y cómo estos responden a sus acciones. Desde la navegación por entornos en 3D hasta la manipulación de objetos virtuales, la interacción juega un papel crucial en la creación de experiencias inmersivas.

Aunque la mayor parte de las aplicaciones tienen estos tres componentes, encontrándose en la zona rayada del esquema de la Figura 1.7, existen aplicaciones que no tienen alguno de estos componentes. Por ejemplo que calcule propiedades de un modelo previamente creado puede no necesitar ni visualización ni interacción.

1.2. OpenGL

Las prácticas de la asignatura se centrarán en la programación con OpenGL y glut, pero veremos otras herramientas para realizar operaciones específicas, concretamente: Unity 3D, Blender, X3Dom y MeshLab.

OpenGL es una API abierta y estándar que oculta el hardware y hace que la aplicación sea portable (Figura 1.8).

Permite manejar:



Figura 1.4: Visualización del modelo 3D de la cabeza de uno de los leones del Patio de los Leones en la Alhambra.

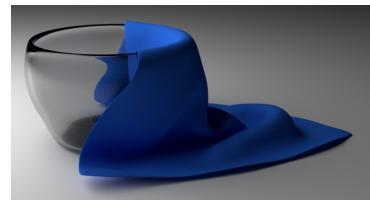


Figura 1.5: Fotograma de la simulación de la caída de una tela realizada con Blender.



Figura 1.6: Fotograma del juego Red Dead Redemption.

- Elementos geométricos
- Propiedades visuales
- Transformaciones
- Especificación de fuentes de luz Hardware
- Especificación de cámara

Existen muchas implementaciones de OpenGL. La estructura de procesamiento en una implementación típica es un cauce. La aplicación envía órdenes a OpenGL haciendo llamadas a la librería. Las órdenes pueden modificar el estado de OpenGL (p.e. asignar el color de dibujo) o indicar que se dibuje un elemento (p.e. un triángulo).

Cuando OpenGL recibe una orden de dibujo transforma el elemento geométrico, le calcula la iluminación y lo rasteriza, utilizando los parámetros almacenados en el estado (Figura 1.9). Estas operaciones no se realizan necesariamente en este orden, con frecuencia el cálculo de iluminación se realiza después de la rasterización.

OpenGL puede estar implementado por software o hardware. En el primer caso las funciones de OpenGL se ejecutan en CPU, en el segundo caso la mayor parte de las operaciones de OpenGL se ejecutan en la GPU (Unidad Gráfica de Procesamiento). La ejecución en GPU acelera enormemente el dibujo ya que las GPU son procesadores paralelos (tipo SIMD). En su estructura más simple la GPU tiene dos niveles de procesadores: procesadores de vértices y de fragmentos trabajando en un cauce.

Las primeras GPUs implementaban un cauce de fijo, con funcionalidad fija. Las GPUs modernas (desde 2002) son programables, permitiendo programar tanto los procesadores de vértices como los de fragmentos (a estos programas se les llama shaders). OpenGL permite programar shaders usando OpenGL Shading Language.

A partir de la versión 3 hay dos modos de funcionamiento: el modo de compatibilidad y el **core profile**. El **core profile** es más reciente y más potente, pero también más complejo ([KSS16]). En el modo clásico OpenGL incluye una programación por defecto de la GPU, por lo que se puede programar sin necesidad de crear los shaders ([Shr+07]). Comenzaremos trabajando con la versión clásica de OpenGL, conocida también como **compatibility mode**.

OpenGL permite dibujar objetos en 2D y en 3D. En este curso trabajaremos, salvo que se indique lo contrario en tres dimensiones. El sistema de coordenadas que utiliza OpenGL tiene el eje Y en vertical y el eje Z hacia el observador (ver Figura 1.10).

1.2.1. Creación de la ventana de dibujo con glut

Para poder utilizar OpenGL el programa debe crear al menos una ventana de dibujo. Esta operación (que pueden depender bastante del sistema operativo y de sistema de gestión de ventanas) no la realiza OpenGL. Para crear y gestionar las ventanas se debe usar una librería específica como QT, Glut o FLTK.

La alternativa más simple es usar glut, que es un toolkit sencillo diseñado específicamente para OpenGL [Kil96]. Glut incluye funciones para

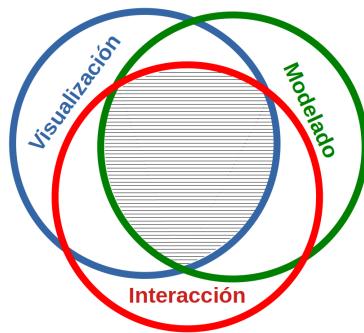


Figura 1.7: Componentes de un sistema gráfico.

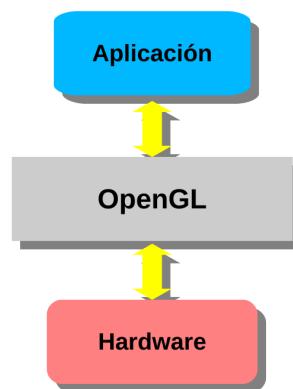


Figura 1.8: OpenGL es una API para la comunicación con el hardware gráfico

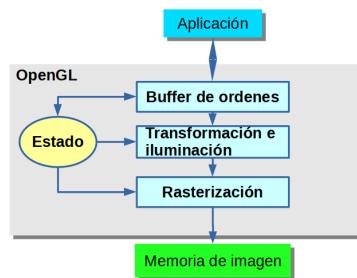


Figura 1.9: Esquema de funcionamiento de OpenGL

interaccionar con el sistema de gestión de ventanas y algunas funciones auxiliares de OpenGL (como la creación de elementos geométricos).

Glut es portable e independiente de la plataforma. Permite crear y gestionar ventanas gráficas y procesar eventos de entrada. Además incluye funciones para dibujar objetos poliédricos simples.

1.2.2. Ejemplo de programa

El siguiente es un ejemplo simple que crea un cubo que gira sobre un plano. El código de este programa (*cubo.c*) se puede descargar de Prado. La figura 1.11 muestra una captura de la aplicación.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 #include <GL/glut.h>
5
6 float roty=30.0;
7
8 void plano( float t ){//Construye un cuadrado horizontal
9     glBegin( GL_QUADS );
10    glNormal3f( 0.0, 1.0, 0.0 );
11    glVertex3f( t, 0, t );
12    glVertex3f( t, 0, -t );
13    glVertex3f( -t, 0, -t );
14    glVertex3f( -t, 0, t );
15    glEnd();
16 }
17
18 void Dibuja( ){
19     float pos[4] = {5.0, 5.0, 10.0, 0.0 };
20     float morado[4]={0.8,0,1,1}, verde[4]={0,1,0,1};
21     glClearColor(1,1,1,1); // Fondo blanco
22     glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
23     glLoadIdentity();
24     glTranslatef(-0.5,-0.5,-100);
25     glLightfv( GL_LIGHT0, GL_POSITION, pos );
26     glRotatef( 20, 1.0, 0.0, 0.0 );
27     glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, verde );
28     plano(30);
29     glRotatef( roty, 0.0, 1.0, 0.0 );
30     glTranslatef(0,5,0);
31     glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, morado );
32     glutSolidCube(10);
33     glutSwapBuffers();
34 }
35
36 void Ventana(GLsizei ancho,GLsizei alto) {
37     float D=ancho; if(D<alto) D=alto;
38     glViewport(0,0,ancho,alto); //fija el area de dibujo
39     glMatrixMode(GL_PROJECTION);
40     glLoadIdentity();
41     glFrustum(-ancho/D,ancho/D,-alto/D,alto/D,5,250);
42     glMatrixMode(GL_MODELVIEW);
43 }
44

```

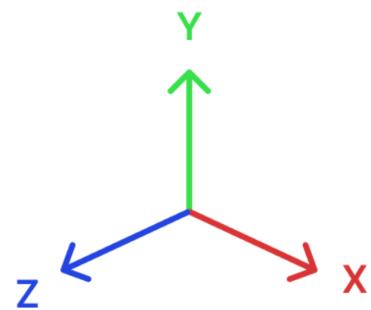


Figura 1.10: Sistema de coordenadas OpenGL

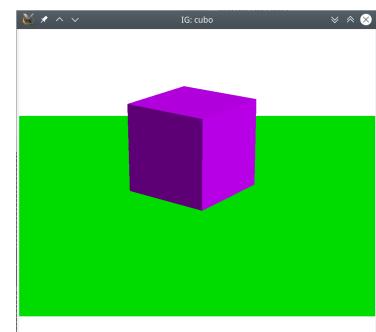


Figura 1.11: Captura de la ejecución del programa cubo

```

45 void idle(){
46     roty +=0.15;
47     glutPostRedisplay();
48 }
49
50 int main( int argc, char *argv[] )
{
51
52     glutInit( &argc, argv );
53     glutInitDisplayMode( GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH );
54     glutCreateWindow("IG: cubo");
55     glutDisplayFunc( Dibuja );
56     glutReshapeFunc( Ventana );
57     glutIdleFunc(idle);
58     glEnable( GL_LIGHTING );
59     glEnable( GL_LIGHT0 );
60     glEnable(GL_DEPTH_TEST);
61     glutMainLoop();
62     return 0;
63 }
```

La línea 4 incluye la cabecera de la librería glut, no es necesario incluir OpenGL ya que glut.h la incluye.

La línea 6 declara e inicializa la variable que se va a usar para indicar la rotación que se va a aplicar al cubo, expresado en grados.

La función **plano** (entre las líneas 8 y 10) crea un cuadrado de tamaño $t \times t$, centrado en el origen de coordenadas en el plano $y=0$. Para crear el cuadrado se le indica a OpenGL que se le van a pasar cuadriláteros (QUADS). La información de las primitivas se pasa entre una llamada a glBegin y una llamada a glEnd. En este bloque se especifica la normal (vector perpendicular de módulo). En este caso, como el plano es horizontal la normal es el vector $(0,1,0)$. A continuación se indican los vértices del cuadrilátero, que se deben dar en sentido antihorario cuando se ve el cuadrilátero por la cara delantera (Figura 1.12).

La función **Dibuja** (líneas 18 a 35) dibuja la escena. Esta función es llamada por glut cada vez que es necesario redibujar la escena. La variable **pos** indica la posición de la fuente de luz (fíjate que está dada como un vector de cuatro componentes, mas adelante explicaremos el significado de la cuarta coordenada). En este caso, la luz está colocada en la coordenada $(5,5,10)$ de la escena. Las variables morado y verde definen dos colores. Están expresadas también como vectores de cuatro componentes, que indican los valores de rojo, verde, azul y opacidad. Las componentes se dan normalizadas entre 0 y 1, indicando el 1 el valor máximo.²

Cada vez que se llama a **Dibuja** se visualiza un nuevo frame. Para evitar que el dibujo se realice mezclando con la imagen del frame anterior es necesario borrar el framebuffer (la memoria de imagen). En la línea 21 **glClearColor** indica el color con el que queremos que se borre (en este caso blanco), y en la linea 22 **glClear** borra el framebuffer. Observa que tiene como parámetro una operación or de dos constantes, cada una indica un valor a borrar. En este caso se está borrando la información de color y la de profundidad. Esta última se usa para evitar que elementos que están mas lejos de la cámara tapen a objetos que están mas cerca.

Para colocar los objetos en la escena se les aplican transformaciones geométricas, en este ejemplo se usan traslaciones (líneas 24 y 30) y rotaciones

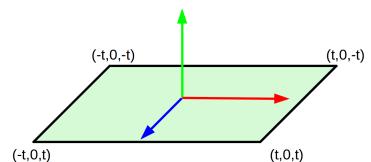


Figura 1.12: Esquema del cuadrilátero dibujado por la función **plano**.

2: Para que se puedan dibujar objetos transparentes no basta con usar valores de opacidad menores que 1.

(líneas 26 y 29). Cuando se indica una transformación geométrica se guarda en el estado de OpenGL y se aplica a todos los elementos que se dibujen a partir de ese momento. Para evitar que las transformaciones de un frame afecten a los frame sucesivos inicializamos la transformación con la identidad antes de empezar a dibujar (línea 23). La traslación 24 empuja toda la escena 100 unidades hacia la parte negativa de z y -0,5 en X e Y.³.

La línea 25 indica la posición en la que está la luz (cada luz tiene un identificador de GL_LIGHTn, en este caso estamos colocando la luz 0).

En la 26 se indica una rotación de 20 grados respecto al eje X (el eje se indica dando un vector, en este caso el (1,0,0), que es el eje X).

La línea 27 indica que el color de la cara delantera de lo que se dibuje a continuación debe ser verde. Este color se aplicará solo al plano, ya que en la línea 31 se cambia el color a morado, antes de dibujar el cubo. El cubo se dibuja con una función de glut **glutSolidCube** (línea 32), que dibuja un cubo del tamaño indicado centrado en el origen. Antes de dibujar el cubo también se aplica una rotación de **rotY** grados respecto al eje Y, que servirá para hacer que el cubo rote y una traslación de 5 unidades en Y, para colocarlo sobre el plano. La última línea de **Dibuja** llama a la función **glutSwapBuffers**, que hace que se muestre el frame que se acaba de dibujar.

La siguiente función **Ventana** (líneas 36 a 43) indica en que parte de la ventana debe dibujarse y como se proyecta el modelo en la pantalla. En la línea 38, **glViewport** especifica el área de la ventana que se va a usar para dibujar, en este caso es toda la ventana. La función **glFrustum** indica la transformación de perspectiva que se va a usar, esto es equivalente a indicar los parámetros de la cámara en una fotografía. Veremos estas funciones mas adelante.

La función **idle**, en las líneas 45 a 48 se ejecuta como función de fondo (siempre que no hay otra función que procesar). Aumenta el ángulo de rotación (línea 46) e indica que es necesario generar un nuevo frame llamando a **glutPostRedisplay** (línea 47).

El programa principal (líneas 50 a 63) crea la ventana de dibujo usando las funciones de glut, declara las funciones que deben responder a los distintos evento y establece algunos parámetros de OpenGL. Para crear la ventana de dibujo inicializa glut (línea 52); establece los parámetros de la ventana con **glutInitDisplayMode** y crea la ventana llamando a **glutCreateWindow**. Esta filosofía de separación de asignación de parámetros de la función que realiza la acción la encontramos con frecuencia en funciones de OpenGL y glut, también hemos visto ya el paso de parámetros haciendo un or. En este caso los parámetros que se pasan para la creación de la ventana son:

GLUT_RGBA Indica que se use formato **RGB α** para los colores.

GLUT_DOUBLE Usa dos copias de la memoria de imagen. Se dibuja en la que está oculta y se intercambian con la llamada a **glutSwapBuffers**. Esto evita el parpadeo (**flickering**).

GLUT_DEPTH Usa un buffer de profundidad para almacenar la distancia a la cámara a la que está el objeto que es visible en cada pixel. Es necesario para hacer eliminación de partes ocultas usando Z-Buffer.

3: La cámara está colocada en el origen de coordenadas mirando a la parte negativa de Z, de este modo la escena se coloca en el campo visual de la cámara

Las tres instrucciones siguientes (`glutDisplayFunc`, `glutReshapeFunc` y `glutIdleFunc`) indican las funciones de nuestro programa que se deben ejecutar cuando: sea necesario redibujar la escena, se cambie el tamaño de la ventana o cuando no haya ninguna acción que procesar. Los argumentos que reciben son las funciones **Dibuja**, **Ventana** e **idle**.

Las siguientes instrucciones son llamadas a la función `glEnable` que activa opciones de OpenGL. Concretamente:

GL_LIGHTING Activa el cálculo de iluminación.

GL_LIGHT0 Enciende la luz cero (**LIGHT0**).

GL_DEPTH_TEST Activa la eliminación de partes ocultas usando el algoritmo Z-Buffer.

Para compilar el programa se puede usar la siguiente orden:

```
1| gcc cubo.c -lglut -lGLU -lGL -o cubo
```

1.2.3. Transformaciones geométricas

Las transformaciones geométricas son funciones que modifican las coordenadas de los puntos. Existen varias transformaciones geométricas simples que son muy útiles en Informática Gráfica para la definición de escenas y animaciones:

- **Traslación:** desplazar todos los puntos del espacio de igual forma, es decir, en la misma dirección y la misma distancia.
- **Escalado:** estrechar o alargar las figuras en una o varias direcciones.
- **Rotación:** rotar los puntos un ángulo en torno a un eje.

La transformación de **traslación** T_d en \mathbb{E}_3 es una transformación geométrica que desplaza cualquier punto $\mathbf{p} = (x, y, z)$ sumándole un vector⁴ $\mathbf{d} = (d_x, d_y, d_z)$, es decir:

$$T_d(\mathbf{p}) \equiv \mathbf{p} + \mathbf{d} = (x, y, z) + (d_x, d_y, d_z) = (x + d_x, y + d_y, z + d_z)$$

La Figura 1.13 muestra el ejemplo de la aplicación de traslaciones a un objeto simple. El vector de traslación está representado en rojo.

Las transformaciones de **escalado** multiplican cada componente por un factor de escala. Una transformación de **escalado** E_s con factor de escala $\mathbf{s} = (s_x, s_y, s_z)$ transforma el punto $\mathbf{p} = (x, y, z)$ según:

$$E_s(\mathbf{p}) \equiv (s_x x, s_y y, s_z z)$$

La Figura 1.14 muestra el ejemplo de la aplicación de un escalado al objeto de la Figura 1.13.

1.3. Eliminación de partes ocultas: ZBuffer

Cuando se visualiza un modelo 3D es normal que se proyecten distintas superficies en la misma zona de pantalla. Para que la visualización sea entendible y se corresponda con la imagen real es necesario que solo se dibujen los elementos que están mas cerca de la cámara (que deben tapar a los que quedan detrás)⁵. A este proceso se le denomina **eliminación**

4: Usamos negrita para denotar vectores

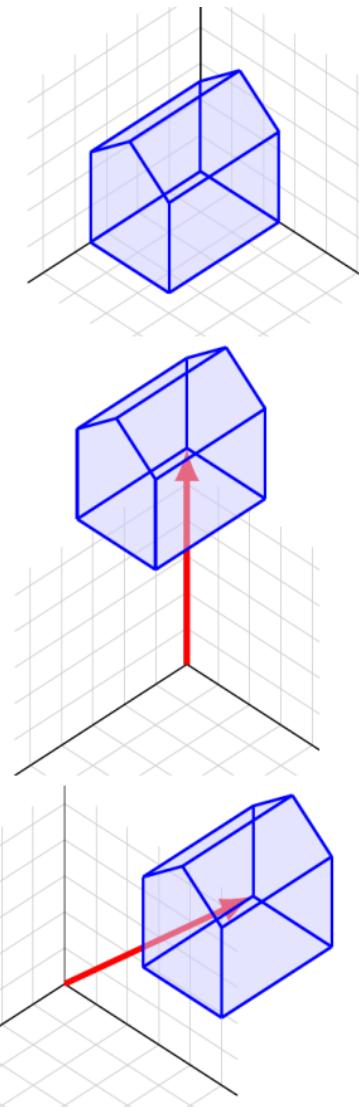


Figura 1.13: Ejemplos de traslaciones. El objeto de la figura superior se ha trasladado con $\mathbf{d} = (0, 6, 0)$ en la central y con $\mathbf{d} = (6, 0, 0)$ en la inferior.

de partes ocultas. Las figuras 1.15 y 1.16 corresponden a la misma escena, visualizada con y sin eliminación de partes ocultas.

La eliminación de partes ocultas suele realizarse con el algoritmo **Z-Buffer**, que realiza el proceso a nivel de pixel. Para ello se utiliza un **buffer de profundidad**, un array 2D de las mismas dimensiones del **frame buffer** en el que para cada pixel se almacena la distancia a la cámara a la que se encuentra el objeto dibujado en el pixel. Esto permite realizar el proceso de un modo muy simple: Para cada pixel que se va a dibujar se calcula su profundidad respecto a la cámara y solo se dibuja si no se ha dibujado previamente un objeto mas próximo. Antes de empezar a dibujar el buffer de profundidad se inicializa a la distancia máxima. El algoritmo es el siguiente:

```

1 color canvas[resx,resy] // Memoria de imagen
2 int zbuffer[resx,resy]=Zfar // buffer de profundidad
3 for each primitiva
4   Rasterizar(primitiva) // se descompone en un conjunto de
   fragmentos
5   for each fragment
6     if zbuffer[x,y] > fragment.depth
7       zbuffer[x,y] = fragment.depth
8       canvas[x,y] = fragment.color

```

En este proceso la función Rasterizar descompone la primitiva en un conjunto de pixeles (fragmentos). Cada uno de ellos tiene asociado el color y la profundidad (depth). Las primitivas son los elementos geométricos que se dibujan y los fragmentos se corresponden con pixeles.

La figura 1.17 muestra un esquema del proceso, la escena está en la parte central de la figura, la imagen generada a la derecha y el buffer de profundidad a la izquierda.

Las profundidades se transforman a un intervalo:

$$z' = \frac{Far + Near}{Far - Near} - \frac{2}{Z} \frac{Far \ Near}{Far - Near}$$

Siendo **Z** la profundidad del fragmento, **Far** la distancia máxima y **Near** la mínima. Esta expresión genera profundidades normalizadas entre -1 y 1, que se discretizan, para almacenarlas como enteros sin signo.

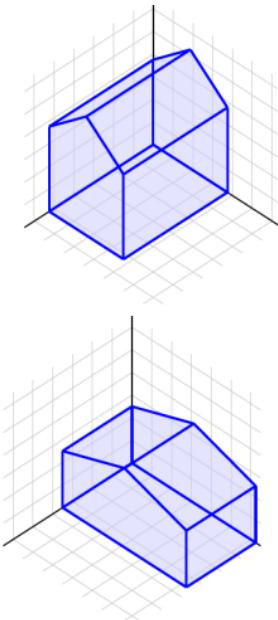


Figura 1.14: Ejemplos de escalados. El objeto de la figura superior se ha escalado con $s=(1.5,1.5,1.5)$ en la inferior con $s=(2.5,1,1)$ en la inferior.

5: Salvo que las superficies sean transparentes

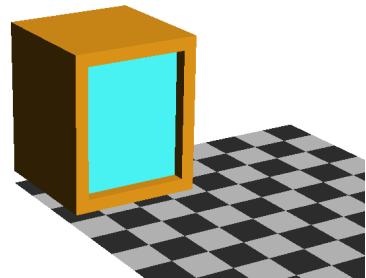


Figura 1.15: Escena visualizada con eliminación de partes ocultas

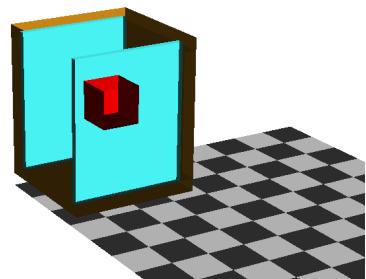


Figura 1.16: Escena visualizada sin eliminación de partes ocultas

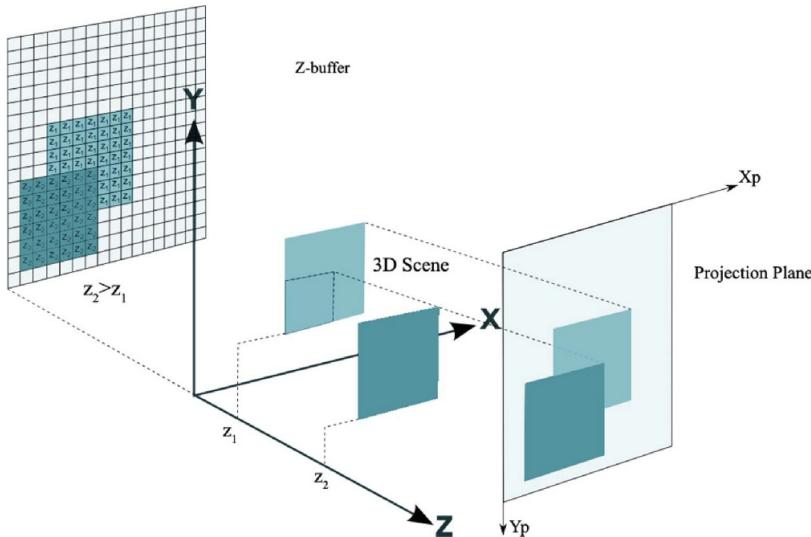


Figura 1.17: Esquema del funcionamiento del Z-Buffer
<https://docs.hektorprofe.net/graficos-3d/24-profundidad-con-z-buffer/>

1.4. Ejercicios

1. Quita el parámetro **GLUT_DOUBLE** de la inicialización de la venta en el programa **cubo** y comprueba y explica el resultado.
2. Comprueba y explica el resultado de quita el parámetro **GLUT_DEPTH**.
3. Comprueba y explica el resultado de comentar la activación de la luz cero.
4. Cual será el resultado de cambiar el valor del incremento de **roty** en la línea 46.
5. ¿Que pasó si el valor del incremento de **roty** es negativo?
6. Modifica el programa para que hacer que el plano se vea marrón.
7. Añade una esfera azul a la derecha del cubo.
8. Piensa ejemplos de aplicaciones que utilice Gráficos pero no tengan los tres componentes: visualización, interacción o modelado.
9. La figura 1.18 muestra dos paralelepípedos, ¿Cual puede ser la razón de que se vea rallada la parte central?
10. ¿Que parámetros se podrían cambiar para reducir este efecto?
11. Los dos últimos parámetros de la función **glFrustum** son las distancias a los planos de recortado delantero y trasero (los elementos que estén mas cerca que el plano delantero o mas lejos que el trasero no se dibuja). ¿Por que se introducen estos parámetros?

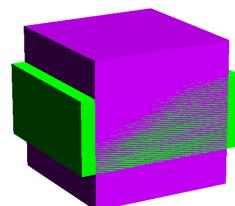


Figura 1.18: Imagen con dos paralelepípedos

Bibliografía

- [Bot+10] Mario Botsch et al. *Polygon mesh processing*. CRC press, 2010.
- [Cig+08] Paolo Cignoni et al. «MeshLab: an Open-Source Mesh Processing Tool». En: *Eurographics Italian Chapter Conference*. Ed. por Vittorio Scarano, Rosario De Chiara y Ugo Erra. The Eurographics Association, 2008. ISBN: 978-3-905673-68-5. doi: [10.2312/LocalChapterEvents/ItalChap/ItalianChapConf2008/129-136](https://doi.org/10.2312/LocalChapterEvents/ItalChap/ItalianChapConf2008/129-136).
- [Kil96] Mark J Kilgard. *The OpenGL utility toolkit (GLUT) programming interface API version 3*. [accedido 26-Abril-2024]. 1996. URL: <https://www.opengl.org/resources/libraries/glut/glut-3.spec.pdf>.
- [KSS16] John Kessenich, Graham Sellers y Dave Shreiner. *OpenGL Programming Guide: The official guide to learning OpenGL, version 4.5 with SPIR-V*. [accedido 26-Abril-2024]. Addison-Wesley Professional, 2016. URL: <https://www.cs.utexas.edu/users/fussell/courses/cs354/handouts/Addison.Wesley.OpenGL.Programming.Guide.8th.Edition.Mar.2013.ISBN.0321773039.pdf>.
- [Shr+07] Dave Shreiner et al. *OpenGL (R) programming guide: The official guide to learning OpenGL (R), version 2.1*. [accedido 26-Abril-2024]. Addison-Wesley Professional, 2007. URL: <https://theswissbay.ch/pdf/Gentoomen%20Library/Game%20Development/Programming/The%20Official%20Guide%20to%20Learning%20OpenGL%20Version%202.1.pdf>.

Informática Gráfica

Juan Carlos Torres

Curso 2024/25

Dpto. Lenguajes y Sistemas Informáticos
Universidad de Granada

Disclaimer

You can edit this page to suit your needs. For instance, here we have a no copyright statement, a colophon and some other information. This page is based on the corresponding page of Ken Arroyo Ohori's thesis, with minimal changes.

CC BY-NC-SA

 This book is released into the public domain using the CC BY-NC-SA. This license enables reusers to distribute, remix, adapt, and build upon the material in any medium or format for noncommercial purposes only, and only so long as attribution is given to the creator. If you remix, adapt, or build upon the material, you must license the modified material under identical terms.

To view a copy of the CC BY-NC-SA code, visit:

<https://creativecommons.org/licenses/by-nc-sa/4.0/>

Colophon

This document was typeset with the help of KOMA-Script and L^AT_EX using the kaobook class.

2

Modelos geométricos

Un modelo en Informática Gráfica es la representación computacional de una escena u objeto. El modelo describe tanto la forma como la apariencia del objeto. La forma del objeto es esencialmente información geométrica. La apariencia del objeto describe como se comporta ante la luz que recibe. En este tema nos centraremos en las propiedades geométricas. Hablamos de modelos geométricos cuando nos centramos en la representación geométrica de los objetos.

Hay diversos métodos de representación de modelos geométricos: basados en la representación de la superficie exterior del objeto; descomposición del espacio ocupado por el objeto en celdas y enumeran las celdas que ocupa (ver Figura 2.1); ecuaciones implícitas; en el proceso de construcción del objeto a partir de primitivas simples con operaciones booleanas (ver Figura 2.2); o en ecuaciones paramétricas 2.3, por citar algunas.

En este capítulo nos centraremos en la representación de la superficie exterior del objeto mediante un conjunto de polígonos (es decir poliedros). Veremos como podemos es posible con esta representación visualizar superficies curvas.

2.1. Representaciones poligonales

Una superficies poligonal o malla es una representación (discreta) de superficie. Las mallas son una forma común de representar objetos tridimensionales en informática gráfica. Están compuestas por tres elementos básicos: vértices, aristas y caras. Un vértice es un punto en el espacio tridimensional, una arista es una línea que conecta un par de vértices, y una cara, en el caso de las mallas de triángulos, es el área delimitada por tres aristas conectadas.

En la superficie aparecen tres tipos de elementos geométricos (Figura 2.4):

Vértices Un vértice es un punto en el espacio. Está definido por sus tres coordenadas (x, y, z).

Aristas Una arista es un segmento de recta en el espacio delimitado por dos vértices. Se puede representar como un par .

Polígonos Un polígono es una porción de un plano delimitada por una secuencia de aristas conectadas por vértices. Podemos describir el polígono mediante la secuencia ordenada de sus vértices.

Un polígono es simple si sus aristas no consecutivas no se intersectan y las consecutivas se intersectan solo en los vértices comunes. Los polígonos simples pueden ser convexos o concavos. Es **concavo** cuando la recta que contiene alguna de sus aristas divide al polígono en dos, y **convexo** en caso contrario.

| | |
|--|----|
| 2.1 Representaciones poligonales | 11 |
| 2.2 Visualización con OpenGL | 12 |
| 2.2.1 Caras traseras | 13 |
| 2.3 Cálculo de normales de cara | 13 |
| 2.3.1 Normales de vértice | 14 |
| 2.4 Sopa de triángulos | 14 |
| 2.5 Formatos de archivo | 15 |
| 2.6 Herramientas: MeshLab | 16 |
| 2.7 Ejercicios | 18 |

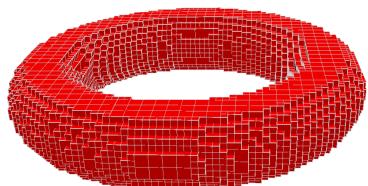


Figura 2.1: Representación de un toroide con un octree.

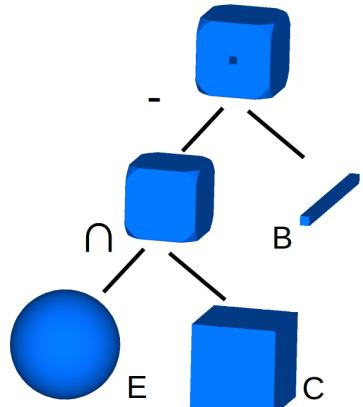


Figura 2.2: Representación de un objeto como una expresión CSG. El objeto se representa como $(E \cap C) - B$.

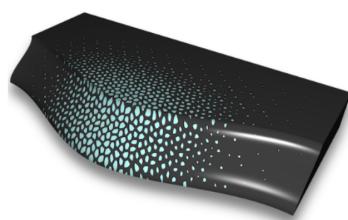


Figura 2.3: Objeto representado mediante volúmenes Bézier paramétricos.

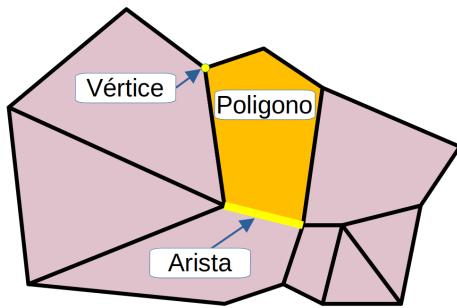


Figura 2.4: Las superficies poligonales están formadas por vértices, aristas y polígonos.

2.2. Visualización con OpenGL

OpenGL utiliza **primitivas** geométricas simples: puntos, líneas, triángulos, cuadriláteros y polígonos, como los bloques constructivos básicos para crear formas más complejas. Por ejemplo, un cuadrado se puede construir usando dos triángulos (o un cuadrilatero), y una esfera se puede aproximar con un gran número de triángulos que aproximan su superficie.

Para dibujar una primitiva (usando legacy mode) se debe llamar a la función **glBegin** indicando como argumento la primitiva que se va a dibujar. A continuación se pasa la información de la (o las) primitivas a dibujar. Los valores que se pueden pasar a **glBegin** para dibujar primitivas simples son los siguientes (Figura 2.5):

GL_POINTS Dibuja un punto en cada uno de los vértices.

GL_LINES Dibuja líneas uniendo cada par de vértices.

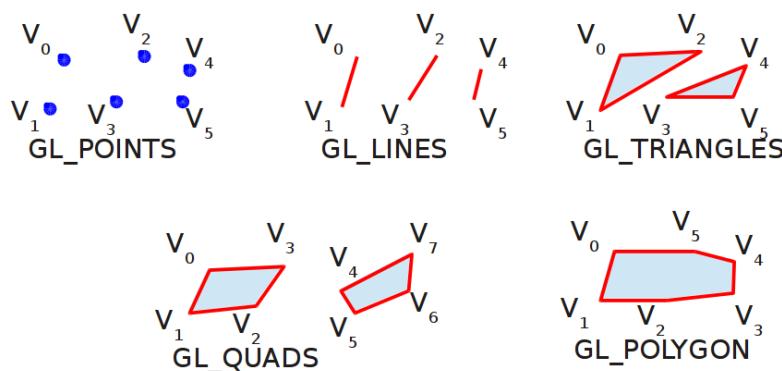
GL_TRIANGLES Dibuja los triángulos formados cada terna de vértices.

GL_QUADS Dibuja cuadriláteros formados por cada grupo de cuatro vértices.

GL_POLYGON Dibuja el polígono formado por la secuencia de vértices completa. Los vértices deben definir un polígono convexo simple.

Los vértices a dibujar se pasan uno a uno con llamadas a la función **glVertex3f**. Se termina con una llamada a **glEnd**. El ejemplo 2.1 se dibuja dos triángulos vecinos a través de la arista (P_0, P_1):

Si se utilizan estas primitivas para dibujar una superficie continua los vértices compartidos por varias primitivas será necesario repetirlos, lo que implica transferirlos a la GPU y procesarlos varias veces. Para reducir estas situaciones se usan primitivas compuestas, que agrupan líneas, triángulos o cuadriláteros (Figura 2.6):



Listing 2.1: Dibujo de dos triángulos

```

1 glBegin(GL_TRIANGLES);
2   // Primer triangulo
3   glVertex3f(x0,y0,z0);
4   glVertex3f(x1,y1,z1);
5   glVertex3f(x2,y2,z2);
6
7   // Segundo triangulo
8   glVertex3f(x1,y1,z1);
9   glVertex3f(x0,y0,z0);
10  glVertex3f(x3,y3,z3);
11 glEnd();
```

Figura 2.5: Resultado de pasar los mismos 6 vértices como diferentes primitivas.

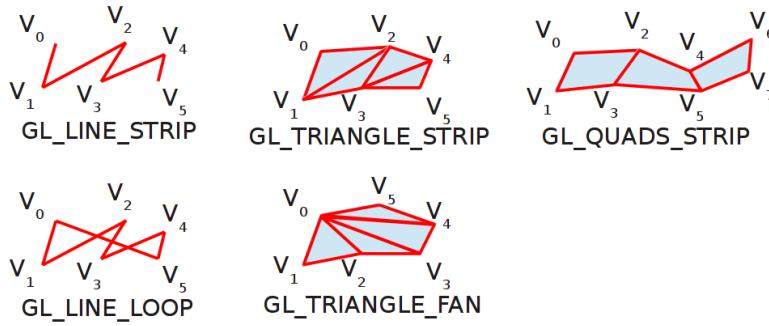


Figura 2.6: Resultado de pasar los mismos 6 vértices como diferentes primitivas complejas.

GL_LINE_STRIP Dibuja una poligonal.

GL_LINE_LOOP Dibuja un polígono.

GL_TRIANGLE_STRIP Dibuja una cinta de triángulos uniendo cada vértice con los dos anteriores.

GL_TRIANGLE_FAN Dibuja un abanico de triángulos. El primer vértice es común a todos los triángulos.

GL_QUADS_STRIP Dibuja una cinta de cuadriláteros.

2.2.1. Caras traseras

Cada polígono tiene dos caras (si te lo imaginas recortado en una cartulina serían las dos caras de la cartulina). Si el polígono forma parte de la superficie de un objeto solo se verá una de las caras de los polígonos¹. Para reducir el número de elementos a dibujar, y por tanto el tiempo de dibujo, las caras traseras no suelen dibujarse. La identificación de las dos caras de los polígonos se realiza comprobando el orden de recorrido de los vértices en la proyección del polígono en pantalla. Por este motivo los vértices deben darse siempre en sentido antihorario (CCW Counter Clockwise) mirandolo desde la cara visible².

1: Salvo que la cámara entre dentro del objeto

2: Este convenio puede cambiarse.

2.3. Cálculo de normales de cara

El color con el que se debe visualizar cada polígono depende del color de su color y de como le incide la luz. Para calcular la iluminación es necesario conocer las normales, que son vectores de módulo uno que indican la dirección perpendicular de la superficie (veremos la relación entre ambas mas adelante). Por este motivo se debe especificar la normal de cada primitiva que se dibuja. Las normales se indican con la función `glvertex3f`.

En el ejemplo 2.2 se asigna normal (1.0, 0.0, 0.0) al primer triángulo y (0.0, 0.0, 1.0) al segundo.

El cálculo de las normales de un triángulo con vértices v_0 , v_1 y v_2 se realiza calculando el producto vectorial de dos de sus aristas, por ejemplo

$$\mathbf{a} = \mathbf{v}_1 - \mathbf{v}_0 \text{ y } \mathbf{b} = \mathbf{v}_2 - \mathbf{v}_0$$

$$\mathbf{n} = (\mathbf{b} \times \mathbf{a}) / \| \mathbf{n} \|$$

El vector \mathbf{n} apunta según la regla de la mano derecha (Figura 2.7).

Listing 2.2: Dibujo con normales

```

1  glBegin(GL_TRIANGLES);
2      // Primer triangulo
3      glNormal3f(1.0,0.0,0.0);
4      glVertex3f(x0,y0,z0);
5      glVertex3f(x1,y1,z1);
6      glVertex3f(x2,y2,z2);
7
8      // Segundo triangulo
9      glNormal3f(0.0,0.0,1.0);
10     glVertex3f(x1,y1,z1);
11     glVertex3f(x0,y0,z0);
12     glVertex3f(x3,y3,z3);
13
14     glEnd();
```

Para que la orientación de las normales sea consistente la orientación de todos los triángulos debe ser la misma.

2.3.1. Normales de vértice

Como veremos mas adelante, para visualizar superficies suaves se calcula la iluminación en los vértices y se interpola el color en el interior de los polígonos (ver figura 2.8). Para ello se debe especificar la normal en cada vértice³. Si la ecuación de la superficie que se está modelando es conocida se puede calcular la normal analíticamente.

En caso contrario se puede calcular la normal en los vértices promediando las normales de las caras (ver Figura 2.9):

$$\mathbf{n} = \frac{\sum_{i=1}^k \mathbf{n}_i}{\|\sum_{i=1}^k \mathbf{n}_i\|}$$

siempre que

$$\left\| \sum_{i=1}^k \mathbf{n}_i \right\| \neq 0$$

siendo n_i la normal en la cara i de las k que comparten el vértice.

Cuando se utilizan normales de vértice se debe llamar a `glNormal3f` antes de cada `glVertex3f` para indicar la normal de vértice.

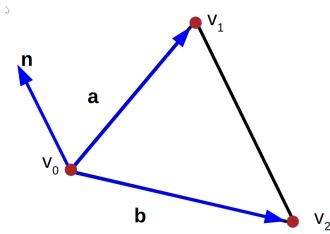


Figura 2.7: Cálculo de la normal de un triángulo

3: Además se le debe indicar a OpenGL usando la llamada a la función de OpenGL `glShadeModel(GL_SMOOTH)`, para volver a dibujar caras planas se usa `glShadeModel(GL_FLAT)`.

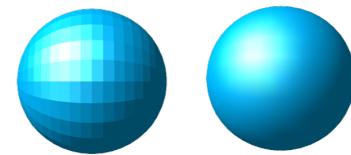


Figura 2.8: Poliedro visualizado con normales de cara (izquierda) y de vértice (derecha)

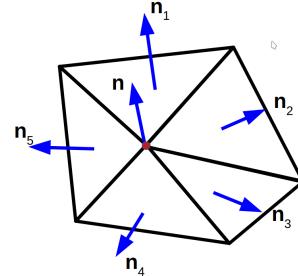


Figura 2.9: Cálculo de la normal de vértice

| | Vertice 1 | | | Vertice 2 | | | Vertice 3 | | |
|----|-----------|-------|-------|-----------|-------|-------|-----------|-------|-------|
| | X | Y | Z | X | Y | Z | X | Y | Z |
| 0 | -6.17 | -4.31 | 5.84 | -2.61 | 2.84 | 7.15 | -5.49 | 0.09 | -2.98 |
| 1 | -8.54 | 4.18 | 1.17 | -3.61 | 7.05 | -7.78 | 5.42 | 8.87 | 0.34 |
| 2 | 6.17 | -6.29 | -3.09 | -1.83 | -0.22 | 3.77 | 8.75 | 9.54 | 5.00 |
| 3 | -8.63 | 5.76 | -9.86 | 9.10 | -6.73 | -1.92 | 8.77 | 9.34 | -3.45 |
| 4 | 6.56 | -6.84 | -1.76 | 4.51 | 8.90 | 2.48 | 5.68 | 4.97 | 0.67 |
| 5 | -0.22 | -4.65 | -3.58 | -1.86 | -6.11 | -5.19 | -3.72 | 2.29 | -5.41 |
| 6 | 2.41 | -1.14 | 6.55 | 0.73 | -0.94 | 8.15 | 3.08 | -1.24 | 2.03 |
| 7 | -8.54 | 5.67 | -6.90 | 4.66 | 2.93 | 6.12 | 9.39 | 1.78 | -3.97 |
| 8 | -7.03 | 4.12 | 7.35 | 9.48 | 9.14 | -7.92 | 3.75 | -3.00 | -9.18 |
| 9 | -6.53 | -8.56 | 1.82 | -5.90 | -9.34 | 5.09 | 5.98 | -1.60 | 5.36 |
| 10 | -0.65 | -2.55 | 7.15 | 0.64 | -1.28 | 8.80 | 6.99 | 5.47 | 2.53 |
| 11 | -8.44 | 7.93 | 5.24 | -5.82 | 9.34 | 8.07 | -9.94 | -8.73 | -8.24 |
| 12 | 7.05 | -2.88 | 6.63 | 6.50 | -8.76 | -9.25 | -2.06 | 9.62 | 1.20 |
| 13 | -6.95 | 4.77 | -2.41 | 3.78 | 4.58 | 9.82 | 0.04 | 9.43 | -0.72 |
| 14 | -1.97 | -3.69 | 5.53 | 8.75 | 9.22 | 6.58 | -6.88 | -0.55 | 4.31 |

Figura 2.10: Representación de una sopa de triángulos

```

1 glBegin(GL_TRIANGLES);
2   for(i=0;i<Mesh.n;i++){
3     glNormal3f(Mesh.t[i].nx, Mesh.t[i].ny, Mesh.t[i].nz);
4     glVertex3f(Mesh.t[i].a.x,Mesh.t[i].a.y,Mesh.t[i].a.z);
5     glVertex3f(Mesh.t[i].b.x,Mesh.t[i].b.y,Mesh.t[i].b.z);
6     glVertex3f(Mesh.t[i].c.x,Mesh.t[i].c.y,Mesh.t[i].c.z);
7   }
8 glEnd();

```

Esta representación tiene varios inconvenientes:

Tiene información redundante. Las coordenadas de los vértices se repiten para cada triángulo que contiene el vértice⁴.

Pueden aparecer fisuras por errores de representación. Las diversas copias de las coordenadas de cada vértice, que están almacenadas como float, pueden no coincidir.

Hacer operaciones es complejo. La mayor parte de las operaciones que se pueden realizar con la superficie requieren poder buscar elementos vecinos. Con esta representación es muy costoso encontrar los triángulos adyacentes a uno dado.

Para resolver los dos primeros problemas evitando la redundancia de vértices se pueden indexar los vértices en lugar de copiar en cada triángulo⁵. De esta forma la representación constará de una lista de vértices y una lista de triángulos, en la que cada triángulo contiene los índices de sus vértices. De esta forma cada vértice se almacena una sola vez y cada triángulo apunta a sus tres vértices. La representación en C usando arrays se muestra en 2.4. La definición de la representación en C++ puede ser la siguiente:

```

1 typedef struct {
2     float x,y,z;
3 } Point; //Coordenadas del vertice
4
5 typedef struct {
6     int v0,v1,v2;
7 } Index; //Indices de los tres vertices de un triangulo
8
9 class Malla : Objeto3D {
10 protected:
11     std::vector < Point > vert; // Lista de vertices
12     std::vector < Index > tri; // lista de triangulos
13     std::vector < Point > normal; // normales de los triangulos
14
15     int nV, nT; // numeros de vertices y triangulos
16
17 public:
18     Malla();
19     ...
20 };

```

Listing 2.3: Estructura de datos de una sopa de triángulos

```

1 typedef struct {
2     float x, y, z;
3 } vertex;
4
5 typedef struct {
6     vertex a, b, c;
7 } triangle;
8
9 typedef struct {
10     int n;
11     triangle* t;
12 } mesh;

```

4: En una malla regular cada vértice está en un promedio de 6 triángulos

5: En la lección siguiente veremos como resolver el problema de la búsqueda de triángulos vecinos.

Listing 2.4: Estructura indexada para una sopa de triángulos

```

1 typedef struct {
2     float x, y, z;
3 } vertex;
4
5 typedef struct{
6     int nv,nt;
7     float vertex[MaxVertex][3];
8     int triangle[MaxTriangle]
9         ][3];
} Mesh;

```

2.5. Formatos de archivo

Para almacenar y transferir mallas de triángulos se suelen usar diversos formatos de archivo. Entre los más comunes están:

PLY Es un formato abierto, sencillo y configurable, diseñado por la Universidad de Stanford.

OBJ Es un formato simple que almacena información básica de vértices y caras. Es popular debido a su simplicidad y facilidad de uso en diferentes plataformas y herramientas.

STL Ampliamente utilizado en la impresión 3D. STL puede almacenar mallas de triángulos utilizando sólo la información de las coordenadas de los vértices de cada triángulo.

```

ply
format ascii 1.0
element vertex 510272
property float32 x
property float32 y
property float32 z
element face 108176
property list uint8 int32 vertex_indices
end_header
0.971510 -1.341210 0.568620
0.982640 -1.344680 0.575240
0.989330 -1.341980 0.579250
.....
0.044910 5.677040 12.747991
3 57004 114239 114246
3 57003 57004 114239
3 57003 114232 114239
3 57002 57003 114232
.....

```

Figura 2.11: Estructura de un archivo PLY.

FBX Es un formato más complejo que puede almacenar no solo mallas, sino también animaciones, texturas, y otros datos relevantes para proyectos de gráficos avanzados y videojuegos.

Los archivos PLY se pueden almacenar en binario o en Ascii, que ocupan mas espacio pero permiten la edición con un editor de textos. Los archivos ply almacenan superficies poligonales indexadas, están organizados en dos áreas (2.11), en la cabecera se identifica el tipo de archivo (en la figura ply almacenado en ascii) y se detalla la información contenida en el archivo. En la figura el modelo tiene 510272 vértices (almacenados como float32 con coordenadas x,y,z) y 108176 caras (cada cara tiene el número de vértices y los identificadores de los vértices). El cuerpo del archivo tiene la lista de vértices (la posición del vértice en la lista es su identificador) y la lista de caras.

Para leer un archivo PLY se debe leer el número de los elementos de la cabecera y a continuación leer y cargar en la estructura de datos cada uno de los vértices. Seguidamente se lee cada cara y se almacena en la lista de caras. En este proceso se pueden detectar errores por duplicación de elementos comprobando que el elemento que se acaba de leer no está ya en la estructura de datos y se pueden calcular y almacenar las normales.

2.6. Herramientas: MeshLab

MeshLab es un software de código abierto utilizado para el procesamiento y edición de mallas 3D⁶ diseñado por el Visual Computing Lab del **ISTI-CNR** [Cig+08]. Es especialmente útil para la limpieza de modelos 3D, la reconstrucción de superficies a partir de puntos 3D y la visualización de mallas.

6: Se puede descargar de <https://www.meshlab.net/#download>

En este curso es especialmente útil para:

- Visualizar modelos, cambiando el modo de visualización.
- Editar modelos: eliminar triángulos o vértices, remallado del modelo.
- Corregir errores: eliminar elementos duplicados, corregir topología, tapar fisuras.

- Generar modelos simples. Puede crear algunos objetos predefinidos (esferas, poliedros simples, terrenos) y calcular operaciones booleanas entre modelos.
- Convertir el formato de los modelos. Nos permite leer archivos en diferentes formatos y exportarlos en un formato diferente.
Importa: PLY, STL, OFF, OBJ, 3DS, COLLADA, PTX, V3D, PTS, APTS, XYZ, GTS, TRI, ASC, X3D, X3DV, VRML, ALN.
Exporta: PLY, STL, OFF, OBJ, 3DS, COLLADA, VRML, DXF, GTS, IDTF, X3D

Para cargar un modelo en MeshLab basta con arrastrar el archivo sobre el icono de meshlab o usar la opción

File > Import Mesh

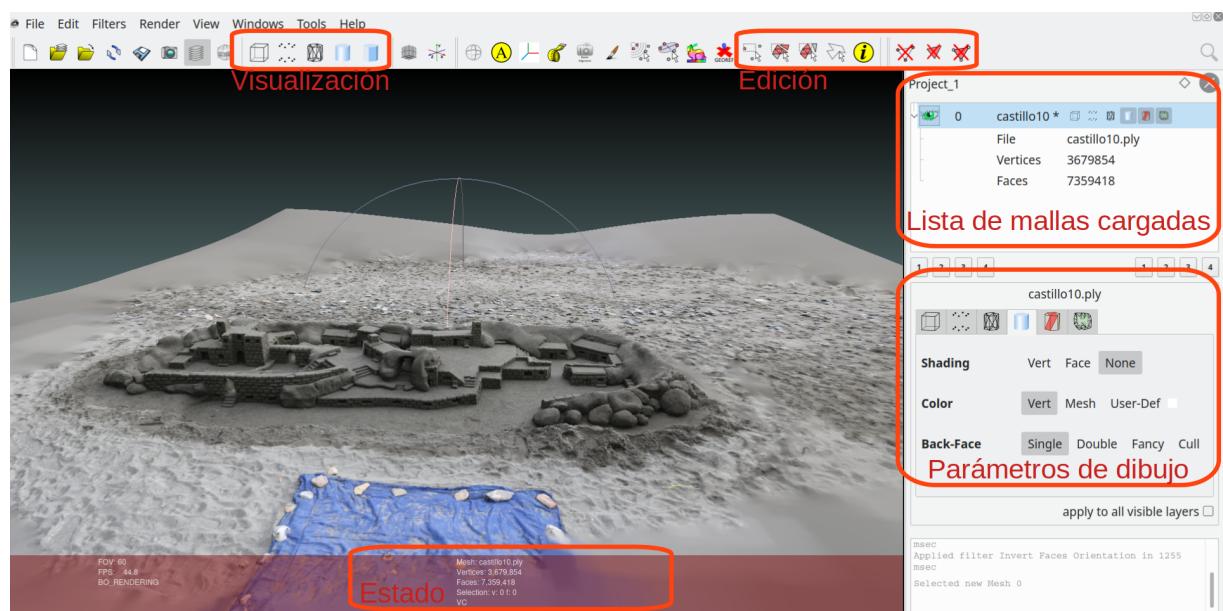


Figura 2.12: Interfaz de usuario de MeshLab.

La Figura 2.12 muestra la interfaz de MeshLab. En ella destacamos los iconos de control de la **visualización**, que controlan que se visualiza (caja envolvente⁷, puntos, aristas o polígonos); La zona de edición, que permite seleccionar y borrar vértices o polígonos; A la derecha el panel de proyecto, en el que se muestra la lista de mallas cargadas, permitiendo seleccionarlas y ocultarlas; Y debajo el panel de parámetros de dibujo de la malla seleccionada, con el que podemos controlar como se visualiza cada tipo de elemento de la malla seleccionada.

En todo momento hay una malla seleccionada, que se destaca en el panel del proyecto con un sombreado celeste (ver Figura 2.12), todas las operaciones se realizan con la malla seleccionada, independientemente de que sea o no visible.

Los parámetros de las mallas se pueden consultar en la barra de estado o en el propio panel del proyecto.

La mayor parte de las operaciones se encuentran en el menú **Filters**. Hay encontraremos las operaciones de creación de mallas **Create New Mesh Layers**, que nos permiten crear mallas simples predefinidas.

7: La caja envolvente es el mínimo paralelepípedo alineado con los ejes que contiene todos los vértices.

Para cambiar el formato de un modelo deberemos abrirlo con MeshLab y usar la orden

File > Export Mesh As

En el panel que aparece deberemos indicar el nombre del archivo y el formato. Por último aparecerá un panel de selección de información a incluir en el archivo.

2.7. Ejercicios

1. Calcular el espacio ocupado por una sopa de triángulos en función del número de vértices para un sistema de 64 bits.
2. obtener cara vecinas de una dada
3. Repetir el cálculo para una estructura indexada.
4. Generar el archivo PLY ascii de un tetraedro con MeshLab.
5. Crear un archivo PLY para representar una pirámide de base cuadrada con un editor de textos. Compruébalo abriendolo con MeshLab.
6. Busca 3 modelos diferentes en formato PLY y comprueba la relación entre número de vértices y de triángulos.
7. Escribe el código OpenGL necesario para dibujar una pirámide de base cuadrada.
8. Diseña un algoritmo para convertir una sopa de triángulos en una estructura indexada.
9. Calcula la complejidad del algoritmo del ejercicio anterior.
10. ¿Qué procesamiento habría que hacer al leer una sopa de triángulos para evitar las fisuras por errores de representación?
11. ¿Pueden aparecer estos errores en una malla indexada?
12. Diseña un algoritmo para determinar si un polígono 2D es simple.
13. Diseña un algoritmo para determinar si un polígono 2D es cóncavo o convexo.
14. Diseña un algoritmo para determinar si un polígono definido en el espacio es plano.

Informática Gráfica

Juan Carlos Torres

Curso 2024/25

Dpto. Lenguajes y Sistemas Informáticos
Universidad de Granada

Disclaimer

You can edit this page to suit your needs. For instance, here we have a no copyright statement, a colophon and some other information. This page is based on the corresponding page of Ken Arroyo Ohori's thesis, with minimal changes.

CC BY-NC-SA

 This book is released into the public domain using the CC BY-NC-SA. This license enables reusers to distribute, remix, adapt, and build upon the material in any medium or format for noncommercial purposes only, and only so long as attribution is given to the creator. If you remix, adapt, or build upon the material, you must license the modified material under identical terms.

To view a copy of the CC BY-NC-SA code, visit:

<https://creativecommons.org/licenses/by-nc-sa/4.0/>

Colophon

This document was typeset with the help of KOMA-Script and L^AT_EX using the kaobook class.

3

Transformaciones geométricas

Las transformaciones geométricas son funciones que modifican las coordenadas de los puntos. Aplicadas a un objeto permiten modificar su posición, orientación o tamaño. En la sección 1.2.3 vimos como definir traslaciones y escalados.

En este tema nos ocuparemos de las rotaciones y de la composición y representación de las transformaciones. Las rotaciones son también transformaciones geométricas básicas, que permiten girar los objetos. El objeto de la parte superior de Figura 3.1 aparece girado en la parte inferior de la Figura.

3.1. Rotaciones

Una rotación en 2D gira los puntos en torno al origen de coordenadas un ángulo α . La transformación R_α de **rotación** de un punto $\mathbf{p} = (x, y)$ es:

$$R_\alpha(\mathbf{p}) = (x \cos(\alpha) - y \sin(\alpha), x \sin(\alpha) + y \cos(\alpha))$$

Un giro con valor de α positivo gira en sentido contrario a las agujas del reloj (sentido antihorario o CCW de las siglas en Inglés de Counter Clockwise).

En 3D hay tres posibles rotaciones básicas, una para cada eje (Figura 3.2). La rotación respecto a un eje no modifica el valor de la coordenada correspondiente a ese eje, para las otros dos coordenadas la expresión es como la de una rotación 2D, con los cambios correspondientes de coordenadas y de signo por la dirección de los ejes.

Por ejemplo, para el giro respecto al eje Y es:

$$R_\alpha^y(\mathbf{p}) = (x \cos(\alpha) + z \sin(\alpha), y, -x \sin(\alpha) + z \cos(\alpha))$$

Los ángulos de rotación en 3D se consideran positivos si se rota en sentido antihorario mirando la escena desde el eje de rotación.

| | | |
|-------|---|----|
| 3.1 | Rotaciones | 19 |
| 3.2 | Composición de transformaciones | 20 |
| 3.3 | Coordenadas homogéneas | 21 |
| 3.4 | TG en OpenGL | 23 |
| 3.5 | Unity | 25 |
| 3.5.1 | Creación de Objetos | 25 |
| 3.5.2 | Cámara y Luces | 26 |
| 3.5.3 | Interacción | 26 |
| 3.6 | TG en Unity | 26 |
| 3.7 | Ejercicios | 27 |

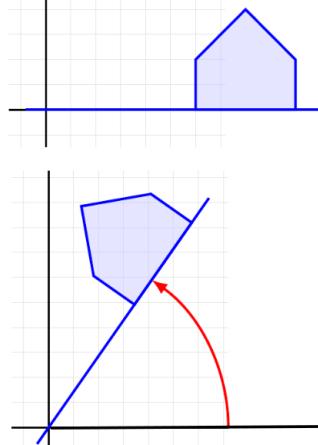


Figura 3.1: Rotación en dos dimensiones.

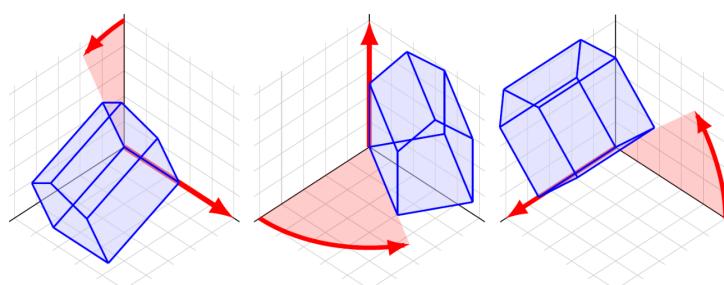


Figura 3.2: Rotaciones básicas en tres dimensiones. Izquierda rotación respecto al eje X , en el centro respecto al Y , derecha respecto al Z .

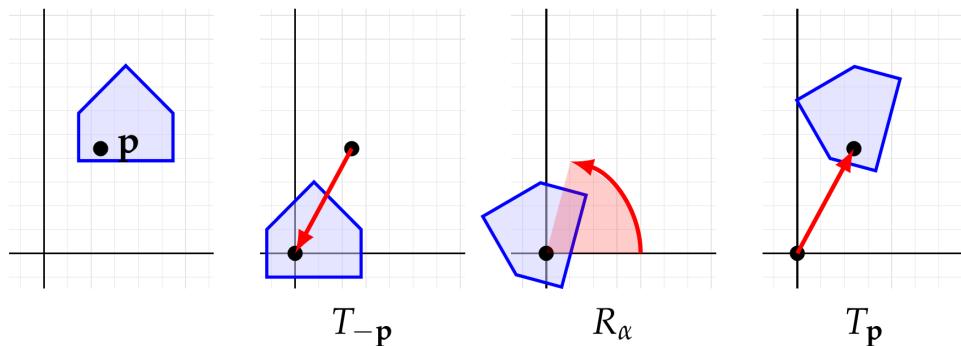


Figura 3.3: Rotación 2D respecto a un punto arbitrario. La figura de la izquierda se rota respecto al punto P realizando las siguientes transformaciones: una traslación T_{-p} , una rotación R_α y una traslación T_p .

3.2. Composición de transformaciones

Las transformaciones geométricas se pueden componer. La composición de transformaciones permite realizar las transformaciones complejas usando varias transformaciones simples. Por ejemplo, para realizar una rotación en 2D respecto a un punto diferente del origen de coordenadas podemos trasladar la figura para llevar el centro de giro al origen, rotar respecto al origen y finalmente volver a llevar el centro de giro a su posición original (Figura 3.3):

$$R_\alpha^P(Q) = T_P(R_\alpha(T_{-P}(Q)))$$

La mayor parte de las transformaciones geométricas tienen inversa. La inversa de una transformación T es la transformación T^{-1} que compuesta con T es la identidad. En el ejemplo anterior T_{-P} es la inversa de T_P .

Podemos componer transformaciones para realizar giros en 3D respecto a ejes arbitrarios. Para girar sobre un eje que pasa por el origen que no coincide con los ejes del sistema de coordenadas, debemos girar respecto a uno de los ejes del sistema de coordenadas para llevar el eje de giro a uno de los planos del sistema de coordenadas (En la Figura 3.4 se ha girado el eje e respecto al eje Y para llevárselo al plano XZ). Seguidamente giramos respecto al eje perpendicular a este plano para llevar el eje de giro hasta uno de los ejes del sistema de coordenadas (en la Figura se gira respecto a Z para llevárselo al eje X). Hacemos el giro y aplicamos las transformaciones inversas para devolver el eje de giro a su posición original.

Calcular los ángulos de giro (β y γ en la Figura 3.4) es simple. Basta con observar que el complementario del ángulo β (naranja en Figura 3.5) está en un triángulo rectángulo cuyos catetos son e_x y e_z , por tanto¹:

$$\beta = 90 - \text{atan2}(e_x, e_z)$$

En la Figura 3.5, los valores de e_x , e_y y e_z están representados como líneas roja, verde y azul respectivamente.

El ángulo γ se puede calcular a partir de la longitud de la proyección del vector e en el plano XZ y de e_y como:

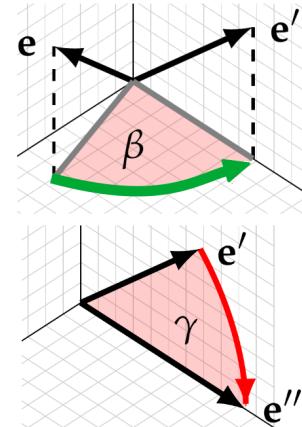


Figura 3.4: Rotación respecto a un eje arbitrario. Para girar respecto al eje e hacemos tres rotaciones: respecto a Y, Z y X .

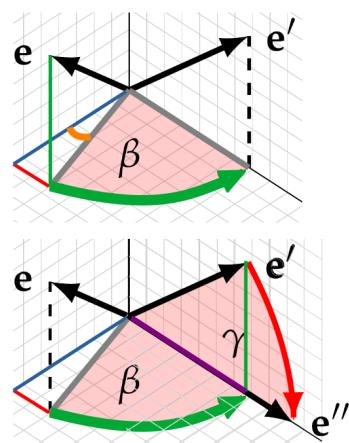


Figura 3.5: Cálculo de los ángulos de rotación para girar respecto a un eje arbitrario.

1: *atan2* devuelve el ángulo cuyo tangente es el cociente de dos números, tomando en cuenta los signos de ambos argumentos para determinar el cuadrante correcto del ángulo.

$$\gamma = \text{atan2}(e_y, \sqrt{e_x^2 + e_z^2})$$

La rotación completa sería

$$R_\alpha^e(Q) = (R_{-\beta}^Y(R_{-\gamma}^Z(R_\alpha^X(R_\gamma^Z(R_\beta^Y(Q))))))$$

Si el eje de rotación no pasa por el origen podemos añadir una traslación para mover el eje al origen de coordenadas.

3.3. Coordenadas homogéneas

Todas las transformaciones que hemos visto (traslación, escalado y rotación) son transformaciones **afines**, esto es, que transforman líneas rectas en líneas rectas, manteniendo las proporciones entre longitudes de segmentos en dichas rectas.

Los escalados y las rotaciones son transformaciones **lineales**. Una transformación T será lineal si y solo si cumple:

$$T(aP + bQ) = aT(P) + bT(Q)$$

para cualquiera par de puntos P y Q , y valores reales a y b . Todas las transformaciones lineales son afines, pero no al contrario.

Las transformaciones lineales se pueden representar como matrices. Para cualquier transformación lineal en \mathbb{E}_3 , existen nueve valores reales (m_{00}, \dots, m_{22}) que determinan la transformación:

$$T(P) = (m_{00}x + m_{01}y + m_{02}z, m_{10}x + m_{11}y + m_{12}z, m_{20}x + m_{21}y + m_{22}z)$$

Es decir la transformación está representada por la matriz $M = (m_{ij})$ (3×3), y su aplicación sobre un punto se puede calcular multiplicándola por el punto (escrito como vector columna)²:

2: En 2D las matrices serían 2×2

$$P' = \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = M P$$

La utilización de matrices facilita la representación de las trasformaciones, su composición y su aplicación.

Las **traslaciones** son afines pero no son lineales, por lo que no pueden representarse como matrices 3×3 . Para poder representar las traslaciones como matrices se utilizan **coordendas homogéneas**.

En coordenadas homogéneas los puntos en \mathbb{E}_n se representan mediante $n + 1$ coordenadas. Es decir, un punto 3D está representado por cuatro coordenadas:

$$P = (x, y, z) = (wz, wy, wz, w)$$

La última componente representa un factor de escala. La tupla (x, y, z, w) representa:

- el punto 3D de coordenadas $(x/w, y/w, z/w)$, si $w \neq 0$.
- la dirección del vector que va desde el origen a (x, y, z) , si $w = 0$ (a estas tuplas se les llama puntos en el infinito).

Los puntos y direcciones 2D en el plano cartesiano también pueden representarse usando el espacio de coordenadas homogéneas 2D, que ahora son ternas de la forma (x, y, w) , que representan

- el punto del plano de coordenadas $(x/w, y/w)$, si $w \neq 0$.
- la dirección del vector que va desde el origen a (x, y) , si $w = 0$.

El espacio de coordenadas homogéneas 2D se puede visualizar de forma semejante a \mathbb{E}_3 . La Figura 3.6 representa este espacio. El plano verde es el plano euclídeo bidimensional en el que cada punto P está representado por una recta que pasa por el origen del espacio en coordenadas homogéneas.

La ventaja de utilizar coordenadas homogéneas es que permite representar todas transformaciones geométricas mediante matrices 4×4 , componiendo transformaciones como producto de matrices³ y aplicarlas multiplicando esta matriz por la matriz columna de coordenadas homogéneas de un punto, que producirá las coordenadas homogéneas del punto transformado.

Dado un punto 3D de coordenadas (x, y, z) se transforma pasándolo a coordenadas homogéneas como $(x, y, z, 1)$, se aplica la transformación multiplicando por la matriz 4×4 que la representa, obteniendo las componentes (x', y', z', w') que se proyecta al punto $(x'/w', y'/w', z'/w')$.

Cualquier transformación lineal T en coordenadas euclídeas 3D (con matriz M) se puede convertir en una transformación lineal en coordenadas homogéneas extendiendo la matriz $3 \times 3 M = (m_{ij})$ añadiéndole una fila y una columna:

$$\begin{pmatrix} m_{00} & m_{01} & m_{02} & 0 \\ m_{10} & m_{11} & m_{12} & 0 \\ m_{20} & m_{21} & m_{22} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

La Figura 3.7 muestra la matriz de una transformación de escalado.

En coordenadas cartesianas, la traslación no puede expresarse como una matriz. Sin embargo, en coordenadas homogéneas, una traslación puede representarse fácilmente con una matriz de transformación, facilitando la combinación de múltiples transformaciones en una sola operación (Figura 3.8).

Además, las coordenadas homogéneas permiten una representación unificada de puntos y vectores (usando el valor cero para w).

En transformaciones afines, el valor de la última coordenada (w) siempre será 0 o 1, sin embargo, en transformaciones proyectivas, w puede tomar otros valores, lo que permite representar proyecciones y perspectivas.

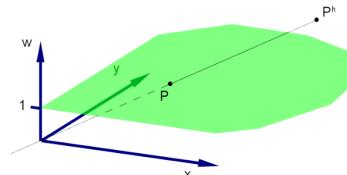


Figura 3.6: Cálculo de los ángulos de rotación para girar respecto a un eje arbitrario.

3: Para transformar un modelo es necesario aplicar la transformación a todos sus puntos, poder componer las matrices de transformación permite aplicar todas las transformaciones realizando una única multiplicación matricial a sus puntos.

$$\begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Figura 3.7: Matriz de escalado 3D en coordenadas homogéneas, con vector de escalado (s_x, s_y, s_z) .

$$\begin{pmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Figura 3.8: Matriz de traslación 3D en coordenadas homogéneas, con vector de traslación (d_x, d_y, d_z) .

3.4. TG en OpenGL

OpenGL almacena, como parte de su estado, una matriz 4x4 que codifica una transformación geométrica como transformación de modelado **modelview** que se aplica a todos los elementos que se dibujan.

Podemos modificar el valor de la matriz modelview usando las llamadas: *glLoadIdentity()* para hacer la matriz de modelado igual a la matriz identidad, o usar llamadas para componer una transformación a la de modelado *glScale*, *glRotate*, o *glTranslate*, con la siguiente sintaxis:

```
glRotatef( GLfloat a, GLfloat ex, GLfloat ey, GLfloat ez )
```

Premultiplica la matriz de modelado por una rotación con eje (*ex*, *ey*, *ez*) de *a* grados.

```
glTranslatef( GLfloat dx, GLfloat dy, GLfloat dz )
```

Premultiplica por una traslación según el vector (*dx*, *dy*, *dz*).

```
glScalef( GLfloat sx, GLfloat sy, GLfloat sz )
```

Premultiplica por un escalado de factor de escala (*sx*, *sy*, *sz*).

Premultiplicar implica que la transformación que se está indicando se aplicará a los objetos antes que las transformaciones que se habían cargado previamente en la matriz de modelado.

El siguiente código dibuja los tres anillos de la figura 3.9.

```

1 void Dibuja( void ){
2     float rojo[4]={1.0,0.3,0.2,1.0}, verde[4]={0.1,1.0,0.2,1.0}, morado
3         [4]={0.8,0.0,0.8,1.0};
4
5     glPushMatrix();
6     glClearColor(1,1,1,1);
7     glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
8
9     dibujar_ejes();
10    glMaterialfv(GL_FRONT,GL_AMBIENT_AND_DIFFUSE,rojo);
11    glutSolidTorus(0.5,3,24,32); // anillo rojo
12
13    glTranslatef(0,4,0);
14    glRotatef(90,0,1,0);
15    glMaterialfv(GL_FRONT,GL_AMBIENT_AND_DIFFUSE,verde);
16
17    glutSolidTorus(0.5,3,24,4); // anillo verde
18
19
20    glTranslatef(0,4,0);
21    glRotatef(90,0,1,0);
22    glMaterialfv(GL_FRONT,GL_AMBIENT_AND_DIFFUSE,morado);
23    glutSolidTorus(0.5,3,24,8); // anillo morado
24
25    glPopMatrix();
26    glutSwapBuffers();
27 }
```

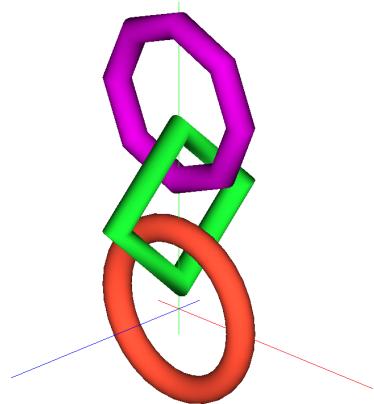


Figura 3.9: Imagen generada con el código de dibujo de tres anillos.

Listing 3.1: Función de dibujo del modelo de la figura 3.9

Los anillos se han dibujado con la función *glutSolidTorus(a, b, n, k)* que dibuja un toro de radio de brazo *a*, radio del toro *b*, aproximado

como una malla con n vértices en los brazos y k vértices en el anillo, sobre el plano XY centrado en el origen de coordenadas. Se han usado diferentes colores y valores de k para distinguir los tres anillos.

El primer anillo dibujado es el rojo (línea 10), no se le ha aplicado ninguna transformación, por lo que se dibuja en el origen.

El segundo anillo dibujado es el verde (línea 17), antes de dibujarlo se han añadido a la transformación de modelado una traslación de (0, 4, 0) y una rotación de 90° en el eje Y, por lo que el anillo verde se rotará 90° y después se elevará 4 unidades en Y⁴.

Por último, para dibujar el tercer anillo (línea 23) se están añadiendo a la transformación de modelado una segunda traslación y otro giro de 90°, por lo que el tercer anillo se rota 90°, se eleva 4 unidades, se vuelve a rotar 90° y se vuelve a elevar 4 unidades, ya que también está afectado por las transformaciones anteriores. El resultado es que se ha elevado 8 unidades y se ha girado 180°.

Además podemos inicializar la transformación de modelado con la identidad:

```
glLoadIdentity()
```

También hay funciones para apilar y desapilar la transformación de modelado:

```
glPushMatrix();  
glPopMatrix();
```

Al llamar a *glPushMatrix* se guarda el valor de la transformación que se recupera invocando a *glPopMatrix*. Este mecanismo es útil cuando se necesita que una transformación no afecte a los elementos que se van a dibujar más adelante. Por ejemplo, supongamos que queremos girar el anillo cuadrado verde para que se muestre tal como se ve en la figura 3.10. Será necesario rotar el anillo verde 45 grados, para ello insertamos una llamada a *glRotatef* en la línea 16 en el siguiente listado. Pero ese giro no debe afectar al anillo morado. Para evitarlo un *glPushMatrix* antes de la rotación (línea 15) y un *glPopMatrix* después del dibujo del anillo verde (línea 18).

```
1 void Dibuja( void ){  
2     float rojo[4]={1.0,0.3,0.2,1.0}, verde[4]={0.1,1.0,0.2,1.0},morado  
3         [4]={0.8,0.0,0.8,1.0};  
4  
4     glPushMatrix();  
5     glClearColor(1,1,1,1);  
6     glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );  
7  
8     dibujar_ejes();  
9     glMaterialfv(GL_FRONT,GL_AMBIENT_AND_DIFFUSE,rojo);  
10    glutSolidTorus(0.5,3,24,32);  
11  
12    glTranslatef(0,4,0);  
13    glRotatef(90,0,1,0);  
14    glMaterialfv(GL_FRONT,GL_AMBIENT_AND_DIFFUSE,verde);  
15    glPushMatrix();  
16        glRotatef(45,0,0,1);  
17        glutSolidTorus(0.5,3,24,4);  
18    glPopMatrix();
```

4: Observa que el orden de aplicación es el inverso al orden en que aparecen en el código

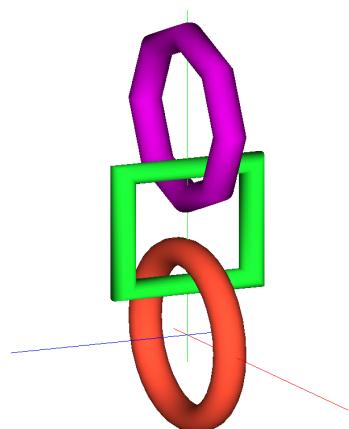


Figura 3.10: Modelo con el anillo verde girado.

```

19
20 glTranslatef(0,4,0);
21 glRotatef(90,0,1,0);
22 glMaterialfv(GL_FRONT,GL_AMBIENT_AND_DIFFUSE,morado);
23 glutSolidTorus(0.5,3,24,8);
24
25 glPopMatrix();
26 glutSwapBuffers();
27 }

```

3.5. Unity

Unity 3D es un motor de desarrollo de aplicaciones gráficas interactivas que ofrece un entorno amigable y versátil. Permite la creación de experiencias tanto en 2D como en 3D y es compatible con múltiples plataformas, incluyendo PC, consolas, móviles y realidad virtual. La programación en Unity se realiza en C#, y cuenta con una comunidad de desarrollo activa y una gran cantidad de recursos y activos disponibles.

Para comenzar a usar Unity, es necesario descargar el [Unity Hub](#), disponible para Linux, Mac y Windows. A través del Unity Hub se gestionan los proyectos y la instalación del editor. Es importante asegurarse de incluir el soporte para Linux Build Support, ya que algunas entregas deberán realizarse en Linux.

Los elementos principales de la interfaz de Unity son (figura 3.12):

Vista de Escena (Scene View) : Permite navegar y editar visualmente la escena. Puede mostrar una perspectiva 2D o 3D, dependiendo del tipo de proyecto.

Ventana de Proyecto (Project Window) : Muestra los assets disponibles en la librería del proyecto. Aquí aparecen los assets importados.

Ventana de Jerarquía (Hierarchy Window) : Representa en texto jerárquico cada objeto en la escena, revelando la estructura de cómo están agrupados.

Ventana del Inspector (Inspector Window) : Permite visualizar y editar las propiedades del objeto seleccionado. El contenido varía según el objeto.

Barra de Herramientas (Toolbar) : Proporciona acceso a herramientas esenciales para manipular la vista de escena y los objetos, controles de reproducción, y opciones de layout del editor¹.

3.5.1. Creación de Objetos

En Unity, la creación de objetos comienza con primitivas simples cuyas propiedades se pueden ajustar en el panel inspector. Los objetos pueden ser transformados geométricamente seleccionándolos y modificando sus parámetros de transformación o utilizando herramientas de edición interactivas. Las transformaciones incluyen posición, orientación y escalado.

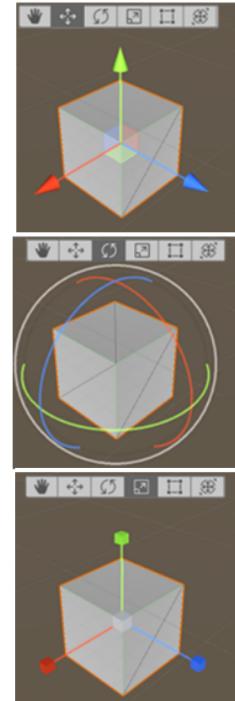


Figura 3.11: Manejadores de transformación en Unity. De arriba a abajo: traslación, rotación y escalado.

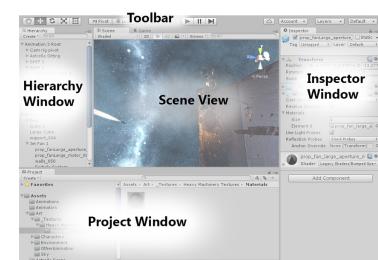


Figura 3.12: Elementos principales de la interfaz de Unity.

3.5.2. Cámara y Luces

Cada escena en Unity contiene al menos una cámara y una luz. Al seleccionar una cámara, aparece una ventana con la vista de la cámara y sus propiedades se pueden ajustar en el panel inspector. Las luces también tienen propiedades ajustables en el panel inspector.

3.5.3. Interacción

Se pueden asociar scripts a los objetos para definir su comportamiento. Cada script tiene dos métodos principales: *Start*, que se ejecuta al iniciar el sistema, y *Update*, que se ejecuta en cada frame. Por ejemplo, un script puede permitir que el programa responda a eventos de entrada, como presionar una tecla para salir de la aplicación.

3.6. TG en Unity

Cada objeto en Unity tiene asociado una transformación, que está formada por un escalado, una rotación y traslación. Los valores de las transformaciones se pueden ver en el panel del inspector (Figura 3.13).

Para transformar un objeto debemos seleccionarlo (en el panel Hierarchy o en la escena) y modificamos los parámetros de sus transformaciones en el panel del inspector.

Alternativamente, podemos modificar sus transformaciones interactivamente usando los manejadores de traslación, rotación o escalado (Figura 3.11).

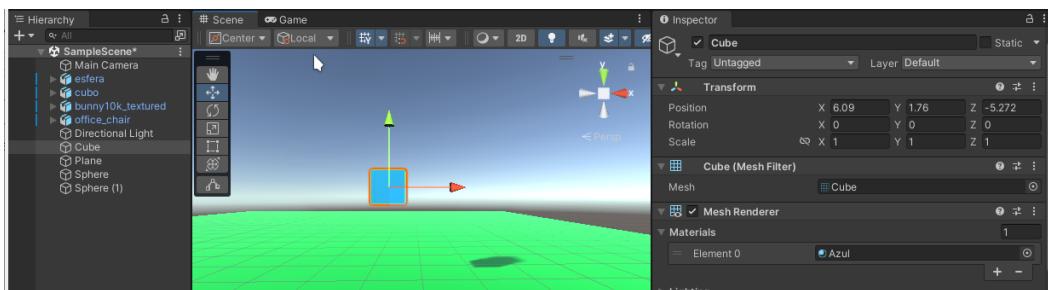


Figura 3.13: Las transformaciones aplicadas a un objeto en Unity se muestran en su panel *Inspector*.

3.7. Ejercicios

- Se dispone de una función *caja()* que crea un paralelepípedo de dimensiones $4 \times 1 \times 2$ como el mostrado en la parte superior de la figura 3.14, escribe el código en OpenGL para dibujar el modelo de la parte inferior de la figura.
- Utilizando la misma función *caja()* escribe el código en OpenGL para dibujar el modelo de la figura 3.15.
- ¿Qué dibuja el siguiente código OpenGL?

```

1 for(i=0;i<20;i++){
2     glTranslatef(0.0, 1.0, 0.0);
3     glRotatef(20.0,0.0,1.0,0.0);
4     caja();
5 }
6 glPopMatrix();

```

En el que *caja()* crea un paralelepípedo de dimensiones $4 \times 1 \times 2$ como el mostrado en la parte superior de la figura 3.14.

- ¿Qué secuencia de transformaciones se deben usar para realizar un giro respecto a eje vertical que pasa por el punto $(2, 0, -1)$?
- Calcula las rotaciones que ese necesario realizar para hacer un giro respecto al eje $(1,1,1)$.
- ¿Como se puede deducir formula de rotación respecto al eje Z?
- ¿Es igual gira 30° respecto al eje $(0,1,0)$ que hacerlo con al eje $(0,-1,0)$?
- La linea 16 del segundo listado de código hace un *glRotatef*(45, 0, 0, 1) rotando respecto al eje Z ¿Es correcto o debería rotar respecto al eje X?
- Escribe el código OpenGL necesario para crear el modelo de la figura 3.16 usando la función *glutSolidTorus*. Los toros tienen radio interior 1 y radio exterior 5.
- Escribe el código OpenGL necesario para crear el modelo de la figura 3.17 usando la función *glutSolidTorus*. Los toros tienen radio interior 1, el radio exterior de los interiores es 8.

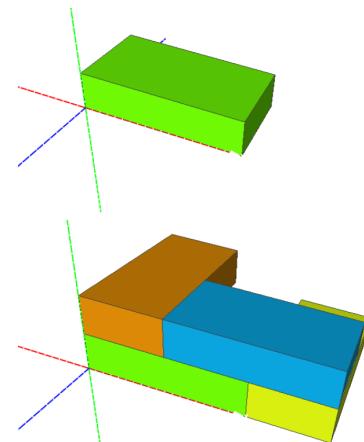


Figura 3.14: Ejercicio 4.1.

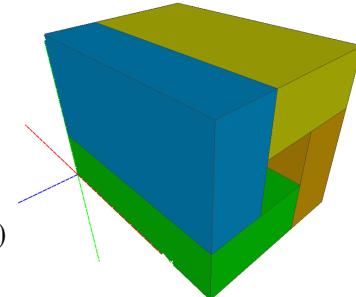


Figura 3.15: Ejercicio 4.2.

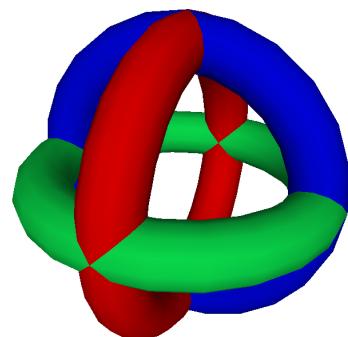


Figura 3.16: Ejercicio 4.9.

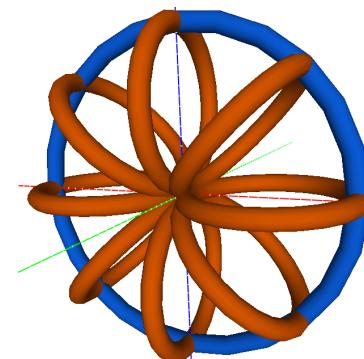


Figura 3.17: Ejercicio 4.10.

Informática Gráfica

Juan Carlos Torres

Curso 2024/25

Dpto. Lenguajes y Sistemas Informáticos
Universidad de Granada

Disclaimer

You can edit this page to suit your needs. For instance, here we have a no copyright statement, a colophon and some other information. This page is based on the corresponding page of Ken Arroyo Ohori's thesis, with minimal changes.

CC BY-NC-SA

 This book is released into the public domain using the CC BY-NC-SA. This license enables reusers to distribute, remix, adapt, and build upon the material in any medium or format for noncommercial purposes only, and only so long as attribution is given to the creator. If you remix, adapt, or build upon the material, you must license the modified material under identical terms.

To view a copy of the CC BY-NC-SA code, visit:

<https://creativecommons.org/licenses/by-nc-sa/4.0/>

Colophon

This document was typeset with the help of KOMA-Script and L^AT_EX using the kaobook class.

4

Grafos de escena

Los modelos no suelen componerse de un único objeto (o una única malla de polígonos). Las escenas de cualquier aplicación real están compuestas de multitud de objetos. Incluso elementos que podemos identificar conceptualmente como un objeto (como por ejemplo un personaje o un vehículo) están integrados por diferentes componentes.

Cada uno de estos objetos y componentes debe tener su geometría definida y debe estar ubicado en la posición correcta en la escena. Aunque es posible definir la geometría de los objetos indicando directamente su posición en la escena, esto no es práctico por varias razones:

- Dificulta la definición de la geometría. Si piensas en la escena de la figura 4.1 tendríamos que definir la posición de cada vértice de cada silla, teniendo en cuenta donde está y que orientación tiene.
- Utiliza información redundante. Los elementos que aparecen repetidos en la escena contienen su información completa.
- Dificulta la edición. Si queremos modificar el modelo de silla tendremos que editar todas las sillas. Incluso, si simplemente queremos desplazar un pupitre tendremos que editar todos los elementos vinculados a él.

Para reducir estos problemas se estructura la escena, de forma que la definición de la geometría sea independiente de la ubicación en la escena, usando transformaciones geométricas, y haciendo que los componentes que están conceptualmente conectados (como la rueda y el chasis de un vehículo) comparten parte de las transformaciones, permitiendo que se puedan transformar de forma solidaria.

En este capítulo veremos como estructurar la escena para minimizar redundancias y facilitar la edición.

4.1. Grafos de escena

Para facilitar el diseño y colocación de los objetos de la escena, estos se definen en su propio sistema de coordenadas (normalmente con su centro, o uno de los vértice de su caja envolvente, en el origen). Por ejemplo, para construir el Buggy de la figura 4.2 podemos crear una rueda delantera, colocada con el centro en el origen de coordenadas (Figura 4.10). De esta forma es fácil transformar sus posiciones.

Para colocar los componentes podemos hacer un escalado, S (para cambiar su tamaño y sus proporciones), una rotación, R (para cambiar su orientación) y una translación, T (para ubicarlo en su posición). Las transformaciones se realizan en este orden, porque es más fácil pensar los parámetros de cada transformación de este modo. La figura 4.5 muestra el efecto de las tres transformaciones 2D en un cuadrado.

Por ejemplo, si queremos colocar el cubo de imagen izquierda de la figura 4.4 para crear el componente que aparece en la imagen a la derecha de la

| | | |
|-------|---------------------------------------|----|
| 4.1 | Grafos de escena | 28 |
| 4.2 | Grafos de escena en OpenGL | 30 |
| 4.3 | Grafos de escena en Unity | 31 |
| 4.4 | Diseño de grafos de escena | 32 |
| 4.4.1 | Descomposición de la escena | 33 |
| 4.4.2 | Diseño de los nodos | 33 |
| 4.4.3 | Implementación del modelo | 34 |
| 4.5 | Ejercicios | 34 |

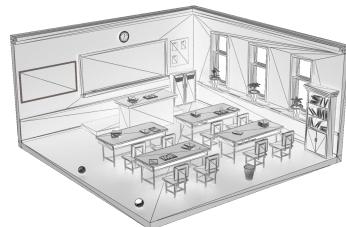


Figura 4.1: Escena simple (Philip Storm: Modelo 3D de aula de dibujos animados de Lowpoly).

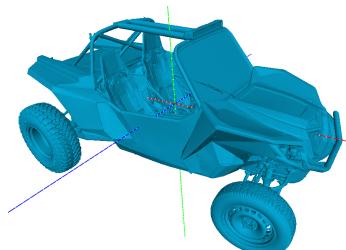


Figura 4.2: Modelo 3D de un buggy (Buggy de Artec 3D).

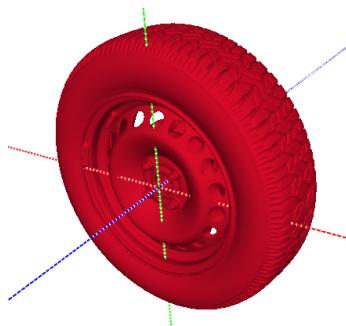


Figura 4.3: Rueda delantera del buggy (Buggy de Artec 3D).

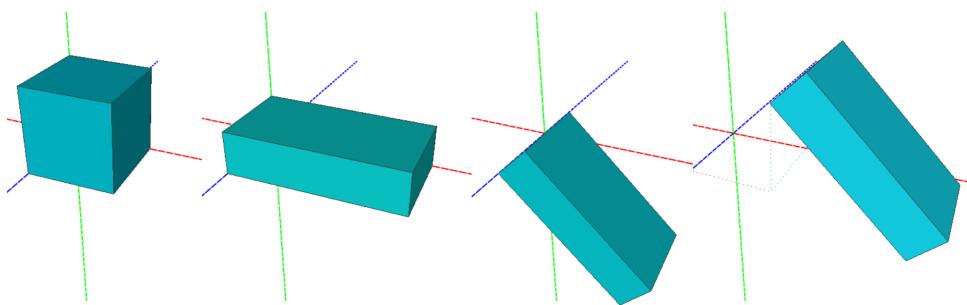


Figura 4.4: Instanciación del cubo de dimensiones $1 \times 1 \times 1$ de la izquierda como el paralelepípedo de la derecha. Aplicamos de izquierda a derecha: un escalado $S(2, 0.5, 1)$, una rotación $Rz(45)$ y una traslación $T(1, 1, 0)$

figura. Si aplicamos las transformaciones en el orden escalado, rotación y traslación (es decir $T(R(S(P))))$), podemos calcular el factor de escala mirando el tamaño final del modelo ($2 \times 0.5 \times 1$) y la rotación viendo que aparece inclinado 45° en Z, y por último la traslación comprobando que el vértice que está en el origen debe ir al punto $(1, 1, 0)$.

Las figuras 4.6, 4.7 y 4.8 muestran el efecto de aplicar las transformaciones en diferente orden.

No obstante, la transformación no es única. En este ejemplo podríamos haber alargado el cubo hacia el eje Y y después haberlo girado 135° . También podríamos haber realizado alguna de las transformaciones en un orden distinto, por ejemplo trasladar antes de rotar, aunque de esa forma es mucho mas difícil calcular la traslación que debemos aplicar, ya que la posición final va a depender también de la rotación.

Por otra parte, hay situaciones en las que se deben realizar las transformaciones necesariamente en otro orden. Por ejemplo, para crear el romboide de la figura 4.9 a partir del cubo anterior, debemos rotar 45 y después escalar en $(2, 0.5, 1)$.

Reutilizar los componentes evita duplicar su información. El modelo de la figura 4.2 tiene 200.000 vértices y ocupa en disco mas de 8MB.

Además, si queremos cambiar el diseño de rueda solo tenemos que cambiar el modelo que estamos instanciando. Y para cambiar su colocación en el modelo bastará con modificar la transformación geométrica.

Podemos usar el mismo proceso para ubicar el buggy en un escenario, instanciando varias veces el modelo utilizando diferentes transformaciones geométricas. De esta forma podemos generar una representación de la escena como Grafo Acíclico Dirigido (DAG), en la que definimos los elementos a un nivel instanciado los componentes del nivel inmediatamente inferior. Esta estructura se suele denominar Grafo de escena en las aplicaciones gráficas.

En el grafo de escena cada nodo representa un componente de la escena y puede contener transformaciones geométricas, geometría y referencias a nodos hijo. Los nodos internos pueden contener información geométrica propia y además instancian nodos a mas bajo nivel. El nodo raíz representa la escena completa.

Las transformaciones definidas en cada nodo afectan a la geometría de ese nodo y a sus nodos subordinados.

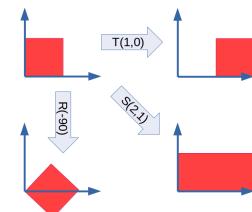


Figura 4.5: Transformaciones geométricas 2D.

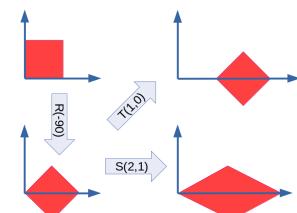


Figura 4.6: Efecto de aplicar la rotación en primer lugar.

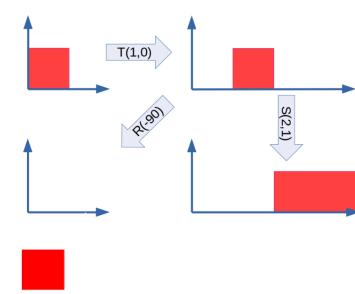


Figura 4.7: Efecto de aplicar antes la traslación.

4.2. Grafos de escena en OpenGL

OpenGL no guarda el modelo de la escena. Cada vez que se debe dibujar un nuevo frame se ejecuta el código de dibujo, por tanto la estructura de la escena estará reflejada en nuestro código.

Si el código está escrito en *C* (u otro lenguaje no orientado a objetos) la forma más simple de representar el grafo de escena es creando una función para dibujar cada nodo. Las funciones de los nodos internos deben llamar a las funciones que dibujan los nodos hijos para instanciarlos. De esta forma se puede instanciar un nodo múltiples veces. En la función que dibuja cada nodo hacemos uso de las funciones **glPushMatrix** y **glPopMatrix** para limitar el ámbito de aplicación de las transformaciones geométricas.

Por supuesto podemos implementar el grafo sin hacer que cada nodo se correspondan con una función, usando bloques **glPushMatrix - glPopMatrix**. Cada bloque equivale a bajar un nivel en el grafo de escena, ya que las transformaciones que se incluyan en este bloque no afectarán a los elementos que se dibujen después de él.

A modo de ejemplo el siguiente código dibuja el modelo de figura 4.11 usando la función **glutSolidCube** para dibujar la primitiva cubo.

```

1 void Base(){
2     glMaterialfv( GL_FRONT, GL_AMBIENT_AND_DIFFUSE, green );
3     glPushMatrix();
4         glTranslatef(0.0,2.5,0.0);
5         glutSolidCube(5);
6     glPopMatrix();
7 }
8
9 void Tapa(){
10    glPushMatrix();
11    glTranslatef(1.25,0.25,0);
12    glScalef(0.5,0.1,1);
13    glutSolidCube(5);
14    glPopMatrix();
15 }
16
17 void Caja(){
18     Base();
19     glPushMatrix();
20     glMaterialfv( GL_FRONT, GL_AMBIENT_AND_DIFFUSE, blue );
21     glTranslatef(-2.5,5,0);
22     glRotatef(30,0,0,1);
23     Tapa();
24     glPopMatrix();
25     glPushMatrix();
26     glMaterialfv( GL_FRONT, GL_AMBIENT_AND_DIFFUSE, red );
27     glTranslatef(2.5,5,0);
28     glRotatef(-40,0,0,1);
29     glTranslatef(-2.5,0,0);
30     Tapa();
31     glPopMatrix();
32 }
```

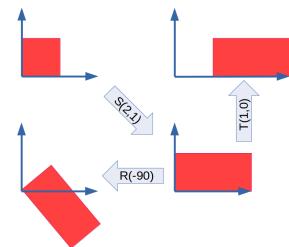


Figura 4.8: Efecto de aplicar el escalado en primer lugar.

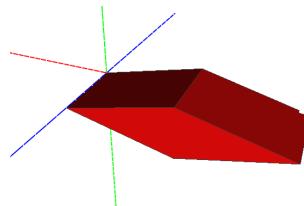


Figura 4.9: Romboide generado a partir del cubo de la figura 4.4. Se le ha aplicado una rotación $R_z(45)$ y un escalado $S(2, 0.5, 1)$.

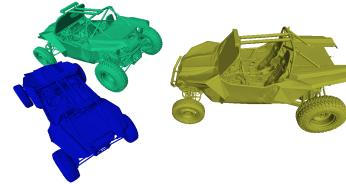


Figura 4.10: Escena con varios buggys (Buggy de Artec 3D).

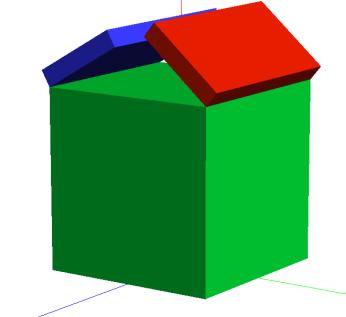


Figura 4.11: Ejemplo de modelo simple creado como un modelo jerárquico usando OpenGL.

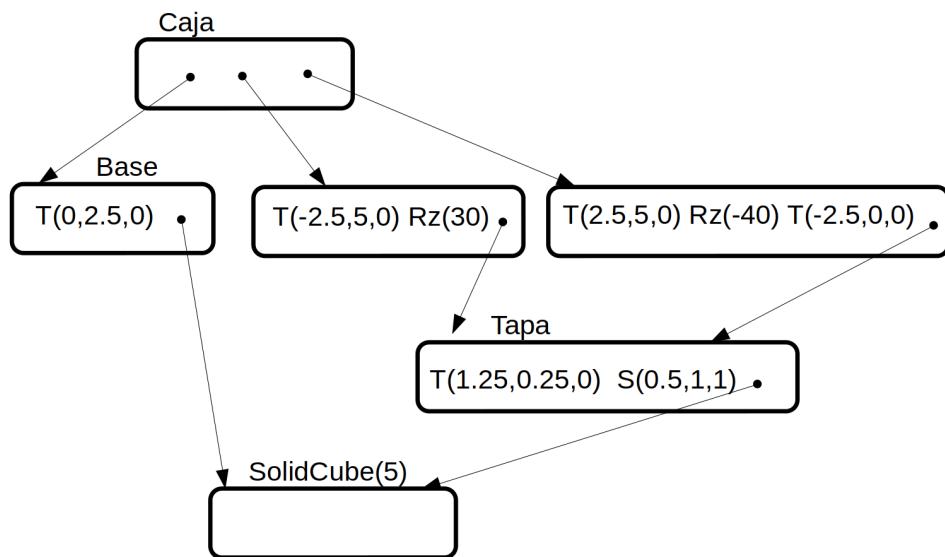


Figura 4.12: Grafo de escena del modelo de la figura 4.11.

Las dos tapas (azul y roja) se han dibujado con la función *Tapa* que es llamada dos veces en la función *Caja*.

El grafo de escena correspondiente a este código se muestra en la figura 4.12.

En este modelo se han utilizado escalados, traslaciones y rotaciones. Las transformaciones geométricas se aplican a las normales, y estas se utilizan en el cálculo de iluminación. Si se utilizan escalados es necesario que OpenGL normalice los vectores normales (los ajuste para que tengan módulo 1), esto se consigue utilizando la llamada

```
glEnable(GL_AUTO_NORMAL)
```

4.3. Grafos de escena en Unity

En Unity el grafo de escena se muestra en el panel *Hierarchy* situado a la izquierda (ver figura 4.13). En él aparece la jerarquía de nodos que conforman el grafo (y algunos elementos más como luces y cámaras que veremos en la lección 6).

Los nodos contienen:

- Un objeto. Los objetos se instancian tomando los assets del panel *Project*.
- Su transformación geométrica, dada como un escalado, una rotación y una traslación. Los valores de las transformaciones se pueden consultar y modificar en el panel *Inspector*.
- Opcionalmente pueden tener una lista de nodos hijos. En la figura 4.13 el nodo *Molino* contiene los nodos *A* y *cuerpo*.

Esta forma de asociar las transformaciones a los nodos facilita la representación y edición del grafo, pero impide realizar determinadas composiciones de transformaciones dentro de los nodos. Vimos que, por ejemplo, para girar respecto a un eje que no pasa por el origen era necesario trasladar antes de realizar la rotación. Para permitir que se

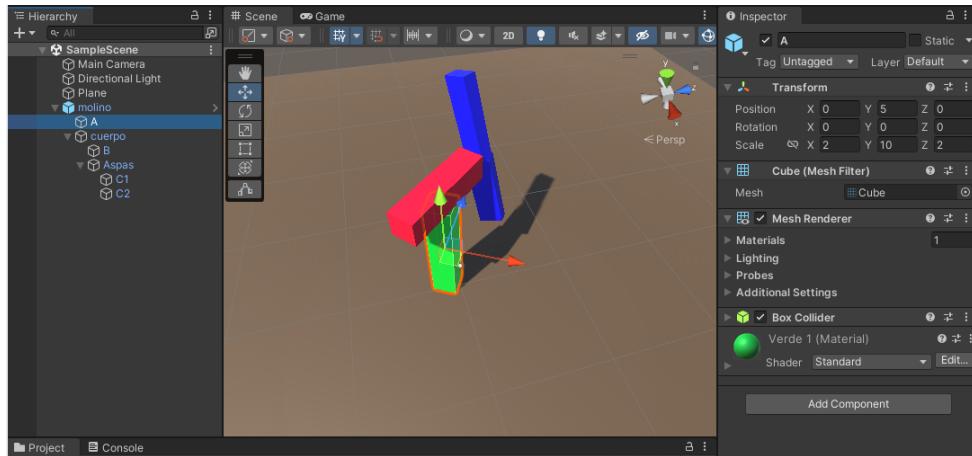


Figura 4.13: El grafo de escena de un proyecto de Unity se muestra en el panel izquierdo (Hierarchy). En este ejemplo Molino está formado por los componentes A y Cuerpo; y este último por B y Aspas; que a su vez está formado por C1 y C2.

puedan realizar estas transformaciones se contempla la posibilidad de incluir nodos sin geometría (*Empty*). De esta forma podemos componer mas transformaciones colocando como padre del nodo un nodo vacío. En la figura 4.13 los nodos *Aspas* y *cuerpo* son nodos vacíos.

En el grafo de escena de la figura 4.12 hay un nodo que no puede crearse en Unity por tener una traslación antes de una rotación. Este nodos se tendría descomponer en dos nodos, en el padre estarían la primera traslación y la rotación y en el hijo la segunda traslación.

Hay otras dos características que ayuda a reducir la complejidad del grafo de escena, a pesar de tener solo un escalado, una rotación y una traslación en cada nodo. Por un lado las rotaciones se indican como tres ángulos de rotación: ángulo de rotación respecto a X, rotación respecto a Y y respecto a Z, en este orden. Es decir

Por otro lado las rotaciones se pueden realizar respecto al centro geométrico del nodo o respecto a su punto pivote. Podemos colocar el punto pivote en la posición que deseemos colocando un nodo vacío en esa posición y haciendo que el objeto que queremos rotar dependa del nodo vacío.

4.4. Diseño de grafos de escena

Diseñar el grafo de escena implica pensar la estructura del modelo y las transformaciones geométricas que contiene. Para su realización se pueden seguir los siguientes pasos:

1. Descomponer la escena recursivamente en componentes mas simples. Este proceso genera el esqueleto del grafo de escena.
2. Decidir las primitivas y transformaciones geométricas que se deben incluir en cada nodo.

Tal como hemos visto anteriormente la estructura de los nodos

4.4.1. Descomposición de la escena

Descomponemos el objeto que queremos representar en partes mas simples. Por ejemplo si queremos crear el grafo de escena del molino de viento de la figura 4.13 podemos descomponerlo en el soporte vertical (pieza A, verde en la figura) y el resto del molino. Aplicamos este proceso a los componentes resultantes hasta obtener elementos que podamos construir de forma sencilla. En el ejemplo de la figura 4.13 no es necesario subdividir el pie ya que es un simple paralelepípedo pero si debemos descomponer el resto del molino.

En este proceso de descomposición debemos tener en cuenta que las partes del modelo que se vayan a reutilizar varias veces, como por ejemplo las sillas de una sala, deben ser un componente (que instanciaremos en distintas partes de la escena).

Por otra parte, los componentes que deban estar afectados por la misma transformación geométrica deben estar en la misma rama del grafo. Esto ocurre por ejemplo en los modelos articulados, como el buggy de la figura 4.2, todos los componentes de la rueda deben estar en la misma rama del grafo, para que giren solidariamente al girar esta.

Como resultado de este proceso obtendremos el grado del modelo y un boceto de cada uno de los nodos. El boceto es esencial para aclarar lo que queremos crear en cada nodo y para poder realizar el paso siguiente. La figura 4.14 muestra una posible descomposición del molino de la figura 4.13.

Tal como hemos visto, el contenido de cada nodo puede depender del sistema en el que se vaya a implementar el modelo, por lo que es posible que al incluir las transformaciones geométricas haya que realizar ajustes en el grafo.

4.4.2. Diseño de los nodos

El siguiente paso es diseñar el contenido de los nodos, es decir, decidir que primitivas y transformaciones geométricas se van a incluir en cada nodo. Si se va a implementar en OpenGL deberemos también decidir en qué orden se aplican. Este proceso se puede realizar de forma independiente para cada nodo, mirando de que componente partimos, lo que representan los nodos inferiores instanciados en el nodo que estamos diseñando, y lo que debe representar este nodo.

Para ello es esencial hacer un boceto de la escena y de cada uno de los nodos (indicando sus dimensiones y ubicándolo en su sistema de coordenadas). Esto es, saber exactamente que geometría debe generar cada nodo.

Por ejemplo, si en el grafo anterior (figura 4.14) partimos de una primitiva para dibujar el Aspa (*primitivaAspa*) como un paralelepípedo con tamaño $1 \times 6 \times 2$ dibujado en el primer octante con un vértice en el origen de coordenadas (figura 4.15), y nuestro nodo *Aspa* debe representar el componente mostrado en la figura 4.16, debemos realizar en el nodo *Aspa* una traslación de $(-0.5, 0, -1)$ antes de instanciar la *primitivaAspa*.

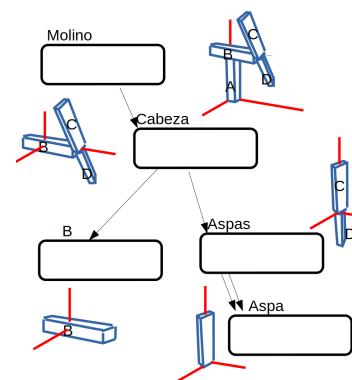


Figura 4.14: Ejemplo descomposición del modelo.

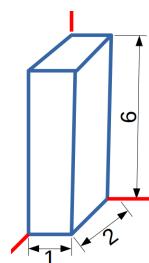


Figura 4.15: Primitiva usada para crear las aspas del molino.

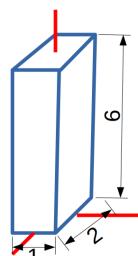


Figura 4.16: Boceto del nodo Aspa.

Si en el nodo *Aspas* las dos palas deben estar giradas 20 grados, cada una en un sentido diferente, deberemos rotar la aspa superior -20° respecto a Y, la segunda aspa se debe desplazar hacia abajo 6 unidades (traslación de $(0, -6, 0)$) y rotarla 40° respecto al eje Y, tal como se muestra en la figura 4.17.

Recuerda que las transformaciones se aplican de derecha a izquierda, por tanto la interpretación de este nodo es: dibujar un *Aspa* la trasladamos 6 unidades en dirección -Y, la rotamos 40° respecto a Y, dibujamos otra *Aspa* y rotamos el conjunto -20° respecto a Y.

El contenido permitido para los nodos será distinto dependiendo del sistema en el que se vaya a implementar el modelo. El nodo diseñado para las *Aspas* se puede implementar en OpenGL, pero no en Unity. En este último tendríamos que subdividirlo.

4.4.3. Implementación del modelo

Al implementar el modelo geométrico pueden aparecer errores, tanto por haberse producido fallos en el diseño como por cometerse errores en la programación. Es mas fácil detectar y corregir estos errores cuando se implementa un modelo pequeño que al probar un sistema grande. Por esta razón el modelo se debe implementar y probar componente a componente.

Esto, que es válido para cualquier tipo de sistema, en nuestro caso se realiza programando el grafo de escena nodo a nodo, comenzando con los nodos mas profundos y comprobando que el modelo generado por cada nodo se corresponden con lo previsto en los bocetos incluidos en el grafo de escena.

De esta forma los posibles errores encontrados estarán circunscritos a un nodo, y será mas fácil corregirlos tanto si son de código como si son de diseño.

4.5. Ejercicios

1. ¿Qué significa y que implica que el grafo de escena sea acíclico?
2. ¿Puede el grafo de escena ser un árbol?
3. Calcula las transformaciones geométricas 2D necesarias para generar cada uno de los elementos de la parte inferior de la figura 4.18 a partir de la primitiva de la parte superior.
4. Dibuja el boceto de cada nodo del grafo de escena de la figura 4.12.
5. Genera el grafo de escena del modelo de derecha de la figura 4.19 usando las primitivas A y B que aparecen a la izquierda.
6. Calcula las transformaciones geométricas necesarias para generar cada uno de los cubos de la parte derecha de la figura 4.20 a partir de la primitiva de la izquierda.
7. Diseña el grafo de escena para construir el objeto de la figura 4.21 usando como componente el objeto creado en el ejercicio 6.
8. Diseña el nodo *Aspas* (figura 4.17) para Unity.
9. Completa el grafo de escena de la figura 4.14.

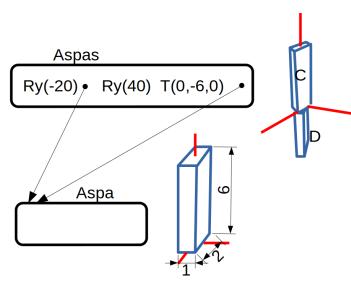


Figura 4.17: Diseño del nodo *Aspas*.

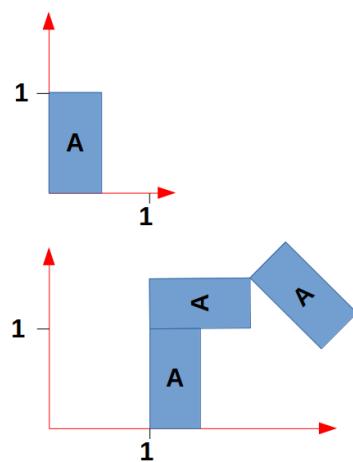


Figura 4.18: Calcula las transformaciones geométricas (ejercicio 3).

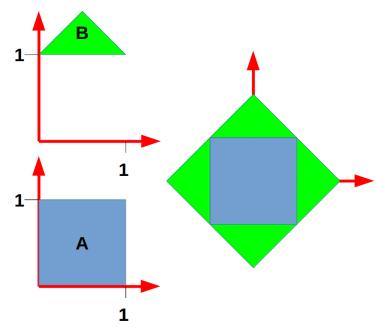


Figura 4.19: Genera el grafo de escena (5).

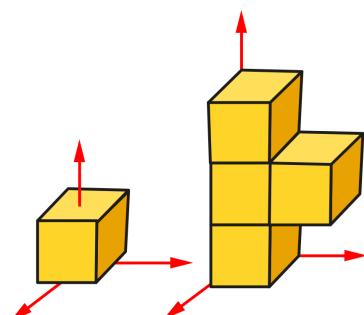


Figura 4.20: Calcula las transformaciones geométricas (ejercicio 6).

10. Diseña el grafo de escena del modelo articulado de la figura 4.22, teniendo en cuenta la pieza B puede girar sobre su eje vertical y las C y D giran respecto al horizontal. Supón que todas las piezas son prismas de dimensión $2 \times 2 \times 10$ y que se usa como única primitiva un paralelepípedo de dimensiones $(2, 10, 2)$ con un vértice en el origen de coordenadas.
11. Diseña el grafo de escena del modelo articulado de la figura 4.23, teniendo en cuenta que hay dos bisagras en el modelo (entre A y B y entre C y D) y que la pieza C puede desplazarse dentro de B. Usa como única primitiva un paralelepípedo de dimensiones $(1, 10, 1)$ con un vértice en el origen de coordenadas.
12. Diseña el grafo de escena de la escalera de la figura 4.24, teniendo en cuenta que hay una bisagra entre los tramos A y D, que permite plegar A, y que los tramos B y C se pueden desplazar. Los cuatro tramos son iguales con la estructura mostrada en la parte derecha de la figura. Usa como única primitiva un cubo de lado 1 con el centro de su cara inferior en el origen de coordenadas.
13. ¿Como habría que modificar el modelo del ejercicio 12 para que las piezas D y A se cierren de forma simétrica (es decir que la parte inferior de ambas este a la misma altura).
14. Diseña el grafo de escena del modelo articulado que aparece a la derecha de la figura 4.25 formada por tres prismas triangulares unidos por dos bisagras colocadas en catetos alternos del prisma central usando como primitiva el prisma de la izquierda.
15. Programa los modelos de los ejercicios 5, 6, 10, 11 y 12 usando OpenGL.
16. Adapta los grafos y crea los modelos de los ejercicios 5, 6, 10, 11 y 12 Unity 3D.

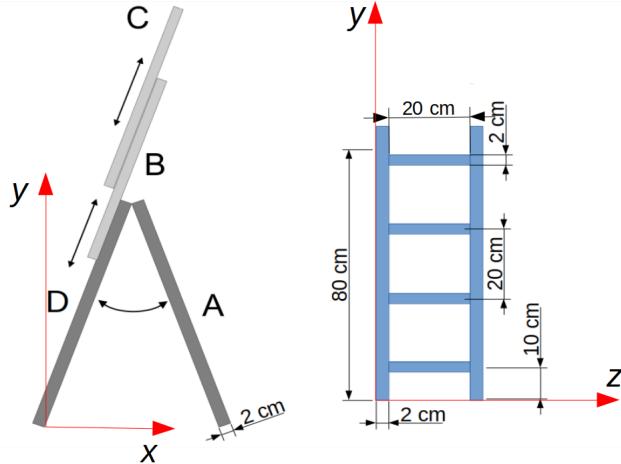


Figura 4.24: Escalera del ejercicio 12.

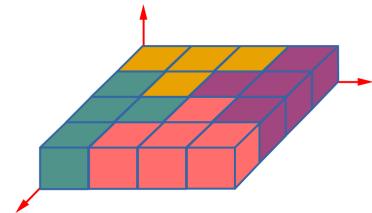


Figura 4.21: Modelo generado usando como componente el modelo de la figura 4.20 (ejercicio 7).

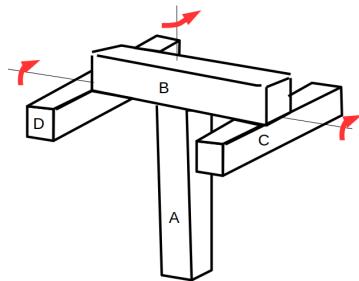


Figura 4.22: Modelo articulado del ejercicio 10.

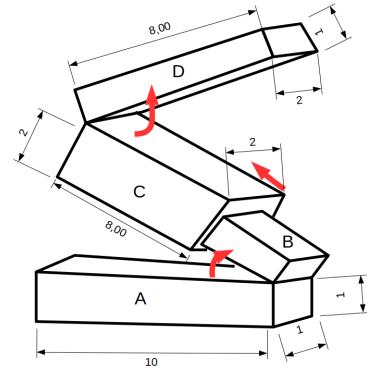


Figura 4.23: Modelo articulado del ejercicio 11.

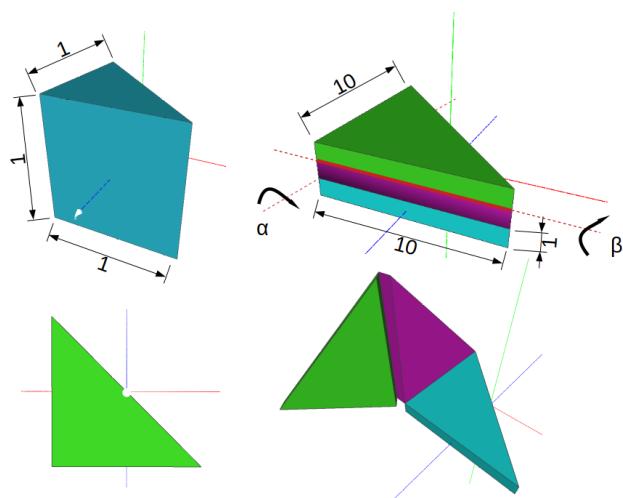


Figura 4.25: Componentes (izquierda) y modelo (derecha) del ejercicio 14.

Informática Gráfica

Juan Carlos Torres

Curso 2024/25

Dpto. Lenguajes y Sistemas Informáticos
Universidad de Granada

Disclaimer

You can edit this page to suit your needs. For instance, here we have a no copyright statement, a colophon and some other information. This page is based on the corresponding page of Ken Arroyo Ohori's thesis, with minimal changes.

CC BY-NC-SA

 This book is released into the public domain using the CC BY-NC-SA. This license enables reusers to distribute, remix, adapt, and build upon the material in any medium or format for noncommercial purposes only, and only so long as attribution is given to the creator. If you remix, adapt, or build upon the material, you must license the modified material under identical terms.

To view a copy of the CC BY-NC-SA code, visit:

<https://creativecommons.org/licenses/by-nc-sa/4.0/>

Colophon

This document was typeset with the help of KOMA-Script and L^AT_EX using the kaobook class.

5

Iluminación

La luz es la radiación electromagnética que puede ser percibida por el ojo humano. De todo el espectro electromagnético el ojo solo es capaz de ver el intervalo comprendido entre los 380 nm y los 780 nm (Figura 5.1). Físicamente la luz es compleja, su comportamiento no se puede explicar totalmente por su naturaleza ondulatoria. Aunque la mayor parte de sus propiedades se corresponden con el comportamiento de una onda:

1. Difracción: La capacidad de la luz de doblar alrededor de obstáculos o de dispersarse al pasar por una pequeña abertura. Esto es similar a lo que ocurre con las ondas en el agua.
2. Interferencia: Cuando dos o más ondas de luz se superponen, pueden reforzarse mutuamente (interferencia constructiva) o cancelarse (interferencia destructiva). Esto da lugar a patrones de franjas claras y oscuras, como los que se observan en las películas delgadas de jabón.
3. Polarización: La luz vibra en todas las direcciones perpendiculares a su dirección de propagación. Sin embargo, es posible filtrar la luz seleccionando sola la que vibra en una dirección específica, creando luz polarizada.

Pero hay comportamientos de la luz que solo se pueden explicar si la consideramos un flujo de fotones:

1. Efecto fotoeléctrico: Cuando la luz incide sobre ciertos materiales, puede liberar electrones. Este fenómeno no puede explicarse con la teoría ondulatoria clásica, pero se entiende perfectamente si consideramos que la luz está compuesta por fotones, partículas de energía que pueden ceder su energía a los electrones.
2. Cuantización: La energía de la luz está cuantizada, es decir, solo puede tomar ciertos valores discretos. Esto se explica si consideramos que la luz está compuesta por fotones, cada uno de los cuales tiene una energía determinada.

Nos interesan dos propiedades de la luz: su intensidad y su color. La intensidad está relacionada con la energía que transporta (con el flujo de fotones). El color está relacionado con la longitud de onda. Vemos el color violeta en un extremo del espectro (40nm) y el rojo en el otro (750 nm) (Figura 5.1).

En nuestras retinas hay dos tipos de células sensibles: conos y bastones. Los bastones no son sensibles a la longitud de onda de la luz, pero tienen una sensibilidad mayor en condiciones de baja iluminación, son los que nos permiten ver cuando hay poca luz. Los conos son sensibles a la longitud de onda, pero necesitan mas intensidad de luz para excitarse que los bastones. Hay tres tipos de conos, cada uno con una curva de sensibilidad distinta: centrada en el azul, en el verde y en el rojo (Figura 5.2).

Nuestro cerebro percibe el color componiendo las señales recibidas por los tres tipos de conos. Esto nos permite generar todas las sensaciones de color que podemos percibir combinando tres fuentes de luz monocromáticas

| | | |
|-------|---|----|
| 5.1 | Interacción luz/materia | 37 |
| 5.2 | Modelo de iluminación local | 38 |
| 5.2.1 | Luces | 38 |
| 5.2.2 | Reflexión y propiedades de los materiales | 38 |
| 5.3 | Sombreado | 40 |
| 5.4 | Iluminación en OpenGL | 41 |
| 5.5 | Iluminación en Unity | 42 |
| 5.6 | GLSL | 43 |
| 5.7 | Sombras y transparencias | 45 |
| 5.7.1 | Materiales Transparentes | 45 |
| 5.8 | Ejercicios | 46 |

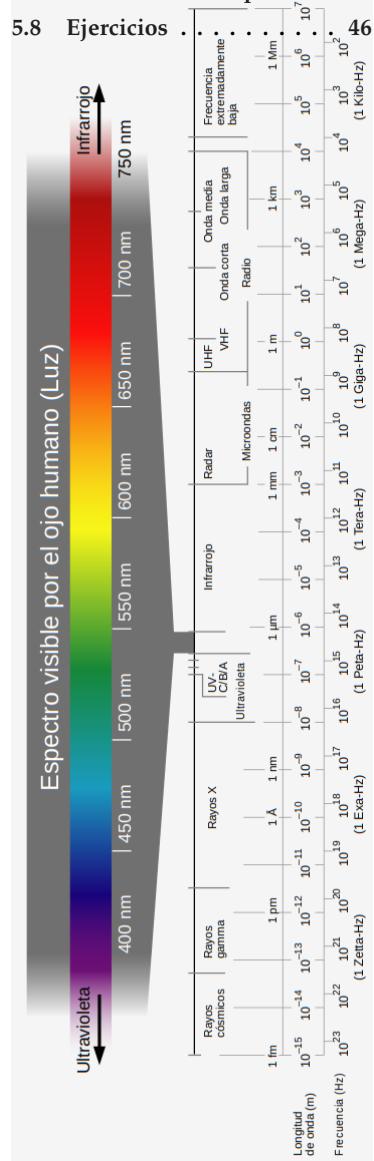


Figura 5.1: Espectro electromagnético <https://es.wikipedia.org/wiki/Luz>.

y nos permite representar los colores como una terna de intensidades de rojo, verde y azul. Estas intensidades se da en el intervalo (0,1) para que la representación sea independiente del dispositivo. Este modelo da lugar a un espacio para la descripción de los colores como un cubo en el espacio 3D.

Hay otros espacios de representación de colores. El espacio de color CIE fue creado por la Comisión Internacional de la Iluminación (CIE, por sus siglas en inglés) y se basa en una serie de experimentos que midieron cómo el ojo humano percibe los colores. Este espacio de color utiliza tres coordenadas (X, Y, Z) para representar cualquier color, y se ha convertido en un estándar internacional en la industria de la impresión, el diseño gráfico y la fotografía, ya que proporciona una forma precisa y objetiva de comunicar y gestionar el color. La coordenada Y representa la luminancia o brillo del color. Un valor de Y más alto indica un color más brillante, mientras que un valor más bajo indica un color más oscuro. Las coordenadas X y Z determinan la cromaticidad del color, es decir, el tono y la saturación. La combinación de X y Z es un diagrama de cromaticidad, que es una representación bidimensional de todos los colores visibles (figura 5.3). En este diagrama, los colores espectrales (arcoíris) se encuentran en el borde, mientras que el centro representa los colores acromáticos (blanco, gris y negro).

Los colores que se pueden generar a partir de tres primarios son los que se encuentran en el triángulo que delimitan en el diagrama cromático.

5.1. Interacción luz/materia

La luz nos permite ver nuestro entorno gracias a su interacción con los objetos. Cuando la luz incide en un objeto se puede producir los siguientes efectos:

1. Reflexión: La luz que incide en la superficie es reflejada por esta.
2. Absorción: La luz es transformada en calor en el objeto.
3. Transmisión: La luz penetra en el objeto. Esto ocurre cuando el objeto es total o parcialmente transparente.

Estos procesos no son excluyentes, de hecho lo normal es que se produzcan varios simultáneamente. Por ejemplo, cuando incide luz blanca sobre un objeto verde se refleja la radiación en la longitud de onda del verde y se absorbe el resto. En un objeto translúcido parte de la luz se transmite y parte se refleja.

La radiación que se transmite se refracta, esto es sufre un cambio de dirección que depende del índice de refracción de los dos medios. Además al propagarse dentro de un medio puede sufrir dispersión (scattering) si el medio es participativo, como el humo o la niebla.

Cuando la luz se refleja puede hacerlo en todas direcciones (reflexión difusa), o en la dirección simétrica a la dirección de incidencia (reflexión especular). Encontramos reflexión difusa en las superficies mate (como una pared) y reflexiones especulares en superficies metalizadas. En la figura 5.4 podemos apreciar ambos comportamientos además de transmisión con dispersión.

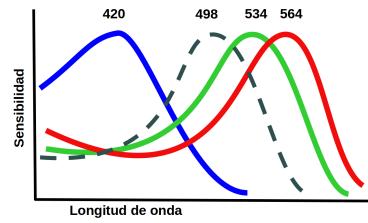


Figura 5.2: Curvas de sensibilidad espectral de los conos y los bastones (en gris discontinuo).

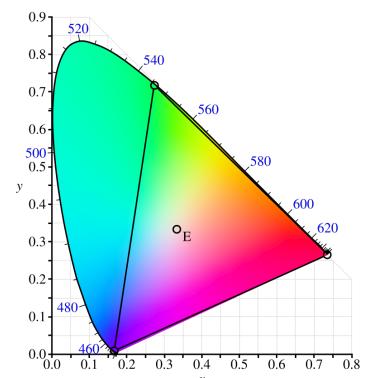


Figura 5.3: Diagrama de cromaticidad del espacio de color CIE (By BenRG, https://en.wikipedia.org/wiki/CIE_1931_color_space). Los colores que se pueden generar a partir de tres primarios son los que se encuentran dentro del triángulo formado por ellos.



Figura 5.4: Escultura de la diosa Aurora realizada por Denys Puech en 1900. Museo Orsay.

5.2. Modelo de iluminación local

Para generar imágenes es tenemos que calcular el color con el que debe verse cada punto de la escena, para ello es necesario disponer de un modelo matemático que nos permita estimar la interacción de la luz con la superficie de los objetos. A este modelo se le llama **modelo de iluminación**.

Vamos a ver un modelo de iluminación simple. Comenzaremos viendo como representar las luces y los materiales.

5.2.1. Luces

Una luz es una fuente de radiación electromagnética en el rango visible. Se caracteriza por su geometría, posición, intensidad en cada longitud de onda (espectro de emisión) y la direcciones en las que emite.

En el modelo mas simple consideramos que las luces son puntuales y que pueden estar colocadas en un punto de la escena, en cuyo caso su posición estará dada por las coordenadas de ese punto, o en infinito, simulando una fuente de luz lejana (como el sol). Las luces posicionales emiten luz en todas direcciones, las que están en el infinito (direccionales) solo en una dirección, con haces de luz (ver figura 5.5). Las luces direccionales se representan con coordenadas homogéneas que tienen valor 0 en la cuarta coordenada.

La forma mas simple de representar el espectro de emisión es como un color RGB, aunque esto implica una simplificación tanto de la distribución de frecuencias como de la intensidad, las limitaciones prácticas no son importantes salvo que el objetivo sea calcular la energía recibida en lugar de generar una imagen. Utilizar colores para representar la intensidad de la luz implica que solo podamos calcular el color resultante y la intensidad relativa respecto a otros puntos de la escena.

5.2.2. Reflexión y propiedades de los materiales

En el modelo de iluminación simple vamos a considerar únicamente materiales opacos, por tanto solo formalizaremos la reflexión. Aunque la reflexión en una superficie real puede ser compleja¹ en el modelo mas simple se representa el comportamiento como una combinación de reflexión difusa y especular pura.

Reflexión difusa

La reflexión difusa ocurre cuando la luz incide sobre una superficie rugosa o mate y se dispersa en múltiples direcciones distribuyendo la luz de manera uniforme en todas las direcciones 5.6, un objeto iluminado por una fuente de luz parecerá tener la misma intensidad de brillo sin importar desde dónde se observe.

Lla reflexión difusa se usa para simular el comportamiento de materiales como la madera, el yeso o el papel, que no tienen brillos especulares. La cantidad de luz reflejada depende del color del material, de la intensidad

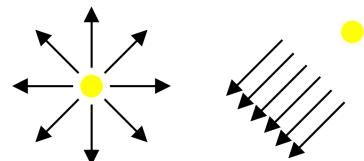


Figura 5.5: Luz posicional (izquierda) y direccional (derecha).

1: Para modelar la reflexión física en una superficie se usan funciones que indican para cada dirección de luz incidente la distribución de direcciones reflejadas (Bi-directional reflectance distribution function, BRDF).

de la luz incidente y de la orientación de la superficie respecto a la luz incidente.

Para modelar la reflexión difusa, utilizamos el modelo de Lambert, en el que la cantidad de luz reflejada es proporcional al coseno del ángulo entre la dirección de la luz incidente y la normal de la superficie. La intensidad de luz incidente por unidad de superficie depende de la orientación de esta respecto a la dirección de la luz incidente (ver figura 5.7).

En términos matemáticos, la intensidad de la luz reflejada (F_d) se puede expresar como:

$$F_d = I_i \cdot k_d \cdot \max((\mathbf{L} \cdot \mathbf{N}))$$

donde:

I_i es la intensidad de la luz incidente.

k_d es el coeficiente de reflexión difusa del material.

\mathbf{L} es el vector de la luz incidente.

\mathbf{N} es el vector normal de la superficie.

Este modelo simplificado permite calcular el color resultante en cada punto de la superficie, considerando únicamente la reflexión difusa. Es especialmente útil en aplicaciones donde se busca un equilibrio entre realismo y eficiencia computacional.

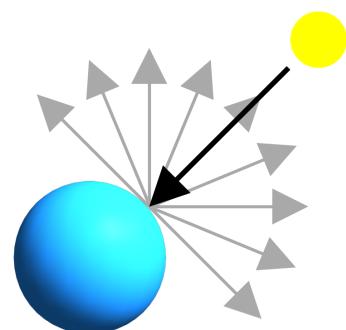


Figura 5.6: Reflexión difusa. Luz incidente en negro, reflejada en gris.

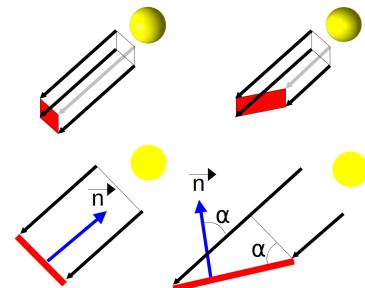


Figura 5.7: La intensidad de luz que incide en una superficie depende de su orientación respecto a la luz. El flujo en la configuración de derecha e izquierda es el mismo, pero la intensidad en cada unidad de superficie es máxima cuando es perpendicular a la dirección de incidencia.

Reflexión especular

La reflexión especular se produce cuando la luz incide sobre una superficie lisa y brillante, como una superficie metálica, reflejándose en una dirección específica. La reflexión especular mantiene la dirección de la luz reflejada de manera coherente (figura 5.8).

Para modelar la reflexión especular utilizamos el modelo de Phong, que simula el brillo y el reflejo de las superficies pulidas. Este modelo considera que la intensidad de la luz reflejada depende del ángulo entre la dirección de la luz reflejada y la dirección del observador.

En términos matemáticos, la intensidad de la luz reflejada F_s se puede expresar como:

$$F_s = I_i \cdot k_s \cdot \max((\mathbf{R} \cdot \mathbf{V}))^e$$

donde:

I_i es la intensidad de la luz incidente.

k_s es el coeficiente de reflexión especular del material.

\mathbf{R} es el vector de la luz reflejada.

\mathbf{V} es el vector de la dirección del observador.

e es el exponente de brillo, que determina la dispersión de la luz reflejada, o lo que es lo mismo la amplitud de la zona de brillo.

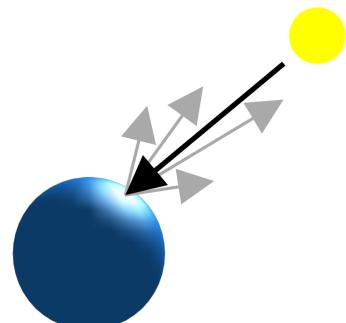


Figura 5.8: Reflexión Especular. La luz reflejada (en gris) se concentra en la dirección simétrica a la de incidencia.

Iluminación del ambiente

La componente ambiente es una parte fundamental del modelo de iluminación, representa la luz que llega a la superficie de forma indirecta (después de reflejarse en otros objetos de la escena). En el modelo de iluminación simple asumimos que ilumina todos los objetos de manera uniforme, sin importar su orientación o posición.

En términos matemáticos, la intensidad de la luz ambiente F_a se puede expresar como:

$$F_a = I_a \cdot k_a$$

donde:

I_a es la intensidad de la luz ambiente global en la escena.

k_a es el coeficiente de reflexión ambiente del material.

La componente ambiente es crucial para evitar que las áreas en sombra de los objetos aparezcan completamente negras, proporcionando así un nivel básico de iluminación que mejora el realismo de la escena.

Modelo de iluminación

En el modelo de iluminación de Phong, la luz total que incide sobre un punto de la superficie se calcula sumando las componentes ambiente, difusa y especular. La componente ambiente asegura que incluso las áreas no directamente iluminadas por una fuente de luz específica reciban algo de iluminación, contribuyendo a una representación más natural de los objetos en la escena.

Si hay mas de una fuente de luz se suman las contribuciones de todas ellas.

5.3. Sombreado

El calculo del color de los objetos de la escena (sombreado o shading) implica la aplicación del modelo de iluminación y puede realizarse a diferente nivel: caras, vértices o píxeles la figura 5.10 muestra el resultado de dibujar el mismo objeto usando cada uno de ellos.

El sombreado de caras (Flat Shading) asigna un solo color a cada cara. Este color se calcula utilizando la normal de la cara, sus reflectividades, un punto cualquiera de la cara y la fuente de luz. Este método es rápido y sencillo de implementar, pero produce bordes visibles entre las caras y color constante en las caras. Es ideal para modelos formados por objetos poliédricos donde la velocidad es crucial.

El sombreado de vértices (Gouraud Shading) calcula el color en cada vértice del polígono utilizando las normales y posiciones de cada vértice y luego interpola estos colores a lo largo de las caras. Esto suaviza las transiciones entre caras, eliminando los bordes visibles. Sin embargo, puede no capturar correctamente los reflejos especulares, ya que estos pueden quedar suavizados. Este método es comúnmente utilizado en

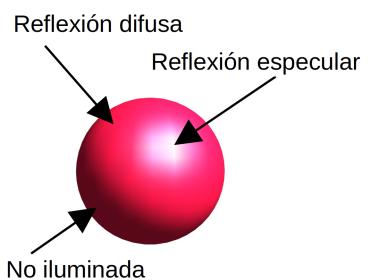


Figura 5.9: La zona no iluminada de la esfera no se ve negra por la iluminación ambiente.

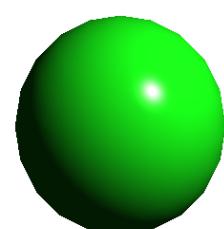
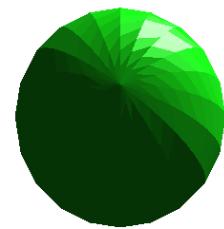


Figura 5.10: Métodos de sombreado. Arriba Flat, Gouraud en el centro, Phong Abajo.

videojuegos y aplicaciones interactivas donde se requiere un balance entre calidad y rendimiento.

El sombreado de píxeles (Phong Shading) calcula el color en cada píxel de la superficie. Determina la normal en el pixel interpolando las normales de los vértices y aplica el modelo de iluminación en cada píxel. Este método produce resultados muy realistas, con reflejos especulares precisos, pero es más costoso computacionalmente. Se utiliza en renderizados de alta calidad donde el realismo es prioritario, como en películas y visualizaciones arquitectónicas, implementándose en GPU.

5.4. Iluminación en OpenGL

OpenGL como parte de la funcionalidad fija (pre-programada), incluye una implementación de un modelo de iluminación simple, que tiene en cuenta la iluminación directa de las fuentes de luz, calculando: reflexión difusa, reflexión especular, iluminación ambiente y emisividad (luz emitida por la superficie del objeto). Para que se calcule la iluminación es necesario activarlo²:

```
glEnable(GL_LIGHTING);
```

Los colores, materiales e intensidades se representan en *rgb* o *rgba* con valores normalizados entre 0 y 1. Cuando hay mas de una fuente de luz o reflectividades altas es posible que el color resultante sea mayor que uno, en este caso las componentes mayores que uno se truncan generando una imagen sobre-expuesta (la imagen inferior de la figura 5.11 esta sobre-expuesta).

Las implementaciones de OpenGL están obligadas a gestionar un número mínimo de 8 fuentes de luz. Cada una de ellas se referencia con un identificador **GL_LIGHT0**, **GL_LIGHT1**, ..., **GL_LIGHT7**. Cada una de estas fuentes de luz puede encenderse y apagarse de forma individual, con:

```
glEnable(GL_LIGHTn) ; // enciende la enesima fuente de luz
glDisable(GL_LIGHTn) ; // apaga la enesima fuente de luz
```

De cada fuente de luz se puede especificar su posición y su intensidad. La posición se da con la función **glLightfv**:

```
GLfloat pos[4] = { 20.0, 10.0, 20.0, 0.0 };
glLightfv(GL_LIGHTn, GL_POSITION, pos);
```

a la que se le indica la luz que se está modificando y un vector 4D con su posición en coordenadas homogéneas. Si la coordenada homogénea (*w*) es cero, la fuente está en el infinito, y la luz incide en la dirección del vector (*x*, *y*, *z*). En caso contrario es una luz posicional.

Las luces están integradas en el grafo de escena. Es decir, les afectan las transformaciones geométricas activas cuando se ubican en la escena usando la función **glLightfv**.

A cada fuente de luz OpenGL le asocia tres intensidades que indican como contribuye en el cálculo de la iluminación difusa, especular y ambiente³. Cada una de estas intensidades se representa como un color:

2: Con la iluminación desactivada, el color de las primitivas dibujadas depende de una terna RGB del estado interno, que se modifica con **glColor**.



Figura 5.11: Dos imágenes del mismo modelo. La inferior está sobre-expuesta.

3: También se pueden especificar focos, que son luces posicionales que iluminan en una dirección.

```
float color[4] = { 0.8, 0.0, 1, 1 };
glLightfv(GL_LIGHTn, GL_AMBIENT, color);
glLightfv(GL_LIGHTn, GL_DIFFUSE, color);
glLightfv(GL_LIGHTn, GL_SPECULAR, color);
```

Al iniciar OpenGL solo está inicializada y encendida la luz **GL_LIGHT0**.

Materiales en OpenGL

OpenGL asocia materiales a los objetos. Cada material se caracteriza por un conjunto de reflectividades: ambiente, difusa y especular⁴. Los objetos pueden tener dos materiales: uno para las caras delanteras y otro para las caras traseras⁵. Para asignar las reflectividades se usa la función **glMaterialfv**, a la que se le indica las caras a las que se va aplicar (**GL_FRONT**, **GL_BACK** o **GL_FRONT_AND_BACK**), el parámetro a fijar (**GL_AMBIENT**, **GL_DIFFUSE**, **GL_SPECULAR**) y un color. Para indicar el exponente de brillo se usa la función **glMaterialf**, indicando la cara, el parámetro **GL_SHININESS** y el valor del exponente). El siguiente código asigna un material verde con brillos blancos:

```
1 GLfloat rd[4]={0.2,1.0,0.2,1.0}, re ={0.6,0.6,0.6,1.0},
2      ra ={0.0,0.2,0.0,1.0};
3 glMaterialfv( GL_FRONT, GL_AMBIENT, ra ) ;
4 glMaterialfv( GL_FRONT, GL_DIFFUSE, rd ) ;
5 glMaterialfv( GL_FRONT, GL_SPECULAR, re ) ;
6 glMaterialf( GL_FRONT, GL_SHININESS, 5 ) ;
```

Los materiales se almacenan en el estado de OpenGL y se aplica a todos los objetos que se dibujen después de asignarlos.

4: También es posible asignar emisividad a los materiales

5: Este solo se usa si no se están eliminando las caras traseras

Sombreado en OpenGL

OpenGL permite, en la funcionalidad fija, realizar el sombreado plano y de Gouraud (por vértices). La función **glShadeModel** permite seleccionar el método de sombreado activo en cada momento (que afectará a todas las primitivas que se dibujen posteriormente). Admite dos valores como argumento **GL_FLAT** y **GL_SMOOTH**.

5.5. Iluminación en Unity

Unity3D utiliza un sistema de iluminación flexible y potente que permite simular diferentes tipos de luces y materiales en una escena. A continuación, se describen los conceptos básicos de luces y materiales en Unity3D.

Luces en Unity3D

Unity3D soporta varios tipos de luces, añadiendo a los tipos existentes en OpenGL luces de área, que emiten luz desde una superficie rectangular. Las luces se crean como objetos en el grafo de escena y están afectadas por las transformaciones de sus nodos padre. Se pueden crear desde la interfaz o desde el código. En la interfaz añadir como *GameObject > Light*.

Fijar parámetros en *Inspector*. La intensidad es independiente del color y se da normalizada.

Para crear y configurar una luz en Unity3D, se puede utilizar el siguiente código en C#:

```
Light myLight = gameObject.AddComponent<Light>();
myLight.type = LightType.Point;
myLight.color = Color.white;
myLight.intensity = 1.0f;
myLight.range = 10.0f;
```

Materiales en Unity3D

Los materiales en Unity3D se crean y configuran utilizando el sistema de shaders de Unity. El color difuso del material es su *Albedo*, la reflectividad especular se controla con dos parámetros *Metalicidad*, que controla cuán metálico es el material y *Suavidad*, que controla si el objeto es pulido o rugoso. Los materiales se crean como *Assets* y sus parámetros se editan en *Inspector*.

También se puede asignar un material a un objeto desde el código del programa:

```
Renderer renderer = gameObject.GetComponent<Renderer>();
Material material = new Material(Shader.Find("Standard"));
material.color = Color.green;
material.SetFloat("_Metallic", 0.5f);
material.SetFloat("_Glossiness", 0.5f);
renderer.material = material;
```

Sombreado en Unity3D

Para cambiar el método de sombreado en Unity3D se debe cambiar el shader (el programa en GPU que dibuja la superficie). Cada material tiene asociado un shader. Si se quiere forzar que en una malla se aparezcan discontinuidades en determinadas aristas independientemente del shader con el que se dibuja se deben duplicar los vértices.

5.6. GLSL

Las versiones recientes de OpenGL no implementa un modelo de iluminación. El cálculo de iluminación debe programarse en shaders. Los shaders son programas que se ejecutan en la GPU para generar las imágenes. Un shader está compuesto de dos programas: un programa para procesar vértices (*vertex shader*) y un programa para procesar la imagen a nivel de pixel (*fragment shader*). Las primitivas llegan al *vertex shader*. La salida de este se rasteriza y se procesa en el *fragment shader*.

GLSL (OpenGL Shading Language) es el lenguaje de programación utilizado para escribir shaders en OpenGL. Los shaders se cargan y compilan en tiempo de ejecución del programa. Abordaremos su estudio más adelante, de momento para clarificar el concepto se muestra el código

de un programa simple que dibuja una esfera usando sombreado de Phong:

```

1 // Variables para los materiales
2 GLfloat mat_diffuse[] = { 0.6f, 0.6f, 0.6f, 1.0f }, GLfloat
   mat_specular[] = { 1.0f, 1.0f, 1.0f, 1.0f }, GLfloat
   mat_shininess[] = { 50.0f };
3 GLuint phongShaderProgram; // Shader program para Phong (con GLSL)
4
5 void loadShaders() {      // Carga los shaders Phong
6   const char* vertexShaderSource = "#version 120\n"
7   "varying vec3 normal, lightDir, viewDir;" 
8   "void main() {"
9     "    vec4 vertexPos = gl_ModelViewMatrix * gl_Vertex;" 
10    "    lightDir = vec3(gl_LightSource[0].position - vertexPos);"
11    "    viewDir = vec3(-vertexPos.xyz);"
12    "    normal = gl_NormalMatrix * gl_Normal;" 
13    "    gl_Position = ftransform();"
14  "}";
15
16  const char* fragmentShaderSource = "#version 120\n"
17  "varying vec3 normal, lightDir, viewDir;" 
18  "void main() {"
19    "    vec3 N = normalize(normal);"
20    "    vec3 L = normalize(lightDir);"
21    "    vec3 V = normalize(viewDir);"
22    "    vec3 R = reflect(-L, N);"
23    "    float lambertTerm = max(dot(N, L), 0.0);"
24    "    vec4 diffuse = lambertTerm * gl_FrontMaterial.diffuse;" 
25    "    vec4 ambient = 0.1*gl_FrontMaterial.ambient;" 
26    "    vec4 specular = vec4(0.0);"
27    "    if (lambertTerm > 0.0) {"
28      "        float spec = pow(max(dot(R, V), 0.0),
29      gl_FrontMaterial.shininess);"
30      "        specular = spec * gl_FrontMaterial.specular;" 
31      "    }"
32    "    gl_FragColor = ambient + diffuse + specular;" 
33  "}";
34
35 GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
36 GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
37 glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
38 glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);
39
40 glCompileShader(vertexShader);
41 glCompileShader(fragmentShader);
42
43 phongShaderProgram = glCreateProgram();
44 glAttachShader(phongShaderProgram, vertexShader);
45 glAttachShader(phongShaderProgram, fragmentShader);
46
47 glLinkProgram(phongShaderProgram);
48 }
49
50 // Dibuja una esfera con sombreado Phong usando shaders GLSL
51 void drawPhongShadedSphere() {
52   glUseProgram(phongShaderProgram);
53   glutSolidSphere(3.0, 20, 20);
54   glUseProgram(0); // Desactiva los shaders

```

54 | }

5.7. Sombras y transparencias

Las sombras y la representación de materiales transparentes permiten aumentar el realismo visual en las escenas renderizadas.

Cálculo de Sombras

El *shadow mapping* es una técnica de generación de sombras simple basada en renderizar la escena desde la perspectiva de la luz, crea una textura de profundidad que almacena las distancias desde la luz a los objetos. A continuación renderiza la escena desde la perspectiva de la cámara. Para cada fragmento calcula su distancia a la luz y la compara con el valor almacenado en la textura de profundidad para determinar si el fragmento está en sombra (figura 5.13). Para usar este algoritmo en OpenGL se debe implementar en el shader. Unity 3D incorpora la generación de sombras utilizando este método.

5.7.1. Materiales Transparentes

La representación de materiales transparentes se realiza mediante el uso de la cuarta componente del color (α). Un valor 1 de α indica que el material es opaco, un valor cero corresponde a una superficie totalmente transparente. La figura ?? muestra una escena con dos superficies transparentes.

OpenGL procesa la transparencia usando el modo de mezcla (*blending*) que permite combinar el color del fragmento que se va a dibujar con el color ya dibujado en el pixel. Por tanto, es necesario dibujar antes los objetos opacos que los transparentes. El siguiente código muestra un esquema del proceso.

```

1 ...
2 glEnable(GL_BLEND);
3 ...
4 void dibuja()
5 ...
6 //Dibujar objetos opacos
7 ...
8 ...
9 ...
10 // Dibujar objetos transparentes
11 ...
12 ...
13 ...
14 ...
15 ...
16

```

La función `glBlendFunc` indica que ecuación de combinación se debe utilizar. En este caso se indica:

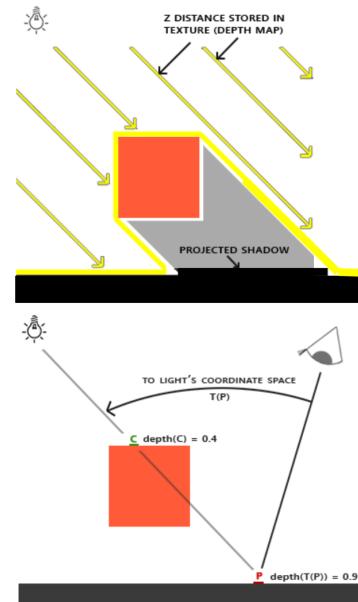


Figura 5.12: Métodos de sombreado. Arriba Flat, Gouraud en el centro, Phong Abajo.

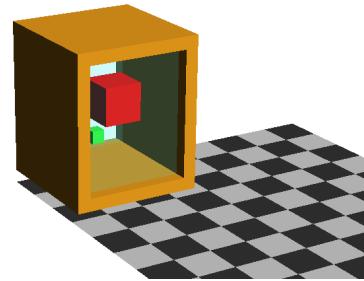


Figura 5.13: Escena con dos cristales semitransparentes.

$$color = Source_{\alpha}Source_{Color} + (1 - Source_{\alpha})Destination_{Color}$$

Donde *Source* es el fragmento a dibujar, y *Destination* es el contenido actual de Frame buffer

Para crear materiales transparentes en Unity3D se debe cambiar el *renderingmode* del material a *transparente* y asignar α en el Albedo.

5.8. Ejercicios

1. ¿Todos los monitores reproducen el mismo color para cada terna RGB?
2. ¿Se puede hacer que reproduzcan los mismos colores?
3. ¿Porque no vemos colores cuando hay poca luz?
4. ¿Que implicaciones tiene que se represente la intensidad de las fuentes de luz con valores RGB normalizados?
5. ¿Que pasa si se añaden muchas fuentes de luz a una escena?
6. ¿Como se puede colocar una luz posicional en el extremo del modelo articulado del ejercicio 5.10 de forma que se mueva con el modelo.
7. En el ejercicio anterior se quiere añadir una superficie plana debajo del modelo de forma que se aprecie el cambio de iluminación en el suelo al mover objeto. ¿Como se debería hacer?.
8. La imagen de la figura 5.14 corresponde a la visualización de un poliedro regular de radio 1, centrado en el origen, que aproxima una esfera.¿Que fuentes de luz y que propiedades de material se han usado para generarla?
9. ¿Que cambios se deben hacer en el código del ejercicio anterior para que la visualización se parezca a una esfera?
10. Identifica donde está colocada la fuente de luz en el código base usado en prácticas.

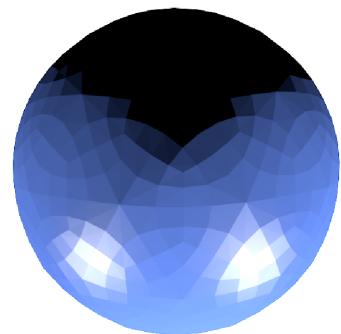


Figura 5.14: Imagen del ejercicio 10.

Bibliografía

- [Bot+10] Mario Botsch et al. *Polygon mesh processing*. CRC press, 2010.
- [Cig+08] Paolo Cignoni et al. «MeshLab: an Open-Source Mesh Processing Tool». En: *Eurographics Italian Chapter Conference*. Ed. por Vittorio Scarano, Rosario De Chiara y Ugo Erra. The Eurographics Association, 2008. ISBN: 978-3-905673-68-5. doi: [10.2312/LocalChapterEvents/ItalChap/ItalianChapConf2008/129-136](https://doi.org/10.2312/LocalChapterEvents/ItalChap/ItalianChapConf2008/129-136).
- [Kil96] Mark J Kilgard. *The OpenGL utility toolkit (GLUT) programming interface API version 3*. [accedido 26-Abril-2024]. 1996. URL: <https://www.opengl.org/resources/libraries/glut/glut-3.spec.pdf>.
- [KSS16] John Kessenich, Graham Sellers y Dave Shreiner. *OpenGL Programming Guide: The official guide to learning OpenGL, version 4.5 with SPIR-V*. [accedido 26-Abril-2024]. Addison-Wesley Professional, 2016. URL: <https://www.cs.utexas.edu/users/fussell/courses/cs354/handouts/Addison.Wesley.OpenGL.Programming.Guide.8th.Edition.Mar.2013.ISBN.0321773039.pdf>.
- [Shr+07] Dave Shreiner et al. *OpenGL (R) programming guide: The official guide to learning OpenGL (R), version 2.1*. [accedido 26-Abril-2024]. Addison-Wesley Professional, 2007. URL: <https://theswissbay.ch/pdf/Gentoomen%20Library/Game%20Development/Programming/The%20Official%20Guide%20to%20Learning%20OpenGL%20Version%202.1.pdf>.

Informática Gráfica

Juan Carlos Torres

Curso 2024/25

Dpto. Lenguajes y Sistemas Informáticos
Universidad de Granada

Disclaimer

You can edit this page to suit your needs. For instance, here we have a no copyright statement, a colophon and some other information. This page is based on the corresponding page of Ken Arroyo Ohori's thesis, with minimal changes.

CC BY-NC-SA

 This book is released into the public domain using the CC BY-NC-SA. This license enables reusers to distribute, remix, adapt, and build upon the material in any medium or format for noncommercial purposes only, and only so long as attribution is given to the creator. If you remix, adapt, or build upon the material, you must license the modified material under identical terms.

To view a copy of the CC BY-NC-SA code, visit:

<https://creativecommons.org/licenses/by-nc-sa/4.0/>

Colophon

This document was typeset with the help of KOMA-Script and L^AT_EX using the kaobook class.

6

Texturas

Hasta aquí hemos creado modelos que tienen colores uniformes, pero los objetos reales no suelen tener un color constante en toda su superficie, bien por cambios en la estructura del objeto, como las vetas de la madera, o bien por imperfecciones o acumulación de polvo.

La figura 6.1 muestra dos imágenes del mismo cubo, arriba dibujado sin aplicar textura y abajo aplicando una textura de madera.

Una textura es una imagen aplicada a la superficie del objeto para modificar su color.

Aplicar una textura a un polígono es como pegarle un vinilo (ver figura 6.2 <http://cf.ycdn.net/>). El nivel de detalle resultante dependerá de la resolución de la imagen usada como textura.

En la imagen de textura habrá normalmente zonas que no se usen. Por ejemplo en la textura de la imagen 6.2 solo se está usando una zona circular de la imagen.

Cuando se aplica a una malla completa podemos pensar en las texturas como recortables que montamos sobre la superficie del objeto. Por ejemplo la plantilla de la figura 6.3 se puede utilizar para aplicar color a las caras de un dado, como se muestra en la figura 6.4. Se puede conseguir el mismo resultado con diferentes imágenes, por ejemplo en la figura 6.3 se podría colocar el cuadrado del dos sobre el seis, o rotar la imagen.

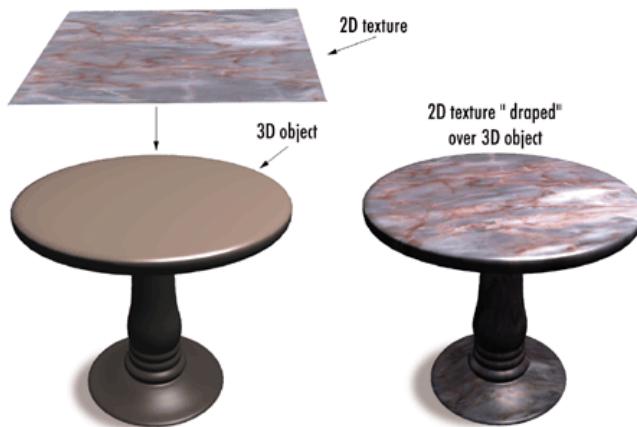


Figura 6.2: Aplicación de textura a una superficie.

| | |
|----------------------------------|----|
| 6.1 Texturas | 49 |
| 6.2 Parametrización | 50 |
| 6.3 Texturas en OpenGL | 52 |
| 6.4 Texturas en Unity | 54 |
| 6.5 Bump Mapping | 54 |
| 6.6 Ejercicios | 55 |

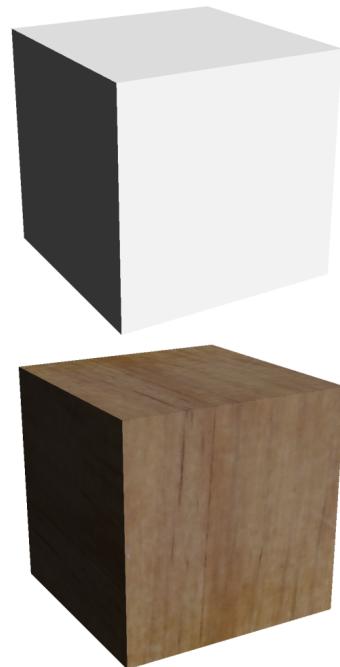


Figura 6.1: Dos imágenes del mismo cubo, sin textura arriba y con textura de madera abajo

6.1. Texturas

Para poder dibujar un objeto con una textura necesitamos, además de la imagen de la textura indicar como se aplica esta al objeto, es decir que punto de la imagen se dibuja en cada vértice de la malla. La figura 6.5 muestra este proceso. En el triángulo destacado en la imagen cada vértice tiene asociada una posición de la imagen, de forma que se le hace corresponder al triángulo del modelo un triángulo en la textura. Las posiciones de la imagen asociadas a los vértices son sus **coordenadas de textura** (u, v).

Las coordenadas de textura están normalizadas, tomando valores entre 0 y 1 en la textura. Es decir las dos esquinas de la diagonal principal tienen coordenadas (0,0) y (1,1). De esta forma las coordenadas son independientes del tamaño de la imagen.

Se pueden utilizar coordenadas de textura (t) mayores que 1 o menores que 0. Dependiendo de la configuración el efecto puede ser tomar el color del borde de la imagen ($\max(\min(t, 1), 0)$) o repetir la textura ($\text{frac}(t)$). Esta última configuración permite una superficie grande con una imagen de textura que se repite.

Normalmente no es posible asignar coordenadas de textura que permitan aprovechar toda la imagen, apareciendo zonas de la imagen no usadas. La figura 6.6 muestra las coordenadas de textura de una tetera, pueden observarse zonas en las que no hay ningún triángulo asociado y por tanto no se usarán. Lógicamente, interesa que el espacio perdido de la textura sea mínimo.

Se aprecia la existencia de agrupaciones de triángulos (se suelen llamar islas). Los vértices que están dentro de la isla tienen asociada su coordenada de textura en la imagen. Los que están en el borde de las islas pueden aparecer en mas posiciones de la imagen, y por tanto tendrán coordenadas de textura diferentes en sus triángulos. Por ejemplo, para el dado de la figura 6.3 el vértice de la esquina superior izquierda de la cara del 4 para la cara del 4, otra para la cara del 2 y otra para la del 1. En estos casos es necesario o bien subdividir la malla o bien almacenar mas de unas coordenadas de textura para cada vértice.

Al dibujar el modelo se consultan las coordenadas de textura de los vértices. Al rasterizar los triángulos se calculan las coordenadas de textura de cada fragmento interpolando las coordenadas de los vértices. Estas coordenadas se utilizan para obtener el color de textura que corresponde al fragmento, calculando el **texel**¹ que le corresponde multiplicando la coordenada (u o v) por la resolución horizontal o vertical de la imagen:

$$X_{tex} = u \cdot ResX$$

$$Y_{tex} = v \cdot ResY$$

Siendo $ResX \times ResY$ la resolución de la imagen. Los valores X_{tex} e Y_{tex} son reales que pueden coincidir con el centro de un texel. En caso contrario, que es lo habitual, podemos utilizar dos estrategias diferentes para obtener el color:

- Asignar el color del texel más cercano.

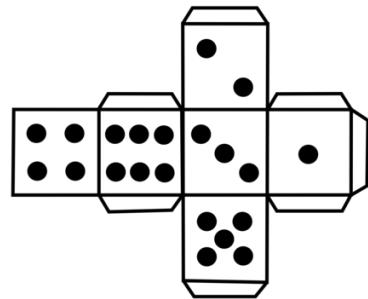


Figura 6.3: La textura para visualizar un cubo como un dado es semejante a un recortable del dado (<https://www.supercoloring.com/>)

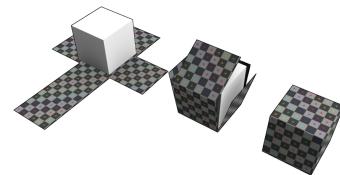


Figura 6.4: Aplicación de la textura de la figura 6.3 a un cubo.

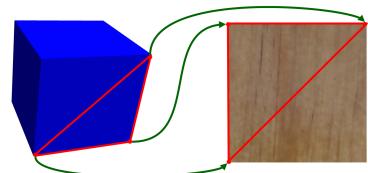


Figura 6.5: Las coordenadas de textura asocian una posición de la textura a cada vértice del polígono.

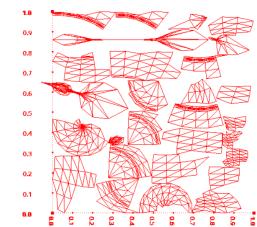


Figura 6.6: Coordenadas de textura creadas para la tetera de la parte inferior. Se muestra la zona de la imagen que se asocia a cada triángulo.

1: Se utiliza el término texel para referirse a los pixel de una textura.

- Hacer una interpolación bilineal de los cuatro texeles próximos.

La figura 6.7 muestra este proceso, en ella el punto rojo representa la posición (X_{tex}, Y_{tex}) obtenida y los signos + los centros de los texeles usados en el cálculo del color. En la parte superior se está usando solamente el texel mas cercano, en la inferior se interpola entre los cuatro texeles cercanos.

La figura 6.8 muestra el efecto de aplicar una textura de rayas a un cubo utilizando ambos métodos de filtrado.

En esta figura cada texel de la textura cubre varios pixeles de la superficie (estamos magnificado la textura). También puede ocurrir que varios texeles de la textura se encuentren en el mismo pixel de la superficie. En este último caso también podemos utilizar estas dos misma estrategias. Si varios texeles se proyectan en el mismo pixel podemos usar el color del más próximo al centro del pixel o hacer una interpolación bilineal entre todos los texeles que se encuentran el pixel.

Si el conjunto de texeles es grande el cálculo de la interpolación es costoso y puede comprometer el rendimiento de la aplicación. Esta situación ocurre cuando la textura tiene más resolución que el dibujo en pantalla de la superficie en la que se está aplicando. Para resolver este problema se pueden precalcular versiones simplificadas de la textura (mipmaps) y utilizar en cada momento la que tenga un tamaño adecuado (figura 6.9).

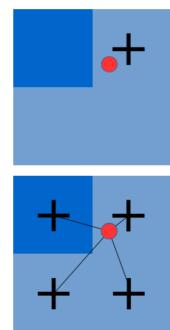


Figura 6.7: Estrategias de asignación del color, arriba usando solo el texel mas cercano, abajo interpolando entre los cuatro vecinos.

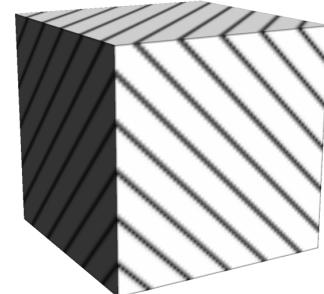
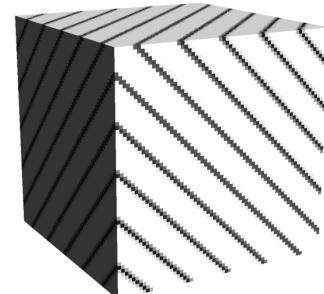


Figura 6.8: Aplicación de una textura a un cubo con ambos métodos de filtrado. Arriba el mas cercano, abajo interpolación.

6.2. Parametrización

La parametrización es el proceso de asignación de coordenadas de textura a un modelo. Cuando el modelo es pequeño y simple (p.e. un cubo) puede hacerse manualmente, pero no es posible en cualquier malla que no sea tan simple. No hay una única forma de asignar las coordenadas de textura a los vértices. la asignación depende de la textura, del modelo y del efecto que se quiera conseguir. En la figura 6.4 se esta aplicando una textura a un cubo de forma que cada cara utiliza una zona diferente de la textura, en este caso la textura sería como la mostrada en la figura 6.3. En la figura 6.5 se está aplicando una textura de madera, que se aplica completa a una cara del cubo, pudiendo aplicarse la misma textura a cada cara reutilizando las coordenadas de textura, generando un resultado como el mostrado en la figura 6.1.

Parametrización procedural

Cuando la superficie del objeto se puede describir, aunque sea de forma aproximada, con una función, se pueden calcular las coordenadas de textura a partir de las coordenadas de los vértices. por ejemplo, para aplicar un imagen de la tierra con proyección mercator como textura a una esfera (figura 6.10) podemos calcular las coordenadas de textura en coordenadas esféricas, esto es, en función de los ángulos de longitud y latitud:

$$u = (\text{longitud} + 180)/360$$

$$v = (\text{latitud} + 90)/180$$

De esta forma estamos asignando coordenadas de textura como si estuviésemos proyectando los puntos en una esfera.

En general podemos calcular las coordenadas de textura usando una función que las calcule a partir de las coordenadas de los vértices:

$$u = f_u(x, y, z)$$

$$v = f_v(x, y, z)$$

Como casos particulares, podemos calcular las texturas proyectando las coordenadas del objeto en un cilindro o en un plano.

Desenrollado

La parametrización procedural es adecuada cuando el objeto tiene una estructura semejante a la de la superficie en la que se proyecta. En caso contrario es necesario utilizar métodos de cálculo de coordenadas de textura que tengan en cuenta la geometría del objeto. Para entender este proceso podemos pensar en un procedimiento para desplegar (*unwrap*) un poliedro realizando cortes en las aristas hasta poder aplanar su superficie. Para obtener una parametrización que se corresponda con la textura de la imagen 6.3 realizaríamos cortes en las aristas que separan la cara del 1 de las del 2, 5 y 4, y en las aristas que separan las caras del 4 y 6 de las del 2 y el 5.

Este proceso se puede realizar de forma totalmente automática o asistida. En los métodos asistidos el usuario puede indicar que aristas se van a cortar (va a ser costuras en la textura). Una forma trivial de desenvolver un poliedro es generar costuras en todas las aristas, de este modo que cada triángulo se proyecta en la textura de forma independiente. Esta parametrización no suele ser aceptable.

Parametrización con MeshLab

Blender permite realizar la parametrización tanto de forma asistida como automática.

MeshLab por su parte permite realizar parametrizaciones triviales, planas, procedurales (indicando la función de proyección) y automáticas (usando un algoritmo de crecimiento de regiones de Voronoi). La parametrización de la figura 6.6 se ha generado con este método.

Los métodos de parametrización se encuentran en el grupo de filtros *Textura*.

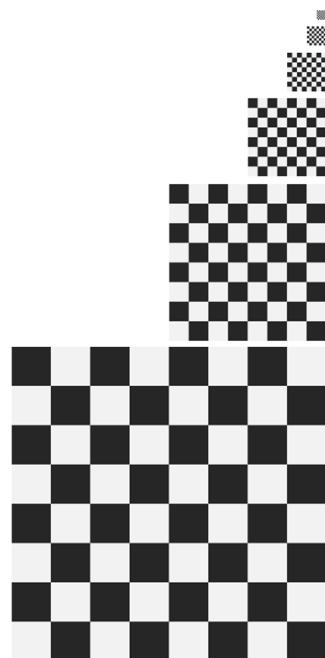


Figura 6.9: Niveles de mipmaps generados a partir de la textura inferior. El tamaño de cada imagen es la mitad de la anterior.

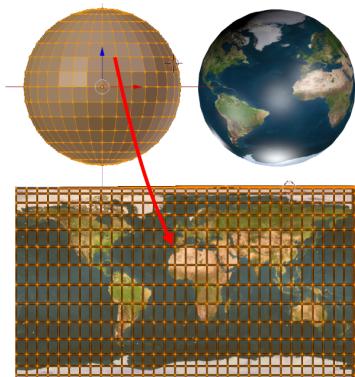


Figura 6.10: Para aplicar la imagen de abajo a una esfera se pueden calcular las coordenadas de textura en función de los ángulos de longitud y latitud.

6.3. Texturas en OpenGL

El procesamiento de texturas en OpenGL se encuentra inicialmente desactivado. Para activarlos se debe usar la orden glEnable. La orden glDisable se pueden usar para desactivarlo: toda l:

```
glEnable( GL_TEXTURE_2D ) ; // habilita texturas
glDisable( GL_TEXTURE_2D ) ; // deshabilita texturas
```

Cuando están habilitadas las texturas, se consulta la textura activa cada vez que un polígono se proyecta en un pixel, antes de calcular el color de dicho pixel, el color de la textura sustituye a las reflectividades del material (usualmente a la difusa y la ambiental)².

OpenGL puede gestionar más de una textura a la vez, aunque solo una está activa en cada momento. Para cargar una nueva textura hay que pedirle a OpenGL un identificador de textura llamando a la función glGenTextures:

```
GLuint idTex ;
glGenTextures( 1, &idTex ) ; // Crea un identificador de textura y
    lo devuelve en idTex
```

En el estado interno de OpenGL se almacena el identificador de la textura activa, que es la que se usará en cualquier operación de visualización de primitivas y en cualquier operación de configuración de la funcionalidad de texturas. Para cambiar la textura activa se usa la función glBindTexture:

```
glBindTexture( GL_TEXTURE_2D, idTex ) ; // Activa la textura 2D
    idTex
```

El primer parámetro, GL_TEXTURE_2D, indica que la textura es bidimensional (en esta lección solamente abordaremos texturas 2D pero también existen otros tipos de textura, como las 1D y las 3D).

La imagen de la textura se le debe pasar a OpenGL, podemos usar para ello la función glTexImage2D, que envía la imagen de la textura como un puntero a la cadena de bytes que contiene la imagen, organizada como una secuencia de texels. Los parámetros de la función indican como está empaquetada la imagen. Por ejemplo, la siguiente llamada

```
glTexImage2D( GL_TEXTURE_2D,
    0,           // nivel de mipmap (para imagenes multiresolucion)
    GL_RGB,     // formato interno
    ancho,      // num. de columnas (GLsizei)
    alto,       // num de filas (GLsizei)
    0,           // tamano del borde, usualmente es 0
    GL_RGB,     // formato y orden de los texels en RAM
    GL_UNSIGNED_BYTE, // tipo de cada componente de cada texel
    texels      // puntero a los bytes con texels (void *)
);
```

indica que se va a transferir (a la textura activa) la imagen almacenada en texels, como una textura 2D; para el nivel mipmap 0; que los texels se deben guardar en RGB; que la imagen tiene ancho x alto texels, sin borde; que la imagen se pasa en RGB con cada componente almacenada como unsigned byte, es decir como intensidades entre 0 y 255. El tamaño de la imagen (tanto en ancho como en alto) debe ser potencia de 2.

2: Si la iluminación está desactivada, el color de la textura sustituye al especificado con glColor.

En el buffer de texels, los tres bytes de cada texel se almacenan contiguos. Los $3n_x$ bytes de cada fila de texels se almacenan contiguos, de izquierda a derecha, y las n_y filas se almacenan contiguas, desde abajo hacia arriba. Con este esquema la imagen ocupará, lógicamente, $3n_x n_y$ bytes consecutivos en memoria.

Al llamar a esta función, OpenGL leerá los bytes de la RAM y los copiará en otra memoria (típicamente la memoria de vídeo o de la GPU, en un formato interno).

En GLU hay una alternativa más simple a `glTexImage2D` que no requiere imágenes de tamaño potencia de dos, y que además genera automáticamente versiones a múltiples resoluciones (mip-maps):

```
gluBuild2DMipmaps( GL_TEXTURE_2D,
    GL_RGB,      // formato interno
    ancho,       // num. de columnas (arbitrario) (GLsizei)
    alto,        // num de filas (arbitrario) (GLsizei)
    GL_RGB,      // formato y orden de los texels en RAM
    GL_UNSIGNED_BYTE, // tipo de cada componente de cada texel
    texels      // puntero a los bytes con texels (void *)
);
```

OpenGL permite especificar como se seleccionarán los texels en cada consulta posterior de la textura activa, cuando el pixel actual es igual o más pequeño que el texel que se proyecta en él usando la función `glTexParameter`. Para seleccionar el texel con centro más cercano al centro del pixel se usa:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
```

Para hacer interpolación bilineal entre los cuatro texels con centros más cercanos al centro del pixel:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

De la misma forma se puede controlar el comportamiento en cuando los texels son más pequeños que los pixeles usando como segundo parámetro `GL_TEXTURE_MIN_FILTER`.

También se usa la función `glTexParameter` para fijar el comportamiento cuando las coordenadas de textura se salen del intervalo (0, 1), utilizando como segundo parámetro `GL_TEXTURE_WRAP_S` para la coordenada de textura u y `GL_TEXTURE_WRAP_T` para la v:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

Las coordenadas de textura se asignan a los vértices con la función `glTexCoord2f`. El código del listado 6.1 dibuja un triángulo dando las coordenadas de textura de sus vértices.

El siguiente fragmento de código muestra la inicialización y uso de la textura:

```
GLuint *texName;
unsigned char* image;

// Lectura de imagen
```

Listing 6.1: Creación un triángulo asignando coordenadas de textura a los vértices.

```
glBegin(GL_TRIANGLES);
glTexCoord2f( s0, t0 );
glVertex3f( x0, y0, z0 );
glTexCoord2f( s1, t1 );
glVertex3f( x1, y1, z1 );
glTexCoord2f( s2, t2 );
glVertex3f( x2, y2, z2 );
glEnd();
```

```

glGenTextures(1, texName);
 glBindTexture(GL_TEXTURE_2D, *texName);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, w, h, 0, GL_RGB, GL_UNSIGNED_BYTE,
             image);
 ...
glBegin(GL_QUAD_STRIP){
    glNormal3f( 0.0, 0.0, -1.0 ); /* Vertical hacia atras */

    glTexCoord2f( 1, 0.0 );
    glVertex3f( x, 0, 0 );

    glTexCoord2f( 0.0, 0.0 );
    glVertex3f( 0, 0, 0 );
    ...
}

```

6.4. Texturas en Unity

Unity puede aplicar texturas a los objetos de forma fácil. En primer lugar debemos cargar la imagen de la textura como un assets. En el panel inspector de la textura se puede ajustar los parámetros de la textura (figura 6.11).

Aplicamos la textura desplazando el asset sobre la malla del objeto. En el inspector del objeto veremos que aparece un material con la textura en el campo *Albedo*. Aparecerá un nuevo assets de material creado a partir del material que tenía el objeto y la textura.

6.5. Bump Mapping

Hemos visto como utilizar texturas para modificar el color de las superficies, pero este no es el único uso de las texturas. También se pueden usar texturas para modificar las normales generando sensación de relieve, modificar la superficie, almacenar información de oclusión.

Bump mapping es el uso de texturas para alterar localmente las normales, generando sensación de relieve en una superficie plana. La textura usada puede tener un solo canal que almacena el desplazamiento de la superficie que se quiere representar respecto a la geometría (height map) o el vector de desplazamiento de la normal (normal map). Un normal map es una imagen en color en la que cada texel representa el vector que se debe sumar a la normal almacenada en el modelo (figura 6.12).

Unity permite hacer bump mapping usando mapas de normales. Al cargar la imagen de la textura se debe configurar el tipo de textura como "Normal Map". La figura 6.13 muestra el resultado de la aplicación de la textura de la figura 6.12 como mapa de normales a un cubo.

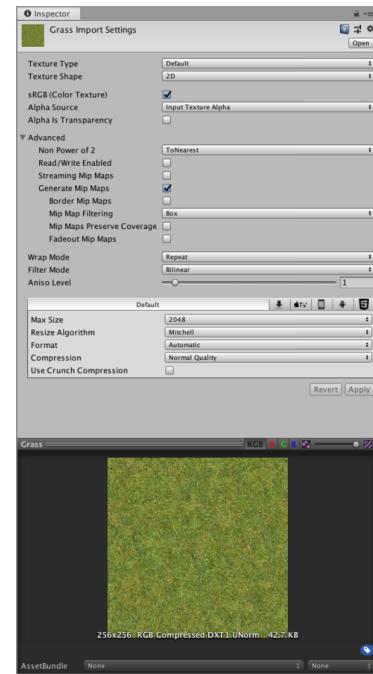


Figura 6.11: Panel de parámetros de una textura en Unity 3D.

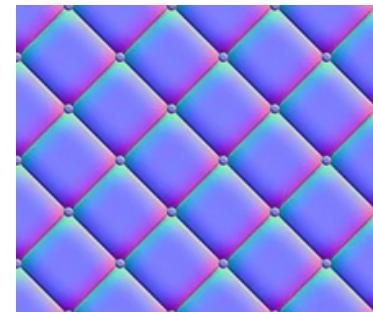


Figura 6.12: Mapa de normales.

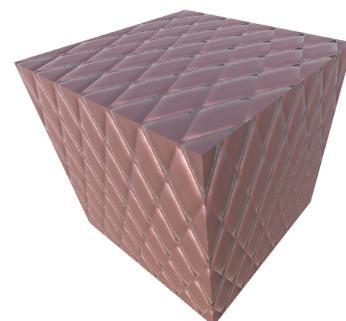


Figura 6.13: Visualización de un cubo con mapa de normales.

6.6. Ejercicios

1. Explica como diseñarías el modelo de la pirámide de Keops para aplicar como textura una fotografía aérea como la mostrada en la figura 6.14.
2. Como alternativa a la representación realizada en el ejercicio 1 plantea la posibilidad de crear la textura usando fotografías de cada una de las caras como la mostrada en la figura 6.15.
3. ¿Como se debe preparar la textura y que parámetros se debe utilizar para aplicarla al suelo de un escenario usando la imagen de la figura 6.16.
4. Se quiere crear un escenario urbano en el que se representarán edificios como paralelepípedos de diferente tamaño, cuya altura, en metros, es siempre múltiplo de 3 y menor o igual que 15. La anchura de cada pared del paralelepípedo, en metros, es múltiplo de 2, hasta un máximo de 16. Para darrealismo a la escena, se aplicara la textura de la imagen 6.17. El espacio de cada ventana (delimitado con un recuadro punteado en la imagen) debe ocupar una zona del edificio de 3m de altura por 2m de anchura. La anchura de cada pared del paralelepípedo, en metros, es múltiplo de 2, hasta un máximo de 16. Explique cómo crearía la geometría, la topología y asignaría las coordenadas de textura en el constructor de un paralelepípedo definido como:

`Edificio::Edificio(int ancho, int alto, int profundo);`

5. ¿Como se puede adaptar el ejercicio anterior para que el sistema funcione cuando las paredes tengan mas de 16 m?
6. Explica como diseñarías el modelo de un tetraedro para aplicarle como textura la imagen mostrada en la figura 6.18.
7. Genera un modelo de tierra usando un mapa mundi como textura como el mostrado en la figura 6.10. Prográmalos en OpenGL.
8. ¿Es posible generar el modelo de la tierra usando como textura dos vistas desde posiciones opuestas como la mostrada en la figura 6.19.
9. Explica como generarias un modelo 3D de un edificio usando textura la imagen de un recortable de cartulina como el de la figura 6.20. Indica como representarías el modelo y la textura.



Figura 6.14: Vista aérea de la pirámide de Keops.



Figura 6.15: Vistalateral de la pirámide de Keops.



Figura 6.16: Imagen para textura de suelo.

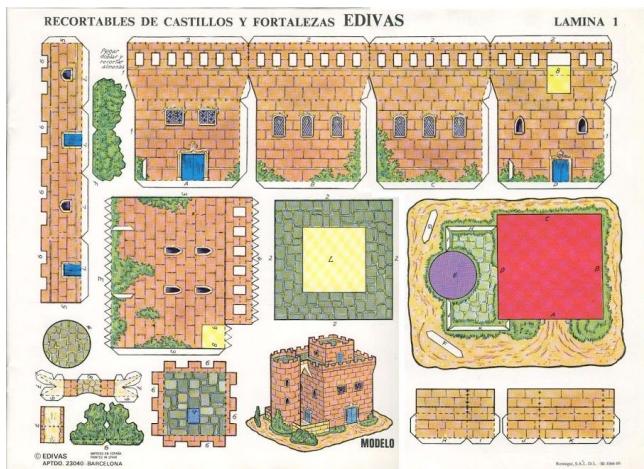


Figura 6.20: Recortable de un castillo.

10. Explica como se puede utilizar texturas para generar un efecto de animación de una superficie, como el movimiento de la hierba con el viento.
11. Plantea como se puede modificar el ejercicio 7 para que se muestre la mitad de la tierra de noche.
12. Se desea incluir una valla publicitaria digital en una escena. La valla ha de mostrar dos imágenes alternadas en el tiempo, intercambiándolas cada 10 segundos. El cambio de una imagen a otra se realiza mediante un desplazamiento vertical que dura un segundo. El objeto que representa la valla es un rectángulo. Diseña la textura y el modelo.



Figura 6.17: Imagen para textura de pared de edificio.

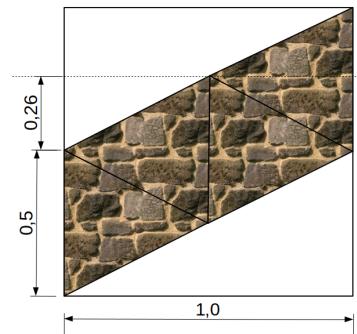


Figura 6.18: Imagen para aplicar como textura a un tetraedro.

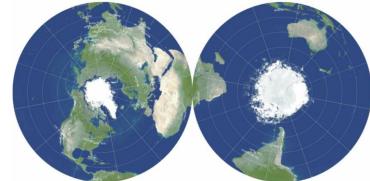


Figura 6.19: Dos imágenes de la tierra.

Informática Gráfica

Juan Carlos Torres

Curso 2024/25

Dpto. Lenguajes y Sistemas Informáticos
Universidad de Granada

Disclaimer

You can edit this page to suit your needs. For instance, here we have a no copyright statement, a colophon and some other information. This page is based on the corresponding page of Ken Arroyo Ohori's thesis, with minimal changes.

CC BY-NC-SA

 This book is released into the public domain using the CC BY-NC-SA. This license enables reusers to distribute, remix, adapt, and build upon the material in any medium or format for noncommercial purposes only, and only so long as attribution is given to the creator. If you remix, adapt, or build upon the material, you must license the modified material under identical terms.

To view a copy of the CC BY-NC-SA code, visit:

<https://creativecommons.org/licenses/by-nc-sa/4.0/>

Colophon

This document was typeset with the help of KOMA-Script and L^AT_EX using the kaobook class.

7

Sistemas interactivos: Selección

Un sistema gráfico interactivo es un conjunto de hardware y software que permite a los usuarios crear, manipular y visualizar modelos, respondiendo de manera fluida a las entradas del usuario. Estos sistemas permiten la interacción entre el usuario y la escena, adaptándose de manera inmediata a cambios como el movimiento de la cámara, la modificación de objetos o la selección de opciones a través de dispositivos de entrada.

En la primera lección se describieron algunos ejemplos típicos de estos sistemas.

Para que un sistema sea interactivo, su respuesta debe ser lo suficientemente rápida para que el usuario perciba la relación causa-efecto de sus acciones sobre los cambios en el modelo. Esto no implica un límite de tiempo de respuesta estricto, como en los sistemas de tiempo real. La **latencia** es el tiempo que transcurre entre la acción del usuario y la actualización de la imagen o la respuesta en el sistema de realidad virtual. Por tanto, en un sistema interactivo debe haber una latencia baja¹.

Los sistemas de realidad virtual son los que requieren la latencia más baja posible. Una latencia baja es crucial para que la experiencia de realidad virtual sea inmersiva y cómoda. Latencias altas pueden generar efectos negativos como mareos o incomodidad, ya que el cerebro percibe el desajuste entre los movimientos y la actualización visual. Idealmente, la latencia en un sistema de realidad virtual debería ser inferior a 20 milisegundos para garantizar una experiencia fluida.

Un aspecto esencial en el diseño de sistemas interactivos es ofrecer retroalimentación como respuesta a las acciones realizadas. Normalmente, esta información es la visualización de la escena modificada (como consecuencia del movimiento de un personaje o la creación de un objeto). En algunos casos se puede utilizar retroalimentación no gráfica (como vibración o sonido) o retroalimentación visual adicional a la visualización de la escena.

La realimentación (*feedback*) es el mecanismo que permite al sistema brindar información relevante al usuario para decidir la siguiente acción. La información de realimentación generada por el sistema depende del estado actual y de la información previamente ingresada por el usuario. Se puede utilizar con diferentes fines: mostrar el estado del sistema, como parte de una función de entrada o para reducir la incertidumbre del usuario. Por ejemplo, al dibujar líneas, una vez introducido el primer punto, se puede mostrar la línea que se crearía en la posición actual del cursor mientras éste se mueve.

Con frecuencia, el usuario debe ingresar una posición que cumpla con una determinada condición, como tener la misma coordenada X que la posición previa. En estas situaciones, la aplicación puede ayudar al usuario proporcionando {técnicas de interacción} que aseguren el cumplimiento de las restricciones de entrada. En el caso anterior, se puede hacer que el cursor solo se mueva en horizontal. Por ejemplo, al

| | |
|---------------------------------------|----|
| 7.1 Subsistema de entrada | 58 |
| 7.2 Interacción con GLUT | 59 |
| 7.3 Interacción con Unity3D | 61 |
| 7.4 Selección | 62 |
| 7.4.1 Selección en OpenGL | 64 |
| 7.4.2 Selección en Unity | 67 |
| 7.5 Ejercicios | 67 |

1: En un sistema interactivo puede haber operaciones específicas que no sean interactivas



Figura 7.1: Interacción gestual con Kinect en un sistema de realidad virtual.

crear elementos en un escenario, el sistema puede ajustar su altitud para que se apoyen en el elemento que esté en esa posición.

Nos ocuparemos en esta lección del subsistema de interacción, centrándonos en las operaciones de selección.

7.1. Subsistema de entrada

Los componentes básicos de un sistema gráfico interactivo son (figura 7.2):

Hardware

- Dispositivos de visualización: Se utilizan para mostrar la escena. Suelen ser monitores, pero también pueden ser otros equipos, como sistemas de proyección o gafas de realidad virtual.
- Dispositivos de entrada: Permiten la interacción del usuario y pueden incluir dispositivos más allá del ratón y teclado, como controladores táctiles, sensores de posición, dispositivos hapticos u otros dispositivos (figura 7.1).

Software

- Modelo geométrico: Contiene la representación de la escena, que podrá ser modificada según las operaciones de entrada recibidas.
- Motor de renderizado: Genera las imágenes a partir del modelo geométrico.
- Subsistema de interacción: Se encarga de interpretar y procesar las acciones realizadas por el usuario.

Cuando se actúa sobre un dispositivo de entrada el subsistema de interacción interpreta la acción, envía al modelo geométrico las ordenes correspondientes y si es necesario envía al motor de rendering la petición para incluir información de realimentación.

En este proceso es importante conseguir que el código de la aplicación sea independiente del modelo y tipo de dispositivo concreto utilizado. Para conseguirlo se independiza la capa de software que accede directamente a los dispositivos de forma que la aplicación a modelos abstractos de los dispositivos (**dispositivos lógicos**). Esto permite por ejemplo que una aplicación pueda leer posiciones de pantalla con diferentes dispositivos, incluso con dispositivos que no generan una posición como información de salida.

Idealmente las APIs gráficas pueden definir los siguientes dispositivos lógicos:

Locator: Devuelve una posición.

Pick: Devuelve el identificador de un objeto.

Stroke: Devuelve una lista de posiciones.

String: Devuelve una cadena de caracteres.

Choice: Devuelve uno de entre n items.

Valuator: Devuelve un número.

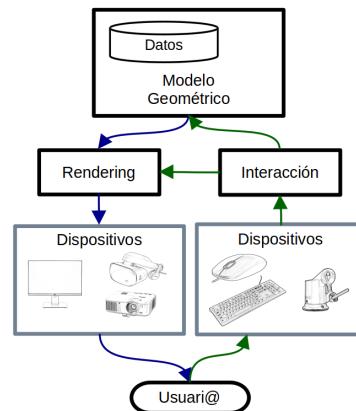


Figura 7.2: Arquitectura de un sistema gráfico interactivo.

La sincronización entre los dispositivos y la aplicación puede realizarse de varios modos. La aplicación puede realizar una **petición** al dispositivo y quedar suspendida hasta que se introduzca la información. Este es el modo de funcionamiento de las operaciones de lectura de un programa. Por ejemplo la sentencia

```
cin >> x;
```

suspenderá el programa hasta que se introduzca un valor. Este modo de funcionamiento es adecuado cuando el programa no puede continuar su ejecución hasta que se realice la lectura. En un sistema interactivo se usa para acciones concretas que necesitan que deben realizarse para que el programa pueda continuar, por ejemplo dar el nombre del archivo que contiene el modelo que se va a cargar.

En algunos casos el dispositivo puede estar realizando una medida de una variable (por ejemplo la altura de agua de un depósito) y este valor puede ser consultado por el programa cuando lo necesite. En este modo de funcionamiento (denominado **sample**) el programa no se suspende ya que el dato está siempre disponible. Este modo es útil cuando la aplicación debe muestrear una variable continua registrada por un dispositivo.

7.2. Interacción con GLUT

La mayor parte de las operaciones en los sistemas interactivos se realizan en modo **evento**. En este modo la acción sobre el dispositivo genera un evento que es capturado por el subsistema de entrada.

La aplicación registra las funciones (**callback**) que se deben ejecutar como respuesta a los eventos que necesita procesar.

El sistema de gestión de eventos (activado al llamar a `glutMainLoop`) que se encarga de encolar los eventos que se producen para los que hay un callback registrado y de llamar a dichas funciones para dar respuesta a los eventos (figura 7.3).

Una vez arrancada la aplicación el flujo de ejecución de la aplicación queda gestionado por GLUT que irá detectando los eventos producidos (tanto externos como internos) y llamando a los diferentes callbacks cada vez que suceda un evento [Kil96]. El sistema de gestión de eventos de GLUT se activa con la llamada a `glutMainLoop()`.

Los principales eventos registrados por GLUT responden a entradas de dispositivos (teclado, ratón, spaceball), modificación de la ventana gráfica o eventos internos (dibujo, temporizador, función de fondo). Para algunos dispositivos hay mas de un evento asociado.

Para cada evento hay una función de GLUT encargada de registrar el callback que lo va a procesar. Las funciones de registro de callback de ratón son las siguientes:

glutMouseFunc: eventos de cambio de estado de los botones del ratón.
glutMotionFunc: eventos de desplazamiento con algún botón pulsado.
glutPassiveMotionFunc: eventos de desplazamiento sin botones pulsados.

La interfaz de los callback debe ser la siguiente²:

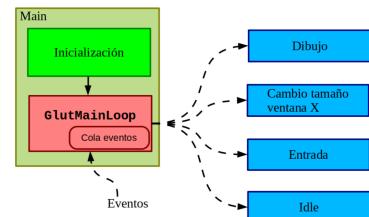


Figura 7.3: Estructura típica de un programa OpenGL con glut. Los rectángulos azules representan callbacks.

2: La interfaz está fijada ya que las llama GLUT.

```
MouseFunc(int button, int state, int x, int y);
MotionFunc(int x, int y);
PassiveMotionFunc(int x, int y);
```

El parámetro *button* recibe el código del botón sobre el que se ha actuado (GLUT_LEFT_BUTTON, GLUT_MIDDLE_BUTTON, o GLUT_RIGHT_BUTTON); *state* el estado actual del botón (GLUT_UP o GLUT_DOWN), *x* e *y* reciben la posición del cursor en coordenadas de la ventana.

Con este modo de funcionamiento no se puede predecir el orden en el que se van a ejecutar las funciones de los distintos callback, por lo que no se puede presuponer cual va a ser el estado de la aplicación cuando se está procesando un evento. Así, por ejemplo, no podemos dibujar una linea de realimentación desde un callback de ratón, porque no se sabe en que estado se encuentra el proceso de dibujo. En este caso será necesario pasar información indicando la línea a dibujar a la función que dibuja, este paso se puede realizar con variables compartidas o con llamadas a funciones del módulo de dibujo.

El siguiente fragmento de código muestra como utilizar los callback de ratón para girar la cámara:

```
1 int MOUSE_LEFT_DOWN=0;
2 int MOUSE_X, MOUSE_Y;
3
4 void clickRaton( int boton, int estado, int x, int y )
5 {
6     if(boton==GLUT_LEFT_BUTTON && estado==GLUT_DOWN) {
7         MOUSE_LEFT_DOWN=1;
8         MOUSE_X=x;
9         MOUSE_Y=y;
10    }
11    else if(boton==GLUT_LEFT_BUTTON && estado==GLUT_UP) {
12        MOUSE_LEFT_DOWN=0;
13        ISINTERACTING=0;
14    }
15    glutPostRedisplay();
16 }
17
18 void RatonMovido( int x, int y )
19 {
20     float ax,ay,az,d;
21     getCamara (ax, ay, az, d);
22     if(MOUSE_LEFT_DOWN) {
23         if(x!=MOUSE_X) ay+=x-MOUSE_X;
24         if(y!=MOUSE_Y) ax+=y-MOUSE_Y;
25         setCamara (ax, ay, az, d);
26         MOUSE_X=x;
27         MOUSE_Y=y;
28         glutPostRedisplay();
29    }
30 }
```

Estas funciones se deberán registrar como callback con las siguientes llamadas:

```
glutMouseFunc (clickRaton);
glutMotionFunc (RatonMovido);
```

La variable *MOUSE_LEFT_DOWN* se utiliza para saber si es el botón derecho el que está pulsado cuando se produce un movimiento de ratón. Las variables *MOUSE_X* y *MOUSE_Y* para recordar la última posición del ratón y calcular con el desplazamiento el desplazamiento a aplicar a la

cámara. Las funciones *getCamara* y *setCamara* acceden a los ángulos de giro de la cámara y su distancia.

7.3. Interacción con Unity3D

Unity escucha continuamente los dispositivos de entrada conectados al sistema (teclados, ratones, gamepads, pantallas táctiles, etc.). Cada vez que ocurre un evento de entrada (por ejemplo, presionar una tecla, mover el ratón, tocar la pantalla), Unity lo detecta y lo almacena en un sistema de entrada interno. Unity no gestiona los eventos de entrada en tiempo real, sino que acumula el estado de los dispositivos de entrada entre cada frame.

La interacción se programa en scripts asociados a los componentes de la escena, que pueden tener un método *update* que se ejecuta en cada frame. Para detectar las entradas del usuario, los scripts utilizan la clase *Input* que ofrece métodos como *Input.GetKeyDown* (para ver si una tecla está pulsada), *Input.GetMouseButtonUp*, y *Input.GetAxis*, que ayudan a consultar el estado de diferentes tipos de entrada en cualquier momento durante la ejecución del juego.

En cada ciclo de frame, Unity procesa los cambios en los estados de entrada (por ejemplo, si una tecla fue presionada, liberada o si un botón del ratón fue pulsado) y actualiza los estados que se pueden consultar mediante la clase *Input*.

Para procesar las entradas de ratón se pueden usar las siguientes funciones de *Input*:

Input.GetMouseButtonDown(int button): Devuelve true en el frame en el que se presiona un botón del ratón.

Input.GetMouseButton(int button): Devuelve true mientras el botón del ratón está pulsado.

Input.GetMouseButtonUp(int button): Devuelve true en el frame en el que se libera el botón del ratón.

Input.mousePosition: Devuelve la posición del cursor del ratón en la pantalla, en coordenadas de ventana.

La clase *Input* implementa un dispositivo lógico *GetAxis* que representa un desplazamiento horizontal o vertical del cursor sobre el que se puede actuar desde diferentes dispositivos.

Además, los script asociados a los objetos pueden también consultar la situación del ratón respecto al objeto:

OnMouseDown: se ha presionado el botón del ratón mientras el puntero está sobre el objeto.

OnMouseUp: se ha liberado el botón del ratón cuando el puntero está sobre el objeto.

OnMouseEnter: El puntero del ratón ha entrado en el área del objeto.

OnMouseExit: Se llama cuando el puntero del ratón sale del área del objeto.

OnMouseOver: Se llama cada frame mientras el puntero del ratón está sobre el objeto.

El siguiente script gira una cámara orbital:

```

1 using UnityEngine;
2
3 public class CameraController : MonoBehaviour
4 {
5     public float mouseSensitivity = 100.0f; // Sensibilidad del
       raton
6     public float distanceFromOrigin = 10.0f; // Distancia de la
       camara al origen
7     private float xRotation = 0.0f;
8     private float yRotation = 0.0f;
9
10    void Start()
11    {
12    }
13
14    void Update()
15    {
16        // Obtener la entrada del raton
17        float mouseX = Input.GetAxis("Mouse X") * mouseSensitivity *
           Time.deltaTime;
18        float mouseY = Input.GetAxis("Mouse Y") * mouseSensitivity *
           Time.deltaTime;
19        // Acumular las rotaciones alrededor de los ejes X e Y
20        xRotation -= mouseY; // Movimiento vertical
21        yRotation += mouseX; // Movimiento horizontal
22        // Limitar la rotacion vertical a 90 grados
23        xRotation = Mathf.Clamp(xRotation, -90.0f, 90.0f);
24        // Rotar la camara alrededor del origen en el eje horizontal
25        transform.RotateAround(Vector3.zero, Vector3.up, mouseX *
           mouseSensitivity * Time.deltaTime);
26        // Rotar en el eje vertical (xRotation)
27        transform.RotateAround(Vector3.zero, transform.right, -mouseY
           * mouseSensitivity * Time.deltaTime);
28        // Asegurarnos de que la camara siempre mire al origen
29        transform.LookAt(Vector3.zero);
30    }
31 }
```

7.4. Selección

La operación de selección permite indicar interactivamente un componente de la escena. Esta operación es esencial en cualquier sistema gráfico, ya que es necesaria para realizar cambios en la escena de forma interactiva, para por ejemplo: eliminar componentes, cambiar sus atributos, copiarlos o aplicarles transformaciones geométricas.

Para poder realizar la selección (*pick*) los componentes de la escena deben tener asociados identificadores. Por ejemplo, para poder seleccionar triángulos, cada uno debe tener asociado un identificador, en este caso puede ser un entero. En otras casos el identificador puede estar compuestos por varios ítems, por ejemplo en un escena en la que aparecen diferentes tipos de objetos articulados el identificador puede ser un par de enteros: (objeto, segmento del objeto).

Los identificadores se pueden asociar a distintos niveles, permitiendo controlar el **ámbito** de la selección. En una misma aplicación podemos

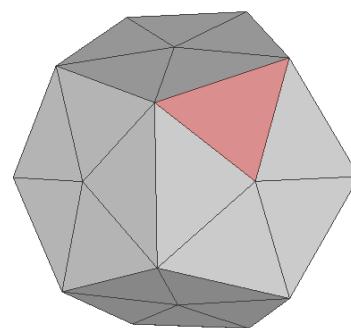


Figura 7.4: Selección de un triángulo.

necesaría seleccionar poliedros, triángulos, aristas o vértices para realizar diferentes operaciones.

El proceso que se realiza para seleccionar un componente de la escena implica:

1. El usuario hace click en una posición de pantalla.
2. Se busca el componente (en el ámbito seleccionado) que se corresponde con esa posición de pantalla.
3. El sistema muestra (como información de feedback) el objeto seleccionado (figura 7.4).

En algunos casos no es necesario dar información de realimentación.

Dependiendo del método de selección empleado, la correspondencia puede implicar que el componente sea el que se dibuje en ese pixel, que se proyecte en el pixel, o que se dibuje o proyecte próximo al pixel.

La búsqueda se puede realizar de varios modos. Los más usados son: Intersección rayo escena y Codificando el id de objeto como color.

Intersección rayo escena

Para determinar qué componente se debe seleccionar se puede construir la semirecta que se proyecta en el pixel partiendo de la cámara (a estas semirectas se les suele llamar **rayos**). Para construirla se calcula la posición 3D del pixel en el plano de recorte delantero³.

Se utiliza este rayo para calcular su intersección con todos los componentes de la escena⁴. El cálculo de la intersección dependerá de la geometría del componente, es diferente calcular la intersección con una esfera, con un triángulo, un paralelepípedo o un triángulo, por citar algunos ejemplos.

A modo de ejemplo, para calcular la intersección de un rayo con una esfera (figura 7.5) se debe encontrar si hay algún que cumpla simultáneamente la ecuación del rayo, que en forma paramétrica es:

$$\mathbf{r}(t) = \mathbf{O} + t \cdot \mathbf{d} \quad (7.1)$$

donde \mathbf{O} es la posición de la cámara, \mathbf{d} el vector director de la recta y $t \in (0, \infty)$ el parámetro que permite recorrer la recta.

La ecuación de la esfera es:

$$(\mathbf{p} - \mathbf{C})^2 - R^2 = 0 \quad (7.2)$$

donde \mathbf{C} es el centro de la esfera y R es su radio.

Para resolver la intersección debemos encontrar puntos que cumplan ambas ecuaciones:

$$(\mathbf{O} + t \cdot \mathbf{d} - \mathbf{C})^2 - R^2 = 0 \quad (7.3)$$

Que es una ecuación de segundo grado, cuya solución es:

3: Y se utiliza la ecuación de la recta que pasa por dos puntos.

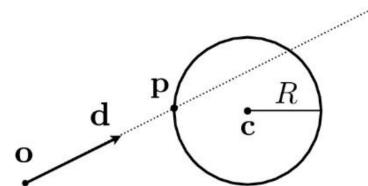


Figura 7.5: Intersección rayo-esfera.

4: Aquí lo que se considere un componente dependerá del ámbito de la selección.

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

con

$$a = d \cdot d$$

$$b = 2(\mathbf{O} - \mathbf{C}) \cdot d$$

$$c = (\mathbf{O} - \mathbf{C}) \cdot (\mathbf{O} - \mathbf{C}) - R^2$$
(7.4)

Por tanto habrá intersección si $(b^2 - 4ac) > 0$. En este caso podremos calcular el punto de intersección.

Si hay varios objetos intersectados podremos decidir quedarnos con el visible, que es el que está mas cerca de la cámara (menor valor de t).

Codificando el id de objeto como color

Para realizar la selección se dibuja la escena haciendo que como colores se usen los identificadores de los componentes. De esta forma, una vez dibujada la escena, podemos leer el valor del pixel en que se encuentra el cursor. La figura 7.7 corresponde a la visualización de identificadores de la escena de la figura 7.6.

En este proceso es necesario que no se modifique el color asignado a cada objeto, para ello se deben desactivar el cálculo de iluminación y el difuminado. También es esencial asignar los colores como valores enteros para evitar que se produzcan errores por redondeo.

Otro aspecto esencial es hacer que esta imagen de selección no se muestre al usuario. Esto se puede conseguir dibujando la imagen en un frame oculto o no intercambiando los buffers de imagen tras la selección.

7.4.1. Selección en OpenGL

OpenGL disponía de un modo específico de selección (basado en recortar la imagen en torno al cursor) que ha quedado obsoleto a partir de la versión 3.0. La forma más simple de realizar una selección en las versiones actuales es Codificando el id de objeto como color.

Para realizar la selección por color se debe dibujar la escena cuando se quiera realizar la selección llamando a la función de dibujo desactivando la iluminación y el dithering. Para garantizar que se dibuja la misma escena es conveniente tener una sola versión del método de dibujo. Para ello se puede crear una función que dibuje la escena, y hacer que el callback de dibujo la llame después de activar la iluminación y el dithering:

```

1 void dibujar()
2 {
3     glEnable(GL_LIGHTING);
4     glEnable(GL_DITHER);
5     glClearColor(0,0,0,1); // Fija el color de fondo
6     dibujoEscena();
7     glutSwapBuffers();
8 }
```

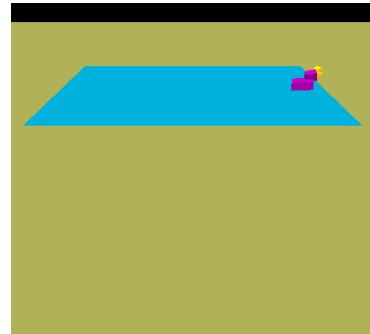


Figura 7.6: Escena de representada con iluminación.



Figura 7.7: Imagen generada con identificadores como colores correspondiente a la escena de la figura 7.6.

En este caso la función *dibujar* será la que se active como callback:

```
glutDisplayFunc( dibujar );
```

El método *dibujaescena* será el encargado de renderizar la escena, tanto para visualizarla como para seleccionar. Por tanto deberá asignar tanto los materiales usados al renderizar como los colores que codifican los identificadores para la selección ⁵.

El siguiente fragmento de código muestra un fragmento de código de ejemplo:

```
1 void dibujaEscena()
2 {
3     glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
4     ...
5     glLightfv( GL_LIGHT0, GL_POSITION, pos );
6     glTranslatef(-5, 3,-5);
7     // Dibuja elemento no seleccionable
8     glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, gris );
9     dibujarFondo();
10    // Dibuja elementos seleccionables compuestos
11    for(i=0;i<n;++i)
12        glMaterialfv( GL_FRONT, GL_AMBIENT_AND_DIFFUSE,color);
13        glPushMatrix();
14        glTranslatef(pato[i].x, 0,pato[i].z);
15        ColorSeleccion( i, 0);
16        dibujaCuerpo();
17        ...
18        glPushMatrix();
19        glMaterialfv( GL_FRONT, GL_AMBIENT_AND_DIFFUSE,naranja);
20        ColorSeleccion( i, 1);
21        dibujaParte1();
22        ...
23        glPopMatrix();
24    glPopMatrix();
25 }
```

5: Con iluminación activada se usarán los materiales y con iluminación desactivada los colores.

En este ejemplo asumimos que tenemos n objetos con la misma estructura. Cada objeto tiene un cuerpo y varias partes que queremos que se seleccionen independientemente. A cada elemento seleccionable se le asignan dos identificadores, el primero es el número de objeto (*i*), el segundo es el número de la parte del objeto.

La codificación de los colores (que en el ejemplo anterior se realiza en *ColorSeleccio*) se debe realizar dando los colores como enteros, para evitar errores de redondeo. Si no se tienen mas de 255 objetos y estos no tiene mas de 255 partes, la función *ColorSeleccio* del ejemplo anterior podría ser:

```
1 void ColorSeleccion( int i, int Parte)
2 {
3     unsigned char r = (i & 0xFF);
4     unsigned char g = (Parte & 0xFF);
5     glColor4ub(r,g,0,0);
6 }
```

En este caso, al no necesitar mas de 255 valores podemos dejar sin usar los canales verde y alfa.

La asignación de color se realiza con la función glColor4ub que recibe cuatro *unsigned bytes* para evitar que se produzcan errores por redondeo.

La selección se iniciará con un evento de entrada, que deberá realizar la selección e interpretar la información obtenida. Por ejemplo, si la selección se realiza al pulsar el botón izquierdo del ratón, el código del callback podría ser:

```

1 void clickRaton( int boton, int estado, int x, int y )
2 {
3     int parte=-1, i=-1,k=0;
4     if(boton== GLUT_LEFT_BUTTON && estado == GLUT_DOWN) {
5         // Se ha actuado sobre el izquierdo que esta ahora pulsado
6         k=pick(x,y,&i,&parte);
7         if(k>-1) { // se ha seleccionado algo
8             switch (parte) {
9                 case 0:
10                 ...
11             }
12 }
```

En el que se ha creado el código de selección en la función *pick* que usa como parámetros la posición del cursor (*x, y*) y el identificador encontrado, devolviendo un valor negativo si no se han leído identificadores. La función *pick* es el núcleo del proceso de selección:

```

1 int pick(int x, int y,int * i, int * parte)
2 {
3     GLint viewport[4];
4     unsigned char data[4];
5     int f=0;
6     glGetIntegerv (GL_VIEWPORT, viewport);
7     glDisable(GL_DITHER);
8     glDisable(GL_LIGHTING);
9     glColorColor(0,0,1,1);    // Fija el color de fondo a negro
10    dibujoEscena();
11    glFlush();
12    glReadPixels(x, viewport[3] - y,1,1, GL_RGBA, GL_UNSIGNED_BYTE,
13                  data);
14    *i=data[0];
15    *parte=data[1];
16    if(data[2]>0) f=-1;
17    glEnable(GL_LIGHTING);
18    glEnable(GL_DITHER);
19    return f
```

Las tres primeras líneas (3,4 y 5 declaran variables para almacenar el tamaño de la zona de dibujo (*viewport*) el color que se va a leer de la imagen de selección y el valor a devolver indicando si se ha seleccionado algún elemento. La línea 6 obtiene las dimensiones del área de dibujo, que se necesita para poder invertir la coordenada *Y*.

A continuación se desactiva el dithering y la iluminación y se fija el color de fondo del buffer de imagen (que puede ser distinto del que se va a usar para renderizar la escena). Este color debe ser diferente de los usados para codificar los identificadores de los objetos. En este ejemplo se está asignando color al canal verde que no se usa en los identificadores.

A continuación se dibuja la escena (línea 10). Se realiza una llamada a `glFlush`, que fuerza a que se realicen todas las funciones de OpenGL que se han enviado hasta ese momento y se lee el color del pixel que se encuentra en la posición del cursor. Los argumentos de esta función son: la posición x,y del cursor, el ancho y alto del área de pixels que se leen, el formato y el buffer en el que se van a leer.

Las líneas siguientes (13, 14 y 15) decodifican el identificador y determinan si se ha pulsado algún objeto seleccionable.

Por último se activa la iluminación y el dithering (esto es necesario solo si no se han incluido en el callback de dibujo).

7.4.2. Selección en Unity

La selección en Unity se puede realizar con el método `OnMouseDown` o calculando la intersección rayo-objeto con el método `Raycast`⁶. El siguiente código realiza la selección usando el `Raycast`:

```

1 void Update()
2 {
3     if (Input.GetMouseButtonDown(0)){// Devuelve verdadero si se ha pulsado el botón en este frame
4         RaycastHit hitInfo = new RaycastHit();
5         bool hit = Physics.Raycast(Camera.main.ScreenPointToRay(Input.
6             mousePosition), out hitInfo); //Devuelve verdadero si el rayo
intersecta el collider de un objeto
7         if (hit) {
8             Selected = hitInfo.transform.gameObject; // Objeto
seleccionado
9             Debug.Log("Hit " + Selected.name); // Muestra el nombre del
objeto en consola
10        }
11    }
12 }
```

6: Es necesario que el objeto tenga asociado un colisionador.

7.5. Ejercicios

1. Dar un ejemplo de situaciones en las que sea preferible utilizar cada uno de los modos de entrada.
2. ¿Es posible implementarla lectura de cadenas de caracteres, utilizando como dispositivo físico un ratón? ¿Porqué / cómo?
3. ¿Es posible crear funciones de entrada en modo evento en un sistema gráfico que tenga solo entrada en modo muestreo?
4. Supongamos un sistema gráfico que dispone tan sólo de entrada en modo evento. ¿Es posible crear una librería de funciones que, utilizando las funciones de entrada de dicho sistema, implemente entrada en modo sample?
5. Realizar una implementación en pseudocódigo de una función de lectura de posición del cursor en modo muestreo para glut.
6. Redactar un algoritmo que permita editar un rectángulo, utilizando dos puntos para seleccionar la operación a realizar. El punto central del borde inferior debe permitir cambiar la posición y la esquina superior derecha el tamaño. Al pulsar el botón izquierdo estando

sobre alguno de los puntos de selección, se entra en el proceso de edición, del que se sale al soltarlo.

7. Implementar en C el procedimiento que permitiría realizar un resaltado, cambiando el color de las aristas del objeto poligonal sobre el que se hace click.
8. Suponiendo el modelo articulado del ejercicio 12 implementado en OpenGL explica detalladamente que tendrías que programar para permitir que se seleccionasen los diferentes tramos de la escalera.
9. Como se debe modificar el callback de ratón del ejercicio anterior para que se puedan modificar los grados de libertad interactivamente cuando se selecciona un tramo, desplazando el ratón mientras el botón izquierdo siga pulsado.
10. Como se debe modificar la codificación del identificador en el ejercicio anterior si queremos tener varias escaleras.
11. Se tiene una escena compuesta por M objetos representados por mallas cada una con N triángulos. Las mallas están almacenadas usando la siguiente clase Malla:

```

1 typedef struct{
2     float x,y,z;
3 } punto;
4
5 typedef struct{
6     int v0,v1,v2;
7 } cara;
8
9 class Malla : public Objeto3D
10 {
11     public:
12     std::vector<punto> v; // vertices
13     std::vector<cara> c; // caras
14     std::vector<punto> nv; // normales de vertice
15     std::vector<punto> nc; // normales de cara
16     ...
17     void draw(){
18         float color[4] = { 0.5, 1.0, 0.5, 1 };
19         glShadeModel(GL_FLAT);
20         glMaterialfv (GL_FRONT, GL_AMBIENT_AND_DIFFUSE, color);
21         glBegin(GL_TRIANGLES);
22         for(int i =0; i<caras.size();i++){
23             glNormal3f(nc[i].x,nc[i].y,nc[i].z);
24             glVertex3f(v[c[i].v0].x,v[c[i].v0].y,v[c[i].v0].z);
25             glVertex3f(v[c[i].v1].x,v[c[i].v1].y,v[c[i].v1].z);
26             glVertex3f(v[c[i].v2].x,v[c[i].v2].y,v[c[i].v2].z);
27         }
28         glEnd();
29     }
30 };

```

Explica como se pueden asignar y codificar los identificadores para que usando el método de selección por color se obtenga el número de objeto y el número de triángulo en la malla.

12. Con la misma representación del ejercicio anterior explica que se debe cambiar para poder seleccionar aristas y vértices.
13. En el ejercicio anterior ¿Como se puede conseguir que se seleccionen los vértices cuando se haga click en una zona próxima al vértice?.
14. ¿Hay alguna limitación al tamaño del modelo impuesta por la

forma de asignar los identificadores? ¿Se puede conseguir que no haya límites al tamaño del modelo?

Informática Gráfica

Juan Carlos Torres

Curso 2024/25

Dpto. Lenguajes y Sistemas Informáticos
Universidad de Granada

Disclaimer

You can edit this page to suit your needs. For instance, here we have a no copyright statement, a colophon and some other information. This page is based on the corresponding page of Ken Arroyo Ohori's thesis, with minimal changes.

CC BY-NC-SA

 This book is released into the public domain using the CC BY-NC-SA. This license enables reusers to distribute, remix, adapt, and build upon the material in any medium or format for noncommercial purposes only, and only so long as attribution is given to the creator. If you remix, adapt, or build upon the material, you must license the modified material under identical terms.

To view a copy of the CC BY-NC-SA code, visit:

<https://creativecommons.org/licenses/by-nc-sa/4.0/>

Colophon

This document was typeset with the help of KOMA-Script and L^AT_EX using the kaobook class.

8

Lectura de posiciones

Una posición es una coordenada. Una forma simple de leer coordenadas 3D es utilizar las operaciones de lectura del lenguaje para leer tres números reales. Por ejemplo en C++:

```
std::cin >> x >> y >> z;
```

Esto no nos sirve en un sistema gráfico interactivo ya que se realiza en modo petición (deteniendo la ejecución del programa) y porque obliga a que la persona que usa el sistema tenga que saber las coordenadas que quiere introducir.

Imaginemos que creamos un juego de la tablero (figura 8.1). Cuando se quiera mover una ficha el usuario tendrá que indicar las coordenadas de la posición en la que quiere dejarla (aunque en el caso de un juego de casillas se podría indicar el número de la casilla).

Introducir posiciones no solo es esencial para poder crear y editar objetos, también hace falta para introducir transformaciones geométricas y parámetros de las cámaras y luces.

8.1. Dispositivos de posicionamiento

Existe una gran variedad de dispositivos que permiten introducir posiciones. Podemos agruparlos en los siguientes grupos en función de la información que generan:

Desplazamientos 2D: Dispositivos que devuelven el desplazamiento realizado sobre una superficie, como el ratón. El sistema de entrada puede generar una posición 2D en pantalla integrando estos desplazamiento y usando como información de realimentación un cursor.

Desplazamientos 3D: Dispositivos que devuelven desplazamientos en 3 dimensiones como el spaceMouse (figura 8.3).

Posiciones 2D: Devuelve una posición en un plano o en pantalla. Ejemplos de este grupo son las pantallas táctiles, los lápiz ópticos y las tabletas digitalizadoras (figura 8.2).

Posiciones 3D: Devuelve una posición en el espacio. Este es el caso los brazos de medición, los dispositivos de seguimiento (trackers) y los dispositivos hapticos.

Gestos: Devuelven los gestos realizados. Estos sistemas reconocen todo o parte del cuerpo y permiten entregar información de los gestos realizados. Por ejemplo la Kinect reconoce todo el cuerpo y los dispositivos Leap Motion identifican gestos realizados con la mano (figura 8.4).

El software del sistema usado para gestionar la interacción muestra abstracciones de estos dispositivos que permiten que los programas puedan funcionar con diferentes tipos de dispositivos. En algunos casos

| | |
|---|----|
| 8.1 Dispositivos de posicionamiento | 70 |
| 8.2 Métodos de posicionamiento | 71 |
| 8.3 Lectura en OpenGL | 74 |
| 8.4 Lectura en Unity | 75 |
| 8.5 Entrada de transformaciones geométricas | 76 |
| 8.6 Ejercicios | 77 |



Figura 8.1: En un juego de tablero la forma natural de interactuar es apuntar a la posición en la que queremos dejar la ficha.



Figura 8.2: Tableta digitalizadora.



Figura 8.3: SpaceMouse. Devuelve desplazamientos 3D y rotaciones 3D.

incluso haciendo que la información obtenida sea diferente de la generada por el programa. Por ejemplo, los ratones generan desplazamientos, pero el callback de ratón de glut devuelve una posición 2D.

Una característica importante de los dispositivos de entrada es el número de variables independientes (grados de libertad) que permiten controlar. Se suele identificar con las siglas *DF* abreviatura de la denominación en inglés (Degree of Freedom). El dispositivo de la figura 8.3 tiene *6DF*.

8.2. Métodos de posicionamiento

Los métodos que se pueden usar para leer posiciones dependerán de si se está trabajando con modelos 2D o 3D y de los dispositivos de entrada disponibles. Nos vamos a centrar en lectura de posiciones utilizando dispositivos 2D, que generan directa o indirectamente una posición en pantalla.

En el proceso de lectura el usuario indicará que va a introducir una posición (por ejemplo usando una opción de un menú para introducir un objeto en el escenario, o pulsando un botón del ratón) e irá desplazando el cursor hasta colocarlo en la posición deseada. Por último confirmará la posición pulsando un botón. Como respuesta a estas acciones el programa debe inicializar la lectura de posiciones (esto puede implicar por ejemplo mostrar una imagen diferente para cursor del ratón), e irá actualizando la posición del cursor con los desplazamientos que realiza el usuario, hasta que este indique el fin de la operación. Al finalizar se deberá calcular la posición introducida en coordenadas de la escena (si no se ha estado haciendo en el ciclo de seguimiento) y actualizar el modelo. La figura 8.5 muestra la correspondencia de las acciones del usuario y de la aplicación.

En el proceso que realiza la aplicación intervienen tres eventos: El inicio de la lectura se realiza como respuesta a un evento *glutMouseFunc*, cada iteración en el ciclo de seguimiento responde a un evento *glutMotionFunc* y el fin de la entrada a otro evento *glutMouseFunc* como se muestra en la figura 8.6. El siguiente fragmento es un ejemplo de lectura de posición en una aplicación 2D que introduce líneas, el dibujo comienza con la pulsación del botón izquierdo y termina cuando se libera el botón izquierdo. En el ciclo de seguimiento se dibuja una linea elástica como información de realimentación.

```

1 int dibujaLinea=0;
2 int linea[4];
3
4 void clickRaton( int boton, int estado, int x, int y )
{
6   if(boton==GLUT_LEFT_BUTTON && estado==GLUT_DOWN) {
7     linea[0]=x;
8     linea[1]=y;
9     linea[2]=x;
10    linea[3]=y;
11    dibujaLinea=1;
12  }
13  else if(boton==GLUT_LEFT_BUTTON && estado==GLUT_UP) {
14    dibujaLinea=0;
}

```

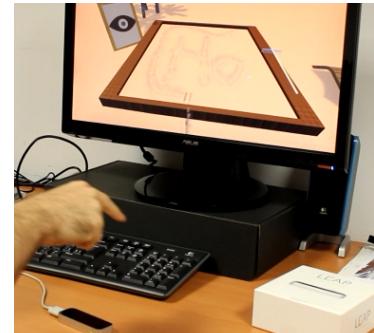


Figura 8.4: Interacción con gestos de la mano usando Leap Motion.

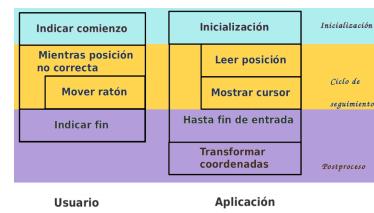


Figura 8.5: Correspondencia entre acciones del usuario y de la aplicación en una lectura de posición.

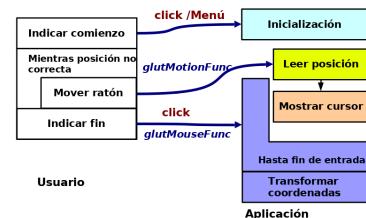


Figura 8.6: Estructura del proceso de lectura en callback.

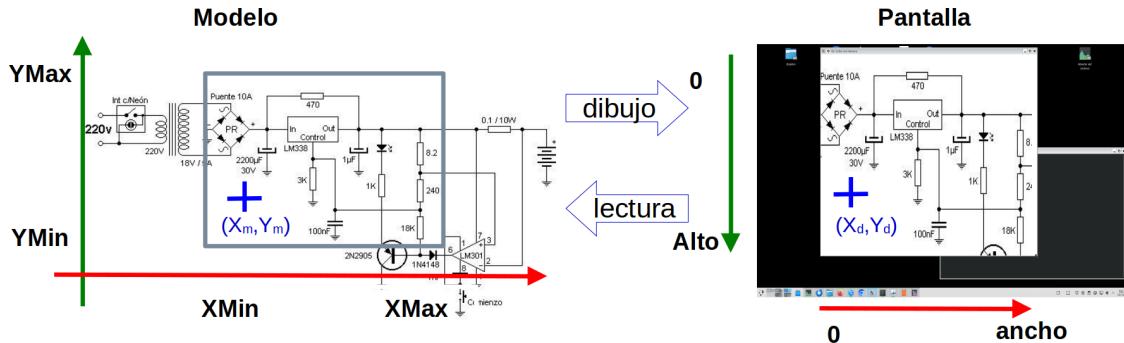


Figura 8.7: Conversión entre coordenadas de la escena y de pantalla en una aplicación 2D.

```

15     addLine(); // Crea la linea en el modelo.
16 }
17 glutPostRedisplay();
18 }

20 void RatonMovido( int x, int y )
21 {
22     if(dibujaLinea) {
23         linea[2]=x;
24         linea[3]=y;
25         glutPostRedisplay();
26     }
27 }

29 Dibuja()
30 {
31     ...
32     if(dibujaLinea) {
33         glBegin (GL_LINES);
34         glColor3f (1, 1, 0);
35         glVertex2f (linea[0],linea[1]);
36         glVertex2f (linea[2],linea[3]);
37         glEnd();
38     }
39     ....
40 }

```

Entrada de posiciones 2D

En una aplicación 2D (por ejemplo para el dibujo de planos de circuitos), el modelo geométrico está definido en el plano. Lo que se muestra en pantalla es una zona rectangular del modelo (figura 8.7). En este caso, la conversión entre coordenadas de la escena y coordenadas de pantalla se realiza con un escalado y dos traslaciones. Estas transformaciones se pueden invertir, por lo que conociendo el área del modelo que se está visualizando (*ventana*) y el área de la pantalla en la que se dibuja (*vireport*) se puede calcular la transformación de coordenadas de pantalla (x_d, y_d) a coordenadas de mundo (x_m, y_m):

$$\begin{aligned} X_m &= X_{min} + X_d * (X_{max} - X_{min}) / \text{ancho} \\ Y_m &= Y_{max} - Y_d * (Y_{max} - Y_{min}) / \text{alto} \end{aligned} \quad (8.1)$$

Entrada de posiciones 3D

En una aplicación 3D en cada pixel de pantalla se proyecta una semirecta, por lo que no es posible invertir la transformación. Así, en la figura 8.8 en el punto marcado en la imagen se proyecta toda la semirecta naranja. Si se está utilizando un dispositivo de entrada 2D es necesario dar información adicional para determinar la profundidad a la que se está indicando el punto.

La determinación del punto 3D puede realizarse con diferentes técnicas. Revisaremos algunas de ellas.

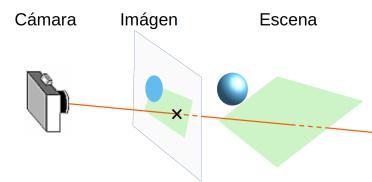


Figura 8.8: Conversión entre coordenadas de la escena y de pantalla en una aplicación 3D.

Utilización de tres vistas

Una de las formas más simples de introducir una posición 3D es descomponerla en varios posicionamientos en 2D. Para ello se puede visualizar la escena como tres vistas ortogonales (sobre los planos XY, YZ y ZX) como se muestra en la figura 8.9. La posición del cursor se muestra en las tres vistas de forma que el usuario modificar dos de las coordenadas en cada una de estas vistas.

Cursor 3D

Otra alternativa es dar como información de realimentación una visualización del cursor que permita percibir su posición en el espacio. Esto puede hacerse visualizando la sombra arrojada por el cursor sobre los objetos de la escena, activando la eliminación de partes ocultas para el cursor o mostrando su proyección sobre los planos del sistema de coordenadas (figura 8.10).

Si se utiliza un dispositivo 2D (como un ratón) para interaccionar será necesario utilizar un botón para seleccionar los ejes sobre los que actúa el dispositivo, por ejemplo moviendo la X y la Y con el botón izquierdo y la Z y la Y con el derecho.

Restricción a una superficie: intersección rayo escena

Vimos en la lección anterior como seleccionar objetos calculando la intersección de un rayo que pasa por el centro de proyección de la cámara y el punto correspondiente a un pixel en el plano de recortado delantero. Se puede utilizar el mismo mecanismo para seleccionar una posición sobre la superficie de un objeto. En este caso en lugar de devolver el identificador del objeto mas cercano con el que se produce la intersección se devolverán las coordenadas del punto de intersección.

Este procedimiento se puede usar para obtener posiciones restringidas a una superficie aunque la superficie no se está dibujando. Por ejemplo

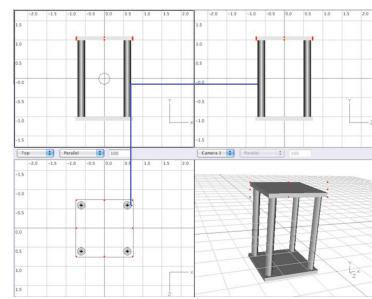


Figura 8.9: Entrada de una posición en una aplicación 3D usando tres vistas ortogonales.

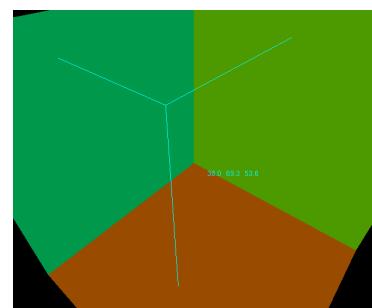


Figura 8.10: Visualización de un cursor 3D mostrando la proyección sobre los planos del sistema de coordenadas.

podemos obtener posiciones en el plano $Y = 0$ calculando la intersección con este plano, que no tiene porque existir en la escena.

Restricción a una superficie: selección de punto

También vimos que se pueden seleccionar componentes codificando su identificador como colores. Podemos realizar un proceso similar para obtener una posición, utilizando como color la posición.

Hay algunas diferencias entre el uso para selección y para posicionamiento. En posicionamiento necesitamos introducir los colores con valores reales, usando una componente del color para cada coordenada. Además necesitamos que se interpolen los colores dentro de los triángulos, por lo que deberemos utilizar sombreado de vértices.

La representación normalizada del color con valores de componentes entre 0 y 1, obliga a normalizar las coordenadas en la caja envolvente de la escena:

$$\begin{aligned}x' &= (x - X_{min}) / (X_{max} - X_{min}) \\y' &= (y - Y_{min}) / (Y_{max} - Y_{min}) \\z' &= (z - Z_{min}) / (Z_{max} - Z_{min})\end{aligned}\quad (8.2)$$

Indirectamente: Selección de objeto

Podemos usar una operación de selección para identificar un objeto y tomar como posición de entrada un punto preestablecido del mismo (por ejemplo el centro de su caja envolvente).

8.3. Lectura en OpenGL

Para leer una posición con OpenGL podemos usar cualquiera de los métodos anteriores.

Para utilizar varias zonas de dibujo se puede hacer uso de las funciones de creación de ventanas de glut:

```
int ventana, subventana;
ventana = glutCreateWindow ("titulo");
subventana = glutCreateSubWindow (ventana,x1,y1,ancho,alto);
```

La primera crea una ventana en el monitor devolviendo el identificador asignado. La segunda crea una subventana en la ventana que se indica comenzando en el pixel $(x1, y1)$. La subventana tendrá dimension ancho x alto, mientras que ventana será una zona de dibujo que ocupará el fragmento restante de la ventana original.

Cada zona de dibujo tendrá su propio contexto para OpenGL: sus propios callback, sus menús y su estado. Cuando necesitemos cambiar la zona de dibujo utilizaremos la función:

```
glutSetWindow( identificadorDeVentana );
```

Es necesario tener en cuenta que los eventos de redibujado se generan para la ventana activa, por lo que si queremos redibujarlas todas tendremos que activarlas una a una y generar el evento de redibujado en cada una de ellas.

Para crear una visualización como la de la figura 8.9 deberemos crear cuatro subventanas en la misma ventana y utilizar proyecciones diferentes para cada una de ellas¹.

Para utilizar un cursor 3D tendremos que hacer que el desplazamiento del ratón afecte a diferentes coordenadas en función del botón que se haya pulsado.

Para construir un rayo se puede utilizar la función *gluUnProject* que calcula un punto en el espacio de la escena en el rayo que pasa por un pixel. Con este punto y el centro de proyección (la posición (0, 0, 0)) podemos calcular la ecuación de la recta que contiene el rayo.

La función *gluUnProject* toma la posición en la pantalla de un punto, la profundidad, las matrices de transformación de modelo y de proyección, el viewport, y devuelve las coordenadas desproyectadas en el espacio del mundo (X_m, Y_m, Z_m). La interfaz de la función es:

```
int gluUnProject(double X, double Y, double Z,
    double *modelMatrix, double *projMatrix, int *viewport,
    double *Xm, double *Ym, double *Zm);
```

Donde (X, Y) es la coordenada de pantalla en pixeles (con la Y invertida), Z es la profundidad en espacio del ZBuffer, que es cero en el plano de recortado delantero y 1 en el trasero.

La profundidad de un pixel se puede leer con la función *glReadPixels*:

```
glReadPixels(X, Y, 1, 1, GL_DEPTH_COMPONENT, GL_FLOAT, &Z);
```

en la que (X, Y) es la posición del pixel, con la Y invertida.

1: Veremos como definir las proyecciones en la lección siguiente

8.4. Lectura en Unity

La forma mas simple de obtener una posición 3D en la escena de manera interactiva utilizando el ratón en Unity 3D es usando la clase *Ray* que permite generar un rayo desde la cámara y la posición del cursor en la pantalla hacia el espacio 3D de la escena. Podemos calcular la intersección del rayo con los objetos de la escena usando el método *Raycast*. El siguiente código muestra como obtener posiciones sobre un plano (no visible) colocado a una altura *planeHeight*. La posición obtenida se encuentra en la variable *hitPosition*:

```
using UnityEngine;
public class RaycastToPlane : MonoBehaviour
{
    public float planeHeight = 0f; // Altura del suelo
    void Update()
    {
        // Detectar si se ha hecho clic con el botón izquierdo
        if (Input.GetMouseButtonDown(0))
        {
            // Obtener la posición del cursor en la pantalla
```

```
Vector3 mousePos = Input.mousePosition;
// Convertir la posicion del cursor en un rayo
Ray ray = Camera.main.ScreenPointToRay(mousePos);
// Crear un plano virtual (plano XZ) a altura planeHeight
Plane groundPlane = new Plane(Vector3.up, new Vector3(0,
planeHeight, 0));
float distance; // Distancia desde el origen del rayo a
interseccion
// Realizar la interseccion del rayo con el plano
if (groundPlane.Raycast(ray, out distance))
{
    // Calcular el punto de interseccion en el espacio 3D
    Vector3 hitPosition = ray.GetPoint(distance);
    Debug.Log("Posicion 3D seleccionada: " + hitPosition);
    ...
}
}
```

8.5. Entrada de transformaciones geométricas

Las transformaciones geométricas se pueden definir a partir de pares de puntos, un punto inicial y un punto final. De esta forma podemos usar los métodos de entrada de posiciones para indicar transformaciones geométricas de forma interactiva.

Una traslación se puede definir mediante el vector que va de la posición original a la posición nueva (figura 8.11). El vector de traslación será:

$$\mathbf{P}_t = \mathbf{P}_2 - \mathbf{P}_1 \quad (8.3)$$

Si el cálculo del vector de traslación se realiza en el ciclo de seguimiento se puede mostrar el objeto trasladado como información de realimentación.

Para especificar una rotación mediante dos puntos en dos dimensiones calculamos el ángulo giro como el ángulo formado por los vectores que va del origen de coordenadas a los dos puntos (figura 8.12). En una rotación 3D se debe especificar en primer lugar el eje de giro. Una vez dado este, el ángulo se puede indicar mediante dos puntos en el plano perpendicular al eje.

De forma análoga se puede indicar el factor de escala realizando el cociente de las coordenadas de los dos puntos (figura 8.13):

$$\mathbf{S}_t = (x_2/x_1, y_2/y_1) \quad (8.4)$$

En cualquier caso, la transformación se puede calcular en el ciclo de seguimiento, dando como información de realimentación el resultado de la transformación.

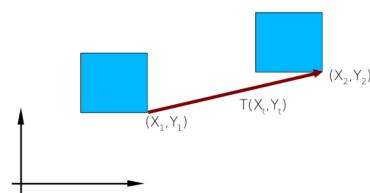


Figura 8.11: Entrada de un vector de traslación usando dos puntos.

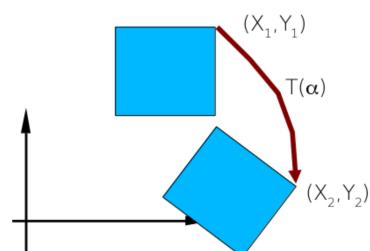


Figura 8.12: Entrada de un vector de rotación usando dos puntos.

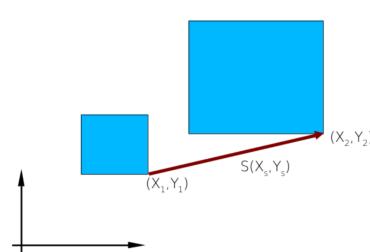


Figura 8.13: Entrada de un vector de escalado usando dos puntos

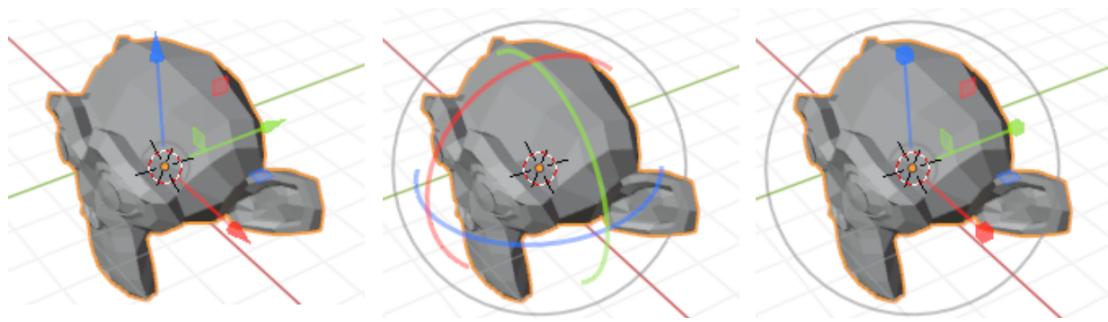


Figura 8.14: Gizmos para traslación, rotación y escalado.

Utilización de gizmos

Los programas interactivos suelen usar gizmos (manipuladores) para realizar las transformaciones geométricas de forma interactiva. Los gizmos son representaciones visuales que permiten manipular objetos en un entorno 3D. Por ejemplo, las flechas o manijas que permiten mover, rotar o escalar un objeto se podrían describir como herramientas de manipulación.

Cada transformación geométrica tiene un gizmo específico que se visualiza sobre los objetos cuando se les va a aplicar una transformación. El usuario puede seleccionar cada uno de los ejes del gizmo y modificar el valor de la componente correspondiente de la transformación geométrica.

8.6. Ejercicios

1. Escriba un procedimiento para la introducción de líneas utilizando la técnica de la línea elástica.
2. Una restricción gravitacional es una transformación de los puntos introducidos que hace que no se puedan introducir posiciones demasiado cerca de las ya introducidas. Supón una aplicación en la que se introducen puntos. Cada nuevo punto introducido se añade a un vector de puntos. Programa una restricción gravitacional que haga que los nuevos puntos se comprueben respecto a los ya existentes y si hay alguna a una distancia menor un valor prefijado u se devuelva el identificador del punto ya existen sin añadir un nuevo punto.
3. Describir cómo se podría realizar la interacción con un ratón, de tal forma que el cursor se moviese sobre un rectángulo, haciendo que cuando salga por un lado, aparezca por el lado opuesto.
4. Implementar en OpenGL un método para introducir una poligonal definida sobre un plano en el espacio.
5. Describe usando pseudocódigo el proceso de interacción para transformar objetos en OpenGL con Glut.
6. ¿Cómo se pueden implementar gizmos para transformar las mallas de un sistema que visualiza mallas indexadas con OpenGL con Glut?
7. ¿Se podría utilizar una lectura de posición para implementar una operación de selección?

8. Escribe la función de callback de ratón de glut para que se seleccione un objeto 2D pulsando el botón izquierdo y se traslade en el plano hasta que se libere el botón.
9. ¿Es posible implementar una lectura de posiciones en modo muestreo usando Glut?
10. Programa una función para obtener las coordenadas de un punto sobre una malla de triángulos?

Informática Gráfica

Juan Carlos Torres

Curso 2024/25

Dpto. Lenguajes y Sistemas Informáticos
Universidad de Granada

Disclaimer

You can edit this page to suit your needs. For instance, here we have a no copyright statement, a colophon and some other information. This page is based on the corresponding page of Ken Arroyo Ohori's thesis, with minimal changes.

CC BY-NC-SA

 This book is released into the public domain using the CC BY-NC-SA. This license enables reusers to distribute, remix, adapt, and build upon the material in any medium or format for noncommercial purposes only, and only so long as attribution is given to the creator. If you remix, adapt, or build upon the material, you must license the modified material under identical terms.

To view a copy of the CC BY-NC-SA code, visit:

<https://creativecommons.org/licenses/by-nc-sa/4.0/>

Colophon

This document was typeset with the help of KOMA-Script and L^AT_EX using the kaobook class.

9

Visualización

Los sistemas gráficos suelen generar como resultado final una imagen en algún dispositivo físico de visualización, normalmente en un monitor raster. Aunque hay otros tipos de dispositivos, como las pantallas estereoscópicas o los cascos de realidad virtual.

Para generar la imagen en un monitor se deben transformar las coordenadas de la escena al sistema de coordenadas del dispositivo, que es bidimensional y discreto.

Cuando el modelo es 2D esta transformación se puede especificar dando el rectángulo de la escena a visualizar (llamado *ventana*) y el del monitor en el que se va a generar la imagen (denominada *viewport*), como se muestra en la figura 9.1.

Cuando la escena es 3D el proceso es mas complejo. Conceptualmente la imagen se genera como si tuviésemos una cámara virtual en el mundo 3D, que se especifica dado su posición, orientación y focal. La imagen que se genera es la proyección del mundo en un rectángulo (el plano de proyección de la cámara virtual, que en este caso sería la *ventana*). Esta proyección se transforma al *viewport* del dispositivo de salida (figura 9.2).

Las operaciones que hay que realizar para generar la imagen de una escena son:

- Transformación de visualización: Transformar coordenadas 3D de los objetos a coordenadas de pantalla.
- Recortado: Eliminación de partes de polígonos fuera de la zona visible.
- Rasterización y eliminación de partes ocultas: Determinar los pixeles cubiertos por cada primitiva.
- Iluminación y texturización: Calcular el color de cada pixel.

Estos pasos pueden realizarse en diferente orden.

9.1. Cauce gráfico

El término cauce gráfico (*graphics pipeline*) hace referencia al conjunto de pasos que se realizan para visualizar polígonos. El procesamiento se realiza en tres fases (figura 9.3):

- Operaciones con vértices.
- Recortado y rasterización.
- Operaciones con fragmentos.

La primera y tercera fases son programables, mientras que la intermedia tiene una funcionalidad fija (figura 9.3).

En las fases programables (operaciones con vértices y con fragmentos) se realizan las transformaciones, la iluminación y la texturización. Ya vimos como se realiza el cálculo de iluminación (Capítulo 5), y que se

| | |
|--|----|
| 9.1 Cauce gráfico | 79 |
| 9.1.1 Transformación de visualización | 80 |
| 9.1.2 Proyección | 81 |
| 9.1.3 Recortado | 82 |
| 9.1.4 Rasterización | 82 |
| 9.1.5 Transformación de viewport | 82 |
| 9.2 Visualización en OpenGL | 82 |
| 9.2.1 Proyección | 83 |
| 9.2.2 Viewport | 83 |
| 9.2.3 Transformaciones de modelado y vista | 84 |
| 9.2.4 Cámara orbital | 84 |
| 9.2.5 Cámara primera persona | 85 |
| 9.3 Cámara en Unity | 86 |
| 9.4 Ejercicios | 86 |



Figura 9.1: La transformación de visualización en 2D transforma la ventana en el viewport.



Figura 9.2: Cámara virtual en la visualización de un escenario 3D.

puede realizar a nivel de polígono, de vértice o de pixel. Cuando se realiza por polígono o vértice se calcula después de la transformación de visualización como último paso de las operaciones con vértices. Si se realiza a nivel de pixel se calcula en las operaciones con fragmentos. En cualquier caso, la aplicación de textura se debe realizar al final del pipeline en las operaciones con fragmentos.

9.1.1. Transformación de visualización

Al visualizar la escena las coordenadas de los objetos sufren diferentes transformaciones geométricas. En primer lugar se transforman para ubicarlos en la escena. A continuación se deben transformar para calcular como se ven desde la cámara. Finalmente se transformar a coordenadas de dispositivo (figura 9.4).

A lo largo de este proceso, existen diferentes sistemas de coordenadas que organizan y definen la representación de los objetos en la escena¹:

- **Coordenadas de objeto (Object Space)** Este es el sistema de coordenadas local de cada objeto. En este espacio, las coordenadas de los vértices se definen en relación con el origen del propio objeto, que suele ser el centro, un vértice u otro punto relevante del objeto (por ejemplo el centro de la base).
- **Coordenadas del mundo (World Space)** Las coordenadas del mundo son un sistema de referencia global en el que se posicionan todos los objetos de la escena. Cada objeto tiene su ubicación, escala y rotación definidas en este espacio. La primera transformación que se realiza es la **transformación de modelado**, que convierte las coordenadas de objeto en coordenadas del mundo.
- **Coordenadas de cámara (View Space)** En el espacio de la cámara ésta está en el origen de las coordenadas, las coordenadas de los objetos están expresadas en función de su posición respecto de la cámara. Para determinar cómo se ve la escena desde un punto de vista concreto, las coordenadas del mundo se transforman mediante la **transformación de vista** en coordenadas de cámara. En el sistema de coordenadas de la cámara el eje Z suele estar invertido para que la coordenada Z coincida con la profundidad.
- **Coordenadas normalizadas (Normalized Coordinates)** Sistema de coordenadas 3D en el que el espacio visualizado se representa dentro de un cubo normalizado. La conversión de coordenadas de cámara a coordenadas normalizadas se realiza mediante la **transformación de proyección**.
- **Coordenadas de dispositivo (Device Space)** Sistema de coordenadas 2D del dispositivo de visualización. La conversión de coordenadas normalizadas a coordenadas de dispositivo se realiza mediante la **transformación de viewport**.

Las transformaciones de modelado y vista realizan escalados, rotaciones y traslaciones. Las transformaciones que se debe aplicar para realizar la transformación de vista se pueden calcular a partir de la posición y orientación de la cámara en el sistema de coordenadas del mundo, que se suele dar con los siguientes parámetros:

PRP (Projection reference point) Punto del espacio foco de la proyección, donde esta situada la cámara.

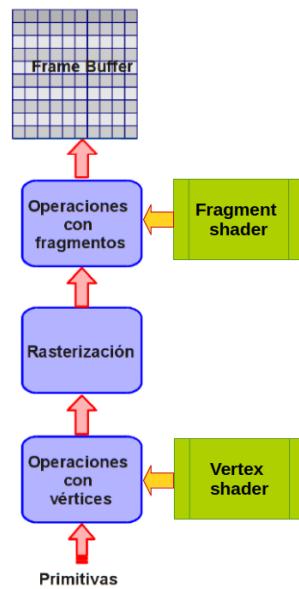


Figura 9.3: Cauce gráfico en GPU programable.

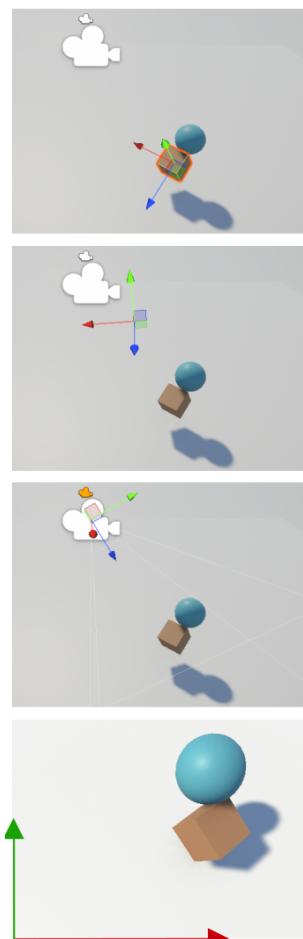


Figura 9.4: Principales sistemas de coordenadas usados en un sistema gráfico. De arriba a abajo: coordenadas de objeto, Coordenadas del mundo, Coordenadas de cámara y Coordenadas de dispositivo.

VPN (View plane normal) Vector perpendicular al plano de proyección.

VRP (View reference point) Indica el punto de interés, o target, que es el punto que se proyecta en el centro de la imagen. Podemos pensar en él como el punto al que se mira ("look at point").

VUP (Vector vertical) Dirección vertical de la cámara.

La transformación de vista transforma los vértices de manera que PRP queda en el origen, VPN queda alineado con el eje Z y el vector VUP es vertical (9.5).

Un sistema de coordenadas está definido por tres vectores ortogonales, como quiera que los vectores introducidos para definir la cámara pueden no ser ortogonales, es necesario generar tres vectores ortogonales: c_x, c_y y c_z que, junto con la posición de la cámara, forman el **sistema de referencia de la cámara**:

$$c_z = \frac{\text{VPN}}{\|\text{VPN}\|} \quad (\text{eje Z paralelo a VPN})$$

$$c_x = \frac{\text{VPN} \times \text{VUP}}{\|\text{VPN} \times \text{VUP}\|} \quad (\text{eje X perpendicular a VPN y VUP})$$

$$c_y = c_z \times c_x \quad (\text{eje Y perpendicular a los otros dos})$$

La transformación de vista se puede construir a partir de estos tres vectores como composición de las siguientes transformaciones (figura 9.6):

- Traslación de $-PRP$, haciendo que el origen de coordenadas este en la cámara.
- Rotar respecto al eje X para llevar el vector VPN al plano XZ.
- Rotar respecto al eje Y para llevar VPN al eje $-Z$.
- Rotación respecto al eje Z para llevar a c_y al eje Y.

9.1.2. Proyección

La proyección se puede realizar en perspectiva o paralela (u ortográfica). La proyección en perspectiva es similar al proceso que ocurre en las cámaras fotográficas cuando la escena se proyecta en el CCD. En una proyección en perspectiva el tamaño de los objetos en la imagen disminuye con la distancia a la cámara.

En una proyección ortográfica las líneas de proyección son paralelas al plano de proyección, por lo que el tamaño de los objetos en la imagen es independiente de la profundidad.

El volumen de visión (espacio de la escena que se muestra en la imagen) es un paralelepípedo en el caso de la proyección paralela y un tronco de pirámide de base rectangular para la proyección en perspectiva (figura 9.7).

La figura 9.8 muestra un esquema 2D del cálculo de la transformación de perspectiva². Es fácil comprobar, por semejanza de triángulos que

$$y'/near = y/z$$

- 1: Internamente se definen algunos sistemas de coordenadas adicionales para facilitar la realización del recortado y la eliminación de partes ocultas.

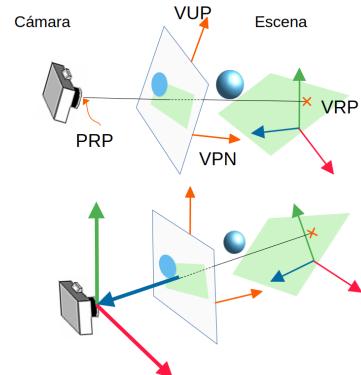


Figura 9.5: Transformación de vista. Arriba parámetros de la cámara en sistema de coordenadas del mundo. Abajo sistema de coordenadas de cámara.

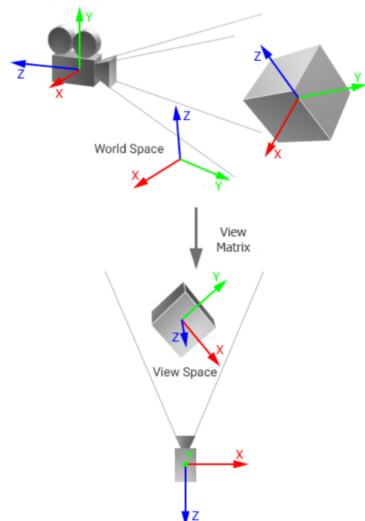


Figura 9.6: Sistemas de coordenadas del mundo y de vista.

2: https://learnwebgl.brown37.net/08_projections/projections_perspective.html

por tanto

$$y' = y \cdot \text{near}/z$$

La transformación se representa como una transformación en coordenadas homogéneas por una matriz con valores no nulos en la última fila, que puede generar un valor de w' distinto de 1.

9.1.3. Recortado

En el recortado se elimina la geometría que no es visible teniendo en cuenta los parámetros de la cámara. Para ello se descartan los polígonos que están totalmente fuera del volumen de visión (*view frustum culling* figura 9.9) y se recortan los que son parcialmente visibles (figura 9.10). Esto es, se calcula y triangula la parte visible de los que cruzan el volumen de visión, descartando el resto. El recortado simplifica los algoritmos de las fases siguientes y reduce el volumen de información que es necesario procesar en ellas.

9.1.4. Rasterización

La rasterización es el proceso de discretización de las primitivas en pixels. La discretización se realiza por hardware procesando los triángulos por líneas de barrido. Un pixel está cubierto por un triángulo si su centro se encuentra dentro del triángulo.

La rasterización puede producir bordes dentados (pixelados) cuando hay un gradiente grande en la imagen. Para evitarlo se pueden usar técnicas de antialiasing.

9.1.5. Transformación de viewport

El término *viewport* hace referencia a la zona rectangular de la ventana donde se dibuja la escena. La transformación de *viewport* es lineal y consta simplemente de escalados y traslaciones.

9.2. Visualización en OpenGL

OpenGL almacena en su estado dos pilas de matrices de transformación. En una almacena la concatenación de las matrices de modelado y vista, en la otra la transformación de proyección. Se puede seleccionar a cual de las dos matrices se le concaténan las transformaciones que se den con la orden *glMatrixMode*:

```
glMatrixMode(GL_MODELVIEW);
glMatrixMode(GL_PROJECTION);
```

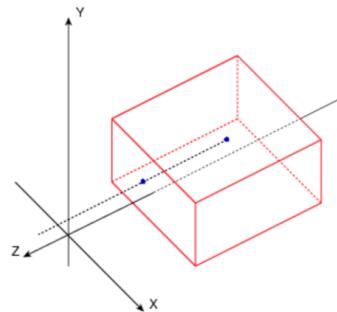
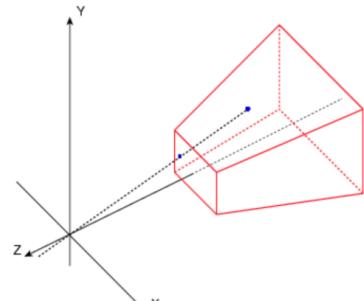


Figura 9.7: Volumen de visión: proyección perspectiva arriba y ortográfica abajo.

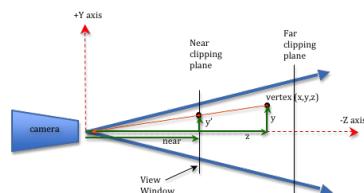


Figura 9.8: Esquema de la transformación de perspectiva.

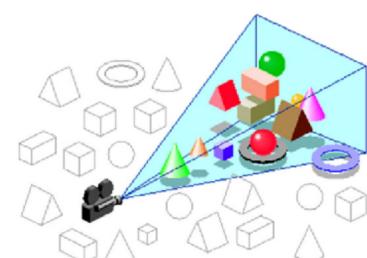


Figura 9.9: View frustum culling. Los objetos que están totalmente fuera del volumen de visión son descartados.

9.2.1. Proyección

La región en forma de pirámide truncada contendrá los objetos de la escena visibles en la imagen, se llama *view-frustum*, y está determinada por estos 6 valores reales (figura 9.11):

n, f = (*near* y *far*). Coordenadas Z (negadas) de la cara delantera y trasera del frustum, determinan su extensión en Z. Se cumple $0 < n < f$. Si z está en el frustum, $-f \leq z \leq -n$ (los objetos visibles están en $Z-$).

l, r = (*left* y *right*). Coordenadas x de la arista vertical izquierda y derecha respectivamente, de la cara delantera del frustum (la más cercana al observador).

b, t = (*bottom* y *top*). Coordenadas y de la arista inferior y superior, respectivamente, de la cara delantera del frustum.

Para usar una transformación de perspectiva se utiliza la función *glFrustum*:

```
glFrustum( GLdouble l, GLdouble r, GLdouble b, GLdouble t,
           GLdouble n, GLdouble f ) ;
```

Si queremos una proyección ortográfica, debemos usar *glOrtho* con los mismos parámetros:

```
glOrtho( GLdouble l, GLdouble r, GLdouble b, GLdouble t,
          GLdouble n, GLdouble f ) ;
```

Estas llamadas generan la matriz de transformación y la componen con la matriz de proyección existente.

Para proyecciones en perspectiva se puede usar la función *gluPerspective* como alternativa (figura 9.12):

```
gluPerspective(GLdouble alfa, GLdouble a, GLdouble n, GLdouble f);
```

esta función equivale a un *glFrustum* con $r = -l$ y $t = -b$, con:

α es la apertura vertical, en grados, del campo de visión. Con valores entre 0 y 180.0).

a es la relación de aspecto de la imagen (ancho dividido por alto).

9.2.2. Viewport

En cualquier momento es posible cambiar la configuración del viewport que OpenGL almacena como parte de su estado llamando a la función *glViewport*:

```
glViewport( GLint xl, GLint yb, GLsizei w, GLsizei h);
```

donde (xl, yb) es la posición (columna y fila) del pixel que ocupa, en la ventana del dispositivo de salida, la esquina inferior izquierda del viewport. Los valores w y h indican la anchura y altura en pixel del viewport.

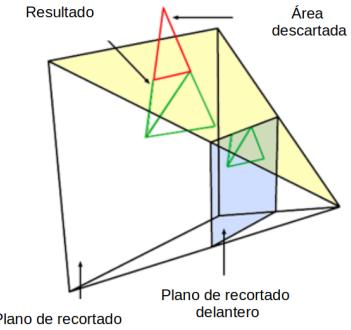


Figura 9.10: Recortado. Se calcula la parte del polígono que está dentro del volumen de visión, que es triángulada (triángulos verdes).

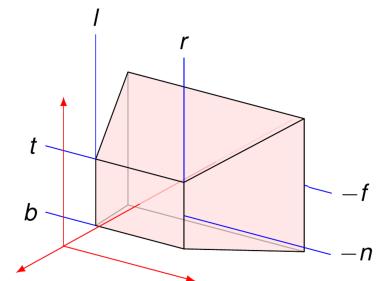


Figura 9.11: Parámetros de definición del frustum.

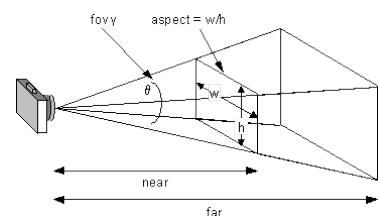


Figura 9.12: Parámetros de definición del frustum con *gluPerspective*.

9.2.3. Transformaciones de modelado y vista

La matriz *modelview* se puede especificar en OpenGL mediante estos pasos:

1. Hacer la llamada *glMatrixMode(GL_MODELVIEW)*, para indicar que las siguientes operaciones actúan sobre la matriz *modelview*.
2. Usar *glLoadIdentity* para hacer que la matriz *Modelview* sea la identidad.
3. Indicar la matriz de vista.
4. Usar una (o varias) llamadas a las funciones *glScale*, *glRotate*, o *glTranslate* para componer la matriz de modelado.

OpenGL construye *modelview* componiendo las matrices que se le proporcionan en los pasos 3 y 4.

La matriz de vista se puede dar indicando explícitamente la transformación del sistema de coordenadas del mundo al de la cámara usando rotaciones y traslaciones.

Alternativamente se puede usar la función *gluLookAt* (de la librería GLU) que permite componer una matriz de vista de forma muy cómoda, ya que acepta directamente los valores de *PRP* (posición de la cámara), *VRP* (target) y *VUP* (vertical de la cámara) como parámetros. Está declarada como sigue:

```
void gluLookAt( GLdouble PRPx, GLdouble PRPy, GLdouble PRPz,
                 GLdouble VRPx, GLdouble VRPy, GLdouble VRPz,
                 GLdouble VUPx, GLdouble VUPy, GLdouble VUPz);
```

El vector *VPN* se calcula como *VRP – PRP*³.

3: Esto hace que no se puedan definir proyecciones oblicuas.

9.2.4. Cámara orbital

Cámara orbital en OpenGL se puede definir usando ángulos de Euler:

```
void dibujoEscena() {
    ...
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    glTranslatef( 0, 0, -D );
    glRotatef( view_rotx, 1.0, 0.0, 0.0 ); // Rotaciones de la
        camara
    glRotatef( view_roty, 0.0, 1.0, 0.0 );
    glRotatef( view_rotz, 0.0, 0.0, 1.0 );
    glTranslatef(PRPx,PRPy,PRPz);
    ...
}
```

donde *PRP* es el centro de atención, *view_rot* son los ángulos de Euler y *D* es la distancia de la cámara al centro de atención.

La interacción se puede realizar con teclado:

```
static void especial(int k, int x, int y) {
    switch (k) {
        case GLUT_KEY_UP:
            view_rotx += 5.0;
            break;
        case GLUT_KEY_DOWN:
            view_rotx -= 5.0;
```

```

        break;
    case GLUT_KEY_LEFT:
        view_roty += 5.0;
        break;
    case GLUT_KEY_RIGHT:
        view_roty -= 5.0;
        break;
    default:
        return;
    }
    glutPostRedisplay();
}

```

o con ratón:

```

void clickRaton( int boton, int estado, int x, int y )
{
    if(boton== GLUT_MIDDLE_BUTTON && estado == GLUT_DOWN) {
        Xref=x;           // Almacena posición (en coor. de pantalla)
        Yref=y;
        RatonPulsado=GL_TRUE;
    }
    else RatonPulsado=GL_FALSE;
    ...

}

void RatonMovido( int x, int y )
{
    if(RatonPulsado) {      // Si el ratón está pulsado
        view_roty -= (Yref -y) / 100.0;    // Cambia offset de imagen
        view_rotx += (Xref -x) / 100.0;
        Xref=x;                     // Actualiza última posición de ratón
        Yref=y;
        glutPostRedisplay();
    }
    ...
}

```

Alternativamente usando *glutLookAt* se puede crear como.

```

void dibujoEscena() {
    ...
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    gluLookAt(PRPx, PRPy, PRPz, VRPx, VRPy, VRPz, VUPx, VUPy, VUPz);
    ...
}

```

donde *VRP* es el centro de atención, *VUP* el vector hacia arriba y *PRP* la posición de la cámara.

9.2.5. Cámara primera persona

La cámara está en la posición de un avatar que se mueve en el escenario. El usuario puede controlar la dirección en la que mira la cámara.

La dirección se puede especificar como dos giros en direcciones ortogonales a la dirección de avance del personaje (con las mismas fórmulas de la posición del caso anterior pero ahora usadas para calcular *VPN*).

$$VPN = (\sin(\phi) \sin(\theta), \cos(\phi), \sin(\phi) \cos(\theta))$$

VUP se debe calcular de forma que no sea perpendicular al plano proyección, por ejemplo haciendo:

$$VUP = ((0, 1, 0) \times VPN) \times VPN$$

Además es conveniente hacer que la orientación de la cámara se ajuste a la dirección de movimiento del personaje. Para ello debemos componer dos giros de la cámara: alinear con la dirección de movimiento del personaje y el indicado por el usuario.

9.3. Cámara en Unity

La cámara es un objeto del grafo de escena. Modificamos sus parámetros en el inspector. Para que la cámara siga a un componente de la escena basta con incluirla en el grafo de escena dependiendo de ese componente (figura 9.13).

Una cámara en tercera persona sigue la posición de un avatar que se mueve en el escenario a una distancia fija de él. El usuario puede controlar la dirección en la que mira la cámara. Ahora la posición de la cámara (\vec{P}_c) será la del personaje menos una cantidad (d) por su dirección de movimiento \vec{D}_p

$$\vec{P}_c = d \vec{D}_p$$

d indica la distancia de la cámara al personaje.

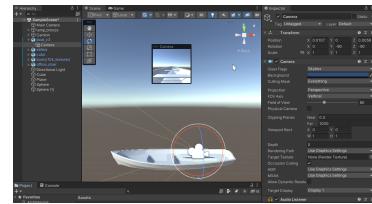


Figura 9.13: Cámara en primera persona en Unity.

9.4. Ejercicios

1. Redacta un algoritmo para hacer view frustum culling en un sistema con proyección ortogonal.
2. Dado un cubo unidad situado con un vértice en el origen del sistema de coordenadas y otro en el punto (1,1,1), indicar los parámetros y transformaciones a usar para visualizarlo desde las siguientes posiciones:
 - Desde el eje Y, a una distancia de 5 unidades mirando hacia el origen.
 - Desde la bisectriz del octante con X negativo e Y,Z positivos, mirando hacia el centro del cubo.
3. Describir la imagen que se generaría en cada uno de los casos anteriores.
4. Redactar procedimientos que efectúen los siguientes recorridos de cámara:
 - Recorrido circular de radio 5 unidades mirando hacia el origen sobre el eje y sobre el plano y=0.

- Recorrido en forma de escalera de caracol, de radio 5 unidades, partiendo de una altura de $y=5$ hasta $y=-5$, mirando hacia el origen.
 - Alejamiento de la cámara, mirando desde la bisectriz del primer octante al origen de coordenadas (comenzando en una distancia de 5 y finalizando a una distancia de 20).
5. Implementar un algoritmo para realizar un efecto zoom en OpenGL.
 6. Describir un algoritmo que simule una caída en barrena vertical en OpenGL. El observador siempre mira al origen de coordenadas, partiendo de una altura $y=20$ hasta $y=1$, haciendo que la cámara gire mientras desciende.
 7. Para definir una vista se puede utilizar el siguiente conjunto de parámetros:
 - PCamara: Posición del observador.
 - PAtencion: Posición a la que mira.
 - Apertura: Angulo de apertura del cono de visión.
 - Spin: Angulo de giro de cámara respecto a la vertical.
 Indicar como se fijarían los parámetros de vista de OpenGL a partir de estos.
 8. ¿Se puede realizar la transición de una proyección en perspectiva a una paralela de forma continua?
 9. Programa una cámara en tercera persona que siga a un objeto en movimiento permitiendo que el usuario ajuste la distancia de la cámara al objeto y la orientación de la cámara.
 10. Crea un efecto de oscilación de cámara como si el observador caminara. La cámara debe moverse ligeramente hacia arriba y abajo mientras se traslada hacia adelante.
 11. Implementa una cámara que simule una cámara de seguridad, que gire automáticamente para seguir un objeto del escenario.
 12. Diseña un sistema que permita indicar interactivamente posiciones en el escenario. Cuando se introduzca una posición la cámara debe avanzar de forma continua hasta ese punto a una velocidad prefijada.
 13. Crea una visualización con dos viewport de forma que se simule la visualización estereoscópica para gafas de realidad virtual. La posición de la cámara en las vistas deben estar ligeramente desplazada.

Informática Gráfica

Juan Carlos Torres

Curso 2024/25

Dpto. Lenguajes y Sistemas Informáticos
Universidad de Granada

Disclaimer

You can edit this page to suit your needs. For instance, here we have a no copyright statement, a colophon and some other information. This page is based on the corresponding page of Ken Arroyo Ohori's thesis, with minimal changes.

CC BY-NC-SA

 This book is released into the public domain using the CC BY-NC-SA. This license enables reusers to distribute, remix, adapt, and build upon the material in any medium or format for noncommercial purposes only, and only so long as attribution is given to the creator. If you remix, adapt, or build upon the material, you must license the modified material under identical terms.

To view a copy of the CC BY-NC-SA code, visit:

<https://creativecommons.org/licenses/by-nc-sa/4.0/>

Colophon

This document was typeset with the help of KOMA-Script and L^AT_EX using the kaobook class.

10

Conejividad en mallas

La conectividad (o topología) de una malla describe cómo están conectados los elementos geométricos entre sí (p.e. que polígonos son vecinos de cada polígono). La información de conectividad es crucial porque define la estructura del objeto, permitiendo realizar operaciones como deformaciones, subdivisiones y otras transformaciones geométricas que son esenciales en la animación y modelado 3D de forma eficiente. La representación de sopa de triángulos vista en el capítulo 2 es muy simple pero tiene muchas limitaciones por no almacenar información de conectividad (Figura 10.1).

La mayor parte de las operaciones requieren acceder a los elementos vecinos, permitiendo que los algoritmos avancen (o naveguen) por la superficie. El que estas relaciones estén almacenadas de forma explícita en la representación influye en la eficiencia de los algoritmos.

A modo de ejemplo, pensemos que queremos saber si una malla es abierta o cerrada¹. Si utilizamos la representación indexada cara-vértice (Sección 2.4) podemos usar un algoritmo iterativo que para cada par de vértices consecutivos de cada cara busque si hay otra cara que los contenga en orden inverso. Este algoritmo tiene complejidad cuadrática con el número de caras.

Algunas de las operaciones usadas frecuentemente en algoritmos que trabajan con mallas de polígonos y que requieren información de conectividad son [Bot+10]:

- a) Acceso ordenado a vértices de una cara.
- b) Acceso a caras que inciden en una arista.
- c) Acceder a los vértices de una arista.
- d) Acceder a las caras de un vértice.

Para facilitar la navegación por la malla se puede incluir en la representación información de adyacencia entre elementos, normalmente como enlaces entre elementos:

- cara-cara: Cada cara indexa a sus caras vecinas
- cara-vértice: Esta es la única relación almacenada en la estructura indexada vista en la sección 2.4.
- cara-arista: Se almacenan explícitamente las aristas y cada cara tiene el identificador de sus aristas.
- arista-cara: Cada arista indexa sus dos caras. Si la malla es abierta puede tener solo una cara.
- arista-vértice: Cada arista indexa sus dos vértices.
- arista-arista: Cada arista indexa las aristas que continúan a partir de sus dos vértices.
- vértice-vértice: Cada vértice almacena la dirección de sus vértices vecinos (a través de las aristas).
- vértice-arista: Cada vértice indexa sus aristas.
- vértice-cara: Cada vértice indexa las caras que lo contienen.

| | | |
|--------|---|----|
| 10.1 | Representación de mallas | 90 |
| 10.1.1 | Aristas aladas | 90 |
| 10.1.2 | Normales de vértice en superficies discontinuas | 91 |
| 10.1.3 | Coordenadas de textura | 92 |
| 10.1.4 | Semiaristas aladas | 92 |
| 10.2 | Problemas topológicos . | 92 |
| 10.3 | Creación de mallas por barrido | 93 |
| 10.4 | Mallas indexadas en OpenGL | 94 |
| 10.5 | Ejercicios | 95 |



Figura 10.1: Es fácil hacer sopas, pero no le gustan a todo el mundo.

1: Una malla es cerrada si todos los polígonos tienen un polígono vecino a través de cada arista

Cada relación almacenada facilita la navegación en la malla a costa de aumentar el espacio ocupado y la complejidad de algunas operaciones, especialmente las que editan la malla. No es por tanto necesario, ni conveniente almacenarlas todas, siempre que las operaciones necesarias se puedan realizar con un número fijo de accesos.

Grafos de navegación

El grafo de navegación de un representación es un diagrama que muestra las relaciones de adyacencia que están presentes en ella. En él aparecen los tres elementos geométricos de la malla: Vértices (Vertices), Aristas (Edges) y Caras (Faces). Un arco en el grafo indica que la relación está almacenada. La figura 10.2 muestra el grafo de navegación completo incluyendo todas las posibles relaciones.

La representación de vértices indexados descrita en el listado 2.3 no contienen aristas, su grafo de navegación es el que se muestra en la Figura 10.3.

10.1. Representación de mallas

Introducir las aristas en la estructura es simple: las caras almacenan una lista de aristas, y las aristas sus dos vértices. Accedemos a los vértices a través de las aristas (Figura 10.4). Pero esto no nos permite resolver las operaciones b) y d) en tiempo constante

10.1.1. Aristas aladas

Una de las formas mas usadas de representar mallas de polígonos es mediante la estructura de aristas aladas. El elemento central en ella son las aristas. De cada cara se almacena solamente el identificador de una de sus aristas. Para cada arista se almacena, además de sus vértices (V_i y V_f), las dos caras que separa la arista (C_d y C_i), y las aristas adyacentes en cada una de estas caras (Figura 10.6). Esta estructura es particularmente útil para algoritmos que necesitan navegar rápidamente a través de la malla, como aquellos usados en refinamiento de mallas, detección de colisiones, o aplicación de texturas.

Asumiendo una ordenación CCW de las caras, las caras directa e inversa se identifican por el sentido de recorrido de la arista. En la cara directa (C_d) la arista se recorre tal como está definida en la estructura (de V_i a V_f). En la cara inversa se recorre en sentido contrario.

El grafo de navegación de la estructura se muestra en la Figura 10.8. Podemos observar como es posible navegar desde cualquier entre cualquier par de elementos bien de forma directa o indirectamente, pasando por las aristas. Con esta estructura es posible realizar las operaciones b) y c) con un acceso a la estructura y la d) con dos accesos.

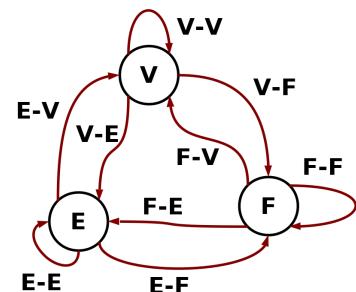


Figura 10.2: Grafo de navegación mostrando todas las posibles relaciones.

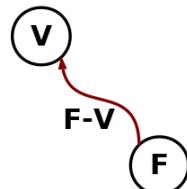


Figura 10.3: Grafo de navegación de una representación de vértices indexados.

| Cara | Aristas |
|------|---------|
| 1 | 0 |
| 2 | 4 |
| 3 | 5 |

| Aristas | V_i | V_f |
|---------|-------|-------|
| 0 | 0 | 1 |
| 1 | 1 | 2 |
| 2 | 2 | 3 |
| 3 | 3 | 0 |
| 4 | 1 | 5 |
| 5 | 5 | 4 |
| 6 | 4 | 2 |

| Vértice | X | Y | Z |
|---------|---|---|----|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 2 | -1 |
| 2 | 0 | 3 | 2 |
| 3 | 0 | 1 | 2 |
| 4 | 2 | 4 | 1 |
| 5 | 2 | 3 | -1 |

Figura 10.4: Estructura caras-aristas-vértices.

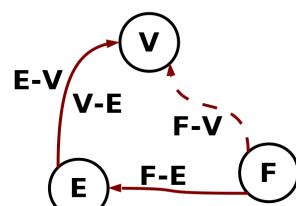


Figura 10.5: Grafo de navegación de la estructura caras-aristas-vértices.

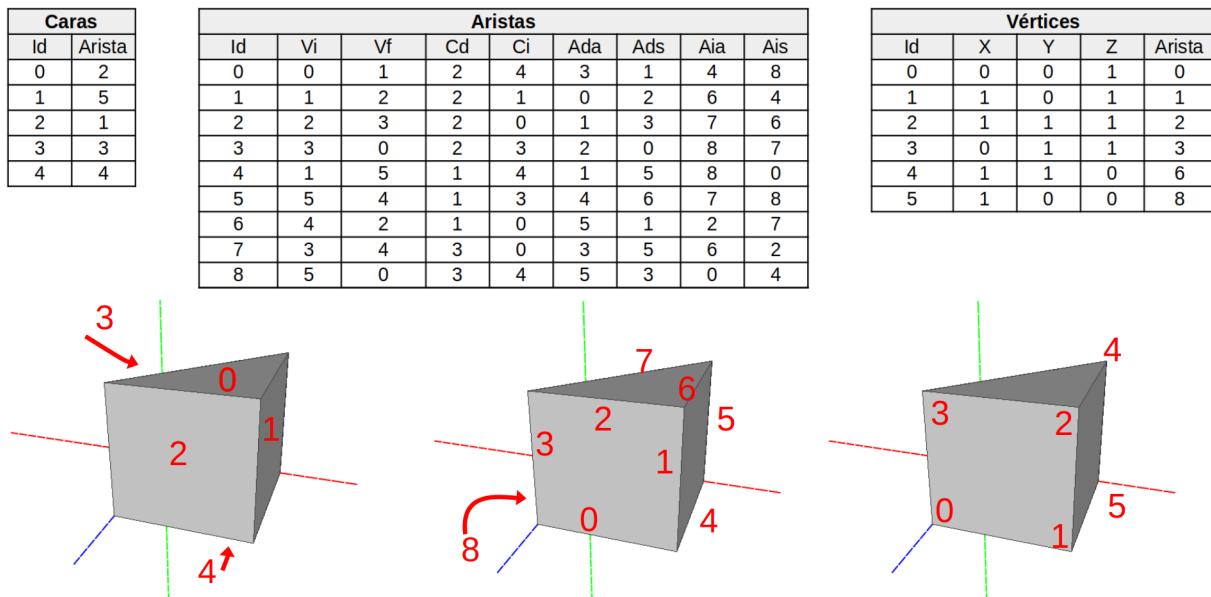


Figura 10.7: Estructura aristas aladas, rellenada con los datos de un prisma triangular. En la imagen se muestran de izquierda a derecha los identificadores de caras, aristas y vértices.

Podemos crear una estructura de aristas aladas a partir una sopa de polígonos, creando la tabla de aristas cuando se están leyendo las caras:

```

1 Para cada vertice
2   Agregar a la tabla de vertices
3 Para cada cara
4   Para cada par de vertices (Va,Vb)
5     Si existe la arista (Vb,Vi)
6       Agregar cara como cara inversa
7     si no
8       Agregar arista a la lista de aristas
9       Agregar cara como cara directa
10  Agregar cara a lista de caras
11  Recorrer aristas y llenar enlaces a arista siguiente

```

Para obtener los vértices de una cara se deben realizar $n+1$ acceso, siendo n el número de vértices. Una vez que accedemos a la primera arista de la cara tendremos que comprobar si la cara que buscamos es la cara directa o inversa de la arista, para obtener los vértices V_i y V_f en el orden correcto y encontrar la arista siguiente de la cara (siguiendo los enlaces A_{ds} o A_{is}).

10.1.2. Normales de vértice en superficies discontinuas

En la sección 2.3.1 vimos que se puede asociar una normal a los vértices calculándola como un promedio de las normales en las caras adyacentes. Esto nos permite calcular una normal que genera una transición suave en la superficie, y por tanto es adecuada para superficies continuas. La Figura 10.9 muestra dos visualizaciones del mismo cilindro. El de la izquierda se ha dibujado con normales de vértice calculadas como promedio de las normales de cara. Puede observarse como desaparecen las discontinuidades en las aristas.

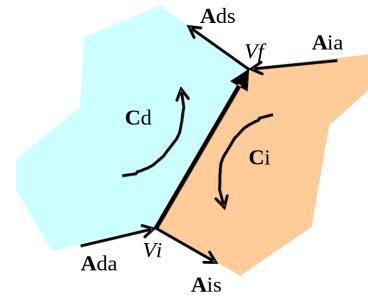


Figura 10.6: Información asociada a las aristas en una representación de aristas aladas.

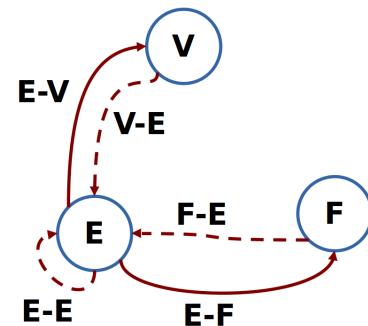


Figura 10.8: Grafo de navegación de una estructura de aristas aladas.

Cuando se quiere que en la superficie de un objeto aparezcan discontinuidades en la orientación de la superficie es necesario utilizar mas de un valor en el vértice.

Esto puede conseguirse, duplicando vértices, o haciendo que la representación del objeto esté formado por mas de una superficie o modificando la estructura de datos para que cada vértice pueda tener mas de una normal.

10.1.3. Coordenadas de textura

Un problema parecido aparece cuando se desenvuelve una malla para calcular coordenadas de textura. Los vértices que están en las costuras tienen mas de una coordenada de textura. MeshLab puede usar coordenadas de textura asociadas a vértice o *wedge*, en este último caso almacena para cada vértice una normal en cada una de sus caras.

10.1.4. Semiaristas aladas

Esta es el principal inconveniente de esta estructura. Para evitarlo se puede usar **semiaristas aladas**. En esta estructura la información de las aristas se almacena como dos semiaristas, cada una contiene solamente la información de una cara y el identificador de su semiarista compañera (figura 10.11). De esta forma se mantiene la misma información topológica facilitando la obtención de la secuencia de aristas(o vértices) de las caras.

En una estructura de semiaristas aladas La información de normales de vértice y de coordenadas de textura se puede almacenar en las aristas, permitiendo que cada vértice tenga asociados valores diferentes en cada cara. Esta información se puede asociar, por ejemplo, al vértice inicial de cada arista.

10.2. Problemas topológicos

El objetivo de guardar explícitamente información de conectividad es poder realizar de forma eficiente algunas operaciones en la malla, concretamente operaciones que requieren consultar el entorno de un elemento.

Independientemente de como se represente la malla puede tener ciertos problemas topológicos que dificultan su procesamiento. Uno de los problemas más comunes es la condición de **non-manifold** que se produce cuando una arista es compartida por más de dos caras o un vértice es compartido por dos láminas de superficie que se tocan en él (Figura 10.12). Esto hace que la malla no pueda ser desplegada en un plano sin superposiciones.

También genera problemas la no **orientabilidad** de la superficie, que impide definir un interior y un exterior consistentes; dos ejemplos clásicos son la banda de Möbius y la botella de Klein.

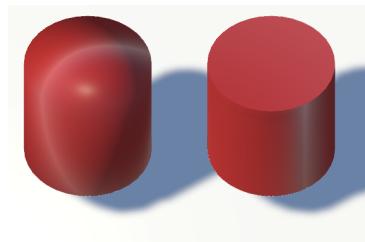


Figura 10.9: Efecto del cálculo de la normal de vértice como promedio de las normales de cara.

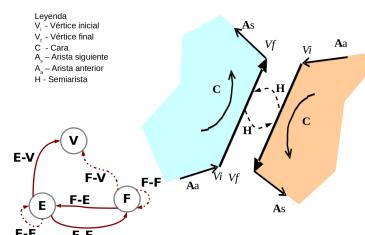


Figura 10.10: Estructura de semiaristas aladas.

| Cara | Arista |
|------|--------|
| 0 | 2 |
| 1 | 5 |
| 2 | 1 |
| 3 | 3 |
| 4 | 4 |

| Saristas | Vi | Vf | h | C | Aa | As |
|----------|----|----|----|---|----|----|
| 0 | 0 | 1 | 9 | 2 | 3 | 1 |
| 1 | 1 | 2 | 10 | 2 | 0 | 2 |
| 2 | 2 | 3 | 11 | 2 | 1 | 3 |
| 3 | 3 | 0 | 12 | 2 | 2 | 0 |
| 4 | 1 | 5 | 13 | 1 | 1 | 5 |
| 5 | 5 | 4 | 14 | 1 | 4 | 6 |
| 6 | 4 | 2 | 15 | 1 | 5 | 1 |
| 7 | 3 | 4 | 16 | 3 | 3 | 5 |
| 8 | 5 | 0 | 17 | 3 | 5 | 3 |
| 0 | 1 | 0 | 0 | 4 | 4 | 8 |
| 1 | 2 | 1 | 1 | 1 | 6 | 4 |
| 2 | 3 | 2 | 2 | 0 | 7 | 6 |
| 3 | 0 | 3 | 3 | 3 | 8 | 7 |
| 4 | 5 | 1 | 4 | 4 | 8 | 0 |
| 5 | 4 | 5 | 5 | 3 | 7 | 8 |
| 6 | 2 | 4 | 6 | 0 | 2 | 7 |
| 7 | 4 | 3 | 7 | 0 | 6 | 2 |
| 8 | 0 | 5 | 8 | 4 | 0 | 4 |

| Vértice | X | Y | Z |
|---------|---|---|----|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 2 | -1 |
| 2 | 0 | 3 | 2 |
| 3 | 0 | 1 | 2 |
| 4 | 2 | 4 | 1 |
| 5 | 2 | 3 | -1 |

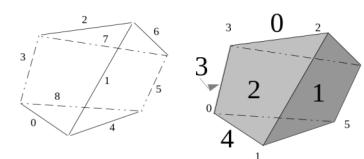


Figura 10.11: Estructura de semiaristas aladas para los modelos de la figura 10.7.

Además, se pueden producir errores si las caras de una malla se cruzan entre sí (generando **autointersecciones**). Esto también genera errores de renderizado.

Otra situación que genera problemas topológicos es la existencia de vértices en T. Un **vértice en T** es un vértice que se encuentra en mitad de una arista, haciendo que esa arista limite con dos aristas.

Estos problemas pueden aparecer como resultado del procesamiento de una malla correcta, y pueden producir desde errores en el procesamiento de la malla hasta hacer que el algoritmo aborte.

Corrección de topología con meshLab

Para corregir los problemas topológicos se puede usar MeshLab. La detección de elementos no manifold se puede realizar configurando los parámetros de dibujo de la malla (Figura 10.13).

Para corregir estos elementos se puede usar los filtros de limpieza y reparación:

Filters > Cleaning and Repairing

En este grupo hay operaciones específicas para cada uno de los problemas topológicos. Algunas de las operaciones simplemente borran (o seleccionan para que después se borren) los elementos afectados. En estos casos puede ser necesario llenar las fisuras creadas usando el filtro **Close Holes**:

Filters > Remeshing, Simplification and Reconstruction > Close Holes

10.3. Creación de mallas por barrido

Para crear la malla que representa un objeto es necesario generar las coordenadas de sus vértices e indicar como se agrupan formando polígonos. Cuando el objeto tiene simetría podemos generar esta información a partir de un perfil de la superficie utilizando la ecuación de la simetría para generar el resto del modelo (barriendo el espacio con el perfil).

Si el barrido se realiza girando el perfil se obtiene una superficie con simetría de revolución. La Figura 10.14 muestra dos ejemplos de superficies creadas por revolución de un perfil, haciéndolo girar respecto al eje Y. En la parte superior el perfil es una circunferencia y la superficie generada es un toro. La imagen inferior es una forma cerrada creada por una curva abierta que toca al eje Y en sus extremos.

Si se quiere crear la malla de revolución utilizando el eje Y como eje de simetría y se parte de un perfil contenido en el plano Z = 0 como el de la parte superior de la Figura 10.15 formado por cuatro vértices, crearemos varias copias del perfil rotándolo respecto al eje Y:

$$V_{ik} \cdot x = V_{0k} \cdot x \cos(2\pi i / N)$$

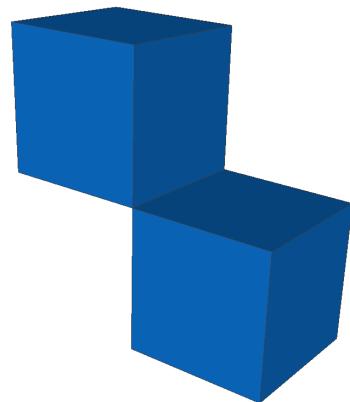


Figura 10.12: Ejemplo de objeto no manifold.

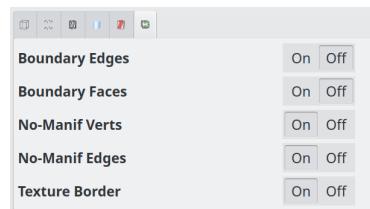


Figura 10.13: Detección de elementos no manifold en MeshLab.

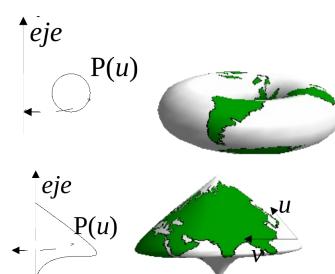


Figura 10.14: Ejemplos de objetos creados por revolución.

$$V_{ik}.z = V_{0k}.x \sin(2\pi i/N)$$

siendo N el número de perfiles utilizados.

Una vez creada la lista de vértices se construyen los elementos (cuadráteros o triángulos) uniendo los vértices de perfiles consecutivos. Si construimos triángulos podemos formar los siguientes²:

$$(V_{ik}, V_{(i-1)k}, V_{i(k+1)}) \text{ y } (V_{(i-1)k}, V_{i(k+1)}, V_{(i-1)(+1)}) \quad i \in (1, N), k \in (0, M - 1)$$

En el cierre de la malla se puede no duplicar los vértices del perfil original, utilizando como valor k en la ecuación anterior $k \% (M - 1)$.

El calculo de los vectores normales de vértices se puede realizar como se vio en el capítulo anterior 2.3.1 o calculándolas a partir de la rotación perfil lo que asegura que se correspondan con la superficie de revolución que se pretende aproximar. La normal del perfil de partida es:

$$(N_{0k}.x = V_{0(k-1)}, V_{0(k+1)}) \text{ y } (V_{(i-1)k}, V_{i(k+1)}, V_{(i-1)(+1)}) \quad i \in (1, N), k \in (0, M - 1)$$

El mismo proceso se puede seguir para crear mallas por barrido lineal, esto es, trasladando el perfil a lo largo de un vector. En la Figura 10.16 se ha creado una superficie cilíndrica trasladando la circunferencia $P(u)$ con el vector \mathbf{r} .

Este proceso se puede generalizar haciendo que la trayectoria que sigue el perfil sea una curva, desplazando y orientando el perfil a lo largo de la curva. La Figura 10.17 muestra la superficie creada por barrido de una circunferencia a lo largo de un perfil helicoidal³.

10.4. Mallas indexadas en OpenGL

El mecanismos de dibujo que hemos visto (`glBegin // glVertex // glEnd`) transfiere los vértices de los elementos a dibujar de la CPU a la GPU en cada frame. Esto hace que el rendimiento de la aplicación tenga un cuello de botella por la velocidad de transferencia entre ambas⁴. Desde que las GPU tienen memoria suficiente es mucho mas eficiente alojar los modelos en la GPU y realizar la transferencia de datos de la CPU solamente cuando se cambia el modelo.

El mecanismo de dibujo en OpenGL ha evolucionado en paralelo al desarrollo de las tarjetas gráficas para mejorar la eficiencia reduciendo la comunicación entre CPU y GPU, basados en transferir la información a dibujar empaquetada (usando estructuras denominadas **Vertex Buffer Objeto**, VBO).

Una revisión histórica de la evolución de estos mecanismos se puede leer en la web [The history of opengl vertex data](#).

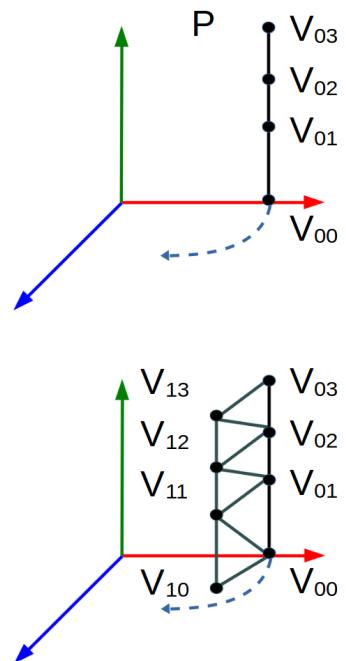


Figura 10.15: Creación de una malla de revolución.

2: Se debe tener en cuenta que el sentido de recorrido debe ser antihorario.

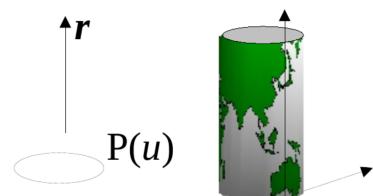


Figura 10.16: Creación de una malla por barrido lineal.

3: (<https://www.computeraideddesignguide.com/basic-tools-in-3d-in-autocad>)

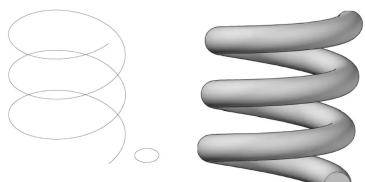


Figura 10.17: Creación de una malla por barrido lineal.

4: Obviamente esto no lo notaremos si el modelo es pequeño.

10.5. Ejercicios

1. Recorrer la estructura de la figura 10.7 para encontrar los vértices de la cara 1.
2. Calcula el espacio ocupado por estructura de aristas aladas en función del número de elementos (vértices, caras y aristas) y del espacio ocupada por cada coordenada y cada índice.
3. La formula de Euler establece una relación entre los números de caras (C), aristas (A) y vértices (V) de un objeto poliédrico

$$V - A + C = 2$$

Comprueba que se cumple en el poliedro de la figura 10.7.

4. Diseña un algoritmo para buscar fisuras en aristas aladas, que devuelva la poligonal que describe en contorno de cada fisura.
5. Diseña un algoritmo calcular normales de vértice en una estructura de aristas aladas.
6. Diseña un algoritmo que a partir de una poligonal contenida en el plano $Z = 0$ genere una malla por revolución respecto al eje Y .
7. ¿En qué influye el número de perfiles usado en una superficie de revolución?
8. La caja englobante de un objeto es el mínimo paralelepípedo alineado con los ejes que lo contiene. Escribe usando pseudocódigo un algoritmo calcular caja englobante de una malla.
9. Diseña una estructura para representar mallas de triángulos que permita localizar en tiempo constante cualquier elemento adyacente.
10. En el proceso descrito para generar mallas por revolución ¿Qué pasa si el perfil es abierto y acaba y/o empieza en el eje Y ?
11. ¿Se puede generar la superficie de la figura 10.16 como malla de revolución?
12. Diseña un algoritmo para comprobar si una malla representada usando aristas aladas es non-manifol.
13. Genera el grafo de navegación de una representación de mallas de triángulos que consta de dos lista:

Triángulos que contiene para cada triángulo sus tres vértices y sus tres triángulos vecinos ($v_0, v_1, v_2, t_0, t_1, t_2$).

Vértices que contiene las coordenadas del vértice y su normal (x, y, z, n_x, n_y, n_z).

14. Diseña un algoritmo para generar la lista de aristas de la representación anterior, contenido para cada arista los índices de sus dos vértices y sus dos caras
15. Diseña un algoritmo para determinar si la orientación de todas las caras es consistente en la representación anterior.

Informática Gráfica

Juan Carlos Torres

Curso 2024/25

Dpto. Lenguajes y Sistemas Informáticos
Universidad de Granada

11.1. Ejercicios

11.1 Ejercicios 96

1. Implementar un algoritmo para calcular la caja envolvente alineada con los ejes (AABB) de un objeto 3D dado. Visualiza el AABB en una escena 3D para comprobar que encierra completamente al objeto.
2. Implementa un algoritmo para calcular esferas envolventes. Visualízalo en las mallas.
3. ¿Como se puede comparar el ajuste de estos dos volúmenes envolventes para una escena? Propón un método para compararlos.
4. Implementa un sistema de colisión utilizando esferas como volúmenes envolventes. Simula el movimiento de dos objetos en un espacio 3D y detecta cuándo colisionan usando sus envolventes.
5. Implementa un algoritmo para crear un sistema jerárquico de volúmenes envolventes para un grafo de escena que representa un objeto. Cada nodo debe tener su propio volumen envolvente.
6. Implementa un algoritmo que optimice las detecciones de colisiones utilizando la jerarquía de volúmenes envolventes del ejercicio anterior.
7. Implementar una simplificación de malla utilizando un algoritmo de reducción de vértices. Dibuja el modelo original y el simplificado para analizar la pérdida de detalle visual.
8. Implementar un algoritmo de interpolación entre dos mallas (Morphing). Permite al usuario controlar el parámetro de interpolación para ver cómo la malla evoluciona entre los dos estados.
9. Programa la simulación de un sistema de esferas confinadas en un recinto cúbico suponiendo que la colisión no es elástica y que cada esfera rebota manteniendo su velocidad en la dirección simétrica respecto a la normal en el punto de colisión.
10. Implementar un algoritmo para calcular colisiones en el sistema anterior utilizando bisección para encontrar el tiempo exacto de colisión entre dos objetos en movimiento.
11. Crear un sistema de animación procedural que genere movimiento plausible para un péndulo usando fuerzas físicas.