Prácticas de Informática Gráfica

Grupos C y D (Curso 2024/2025)

Germán Arroyo y Juan Carlos Torres

2024-06-06

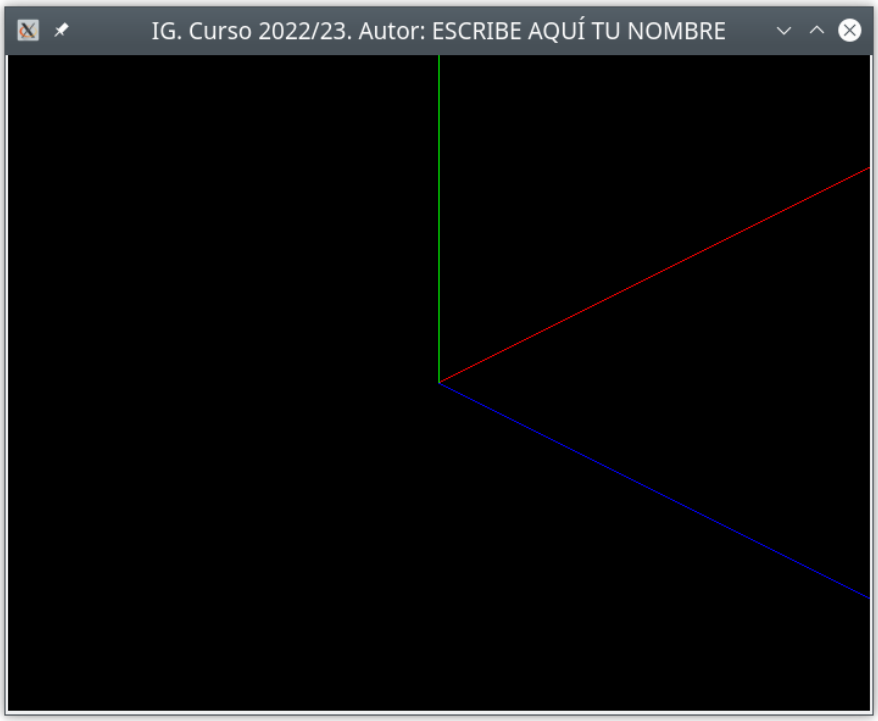
Práctica 1: Programación con una biblioteca de programación gráfica

Objetivos

- Saber crear programas que dibujen geometrías simples con OpenGL.
- Entender la estructura de programas sencillos usando OpenGL y glut.
- Aprender a utilizar las primitivas de dibujo de OpenGL.
- Distinguir entre la creación del modelo geométrico y su visualización.

Código inicial

Para el desarrollo de esta práctica se entrega el esqueleto de una aplicación gráfica programada en C++ sobre OpenGL y glut, formado por los siguientes módulos:



[r0.50]

- `practicasIG.c`: Programa principal. Inicializa OpenGL y glut, crea la ventana de dibujo, y activa los manejadores de eventos (Debes editarlo para escribir tu nombre en el título de la ventana X).
- `entradaTeclado.c`: Contiene las funciones que respoden a eventos de teclado (Todas las prácticas).
- `modelo.c`: Contiene la función que dibuja la escena, y las funciones de creación de objetos (Todas las prácticas).
- `mouse.c`: Contiene las funciones que responden a eventos de ratón (Práctica 5).
- `visual.c`: Contiene las funciones de proyección transformación de visualización, y el *callback* de cambio de tamaño de ventana (Práctica 4 y 5)
- `file_ply_stl.cc`: Funciones de lectura de archivos ply (práctica 2).
- `practicasIG.h`: Incluye los archivos de cabecera de todos los módulos.

Abre los diferentes archivos y mira su contenido, pero no te preocupes por las cosas que en este momento no entiendas.

En las primeras prácticas trabajaremos solamente con los archivo `modelo.c` y `entradaTeclado.c`.

El código base se puede compilar usando el fichero `Makefile` incluido. Si lo ejecutas debes ver los ejes del sistema de coordenadas, dibujados con una cámara

orbital que mira al origen de coordenadas. Puedes girar la cámara alrededor del origen (usando las teclas **X** e **Y** o bien las teclas de movimiento del cursor: ←, ↑, →, ↓) y alejarla o acercarla (usando **d** y **D** o las teclas de avance y retroceso de página: **Re Pág** y **Av Pág**).

Funcionalidad a desarrollar

Se deberá crear y visualizar una pirámide de base cuadrada y un cubo (ver Fig. 1.1), que se visualizarán con los siguientes modos:

- Puntos
- Alambre
- Sólido sin iluminación
- Sólido con iluminación

El cambio del modo de visualización se hace usando el teclado.

Se debe utilizar un color diferente para cada modelo, y los dos deben dibujarse en el escenario sin solaparse cerca del origen de coordenadas.

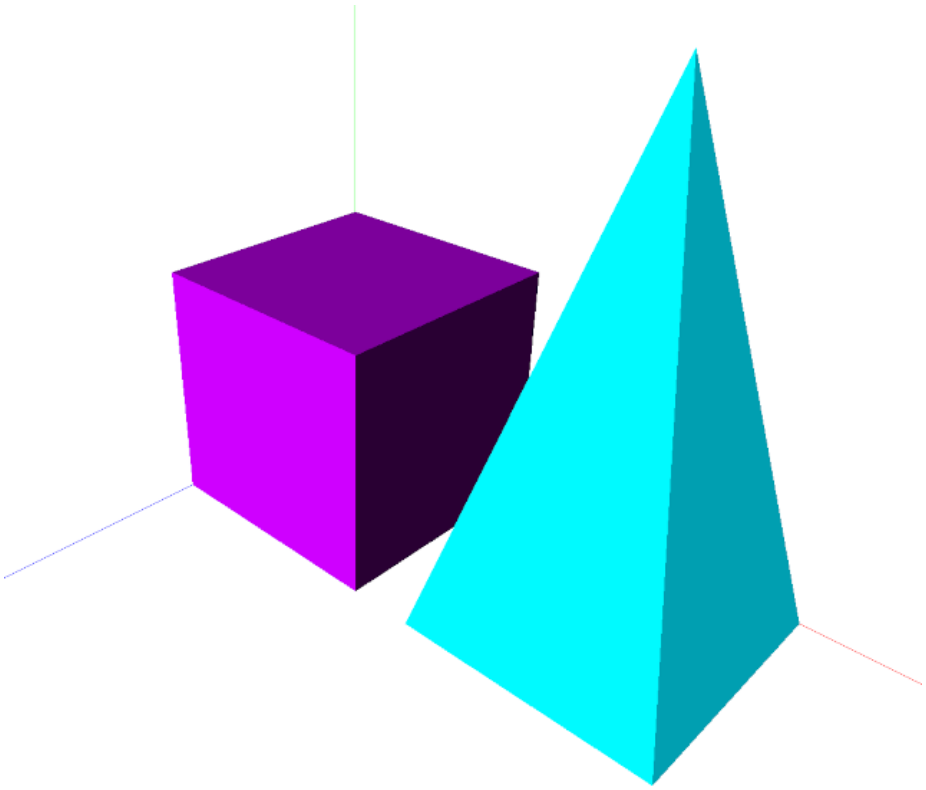


Figura 1: Escena creada en la práctica 1

Procedimiento

Antes de nada abre los archivos `practicasIG.c`, `entradaTeclado.c` y `modelo.c`, familiarízate con el código.

Observa la clase objeto3D (definida en `modelo.h`):

```
class Objeto3D
{
  public:

  virtual void draw( ) = 0; // Dibuja el objeto
} ;
```

y la definición de `Ejes` como una clase derivada de `Objeto3D`. Los objetos que crees en la práctica los debes crear igualmente como clases derivadas de `Objeto3D`.

A continuación añade tu nombre a la ventana X. Para ello edita la llamada a `glutCreateWindow` en `practicasIG.c`.

Comienza dibujando el cubo. Para ello crea una clase `Cubo`, con un constructor al que le puedas pasar el tamaño `Cubo(float lado)` e implementa su método `draw`. Puedes utilizar triángulos o cuadriláteros para dibujarlo. Crea un cubo y llama a su método `draw` desde la función `Dibuja` para que se dibuje en cada fotograma (*frame*). Coloca la llamada al final del dibujo (antes del `glPopMatrix`).

Para que se calcule la iluminación debes asignar a cada cara su normal (puedes hacer un dibujo para ver que normal tiene cada cara), y ten en cuenta que los vértices se deben dar en sentido antihorario mirando el modelo desde fuera:

```
glBegin( GL_QUADS );
    glNormal3f( -1.0, 0.0, 0.0 );
    glVertex3f( x, 0, 0 );
    glVertex3f( x, y, 0 );
    glVertex3f( x, y, z );
    glVertex3f( x, 0, z );
    ...
```

Ahora crea la pirámide siguiendo el mismo procedimiento, utilizando dos parámetros en el constructor (`lado` y `alto`), y haz que se dibuje junto al cubo, para ello puedes aplicarle una traslación. No es difícil calcular geométricamente las normales de las caras, pero no hace falta que te preocupes de esto ahora, puedes asignarlas de forma aproximada (veremos como calcularlas mas adelante). Para que se dibuje de otro color crea otra variable para representar el nuevo color y aplícalo antes de dibujar la pirámide:

```
Cubo.draw();
glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, color2 );
glTranslatef(5,0,0);
Piramide.draw();
```

Para cambiar el modo de visualización puedes usar la función glPolygonMode para indicarle a OpenGL que debe dibujar de cada primitiva (`GL_POINT`, `GL_LINE` o `GL_FILL`):

```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
```

Para modificar interactivamente el modo de dibujo crea una variable en `modelo.c` para almacenar el modo actual, e inicialízala a `GL_FILL`:

```
int modo = GL_FILL;
```

Crea una función para cambiar el modo (p.e. `void setModo(int M)`) que asigne el valor de `M` al modo. Tendrás que añadir su cabecera a `modelo.h`.

Añade casos en el `switch` de la función `letra` en `entradaTeclado` para responder a las pulsaciones de las letras **P**, **L**, **F**. Haz que en cada una se llame a `setModo` con el modo que corresponda.

Para activar y desactivar el cálculo de iluminación se puede seguir un procedimiento parecido, usando la tecla **I**: Crea una variable para indicar si se activa la iluminación y una función para modificar su valor. Haz que se cambie su estado pulsando la letra **I**. Por último, haz que se active o desactive el cálculo de iluminación en la función `Dibuja` llamando a

```
glDisable (GL_LIGHTING);
```

o

```
glEnable (GL_LIGHTING);
```

Ten en cuenta que la función que dibuja los ejes activa el modelo de iluminación en el código de partida. Tendrá que hacerlo solamente cuando se esté en el modo *Sólido con iluminación*.

Temporización

Se recomienda realizar esta práctica en una única sesión. Para ello, debe trabajarse previamente en casa.

Prácticas de Informática Gráfica

Grupos C y D (Curso 2024/2025)

Germán Arroyo y Juan Carlos Torres

2024-06-06

Práctica 2: Modelos poligonales

Objetivos

- Entender la representación de mallas de triángulos
- Saber calcular las normales de los triángulos y los vértices
- Saber leer, representar y visualizar una malla de triángulos representada en un archivo PLY

Código inicial

Partiremos del código creado en la práctica 1.

Funcionalidad a desarrollar

- **Representación de mallas de triángulos**. Se creará una clase para representar mallas de triángulos, con posibilidad de almacenar normales por vértice y por triángulo.
- **Objetos ply**. Se creará un constructor de mallas de triángulos que lea la información de la malla de un archivo ply.
- **Cálculo de normales**. Para ambos tipos de objetos se deberán calcular las normales de cara y de vértice.
- **Dibujo en modos FLAT y SMOOTH** El método de dibujo debe permitir visualizar las mallas tanto en modo FLAT como en modo SMOOTH.
- **Creación de escena**. Se incluirá en la escena al menos dos ply uno dibujado en modo FLAT y el otro en modo SMOOTH.

La Fig. 1.1 muestra un posible resultado de la práctica. La escena creada debe contener dos modelos PLY (cada uno dibujado con un modo de sombreado diferente) y una superficie de revolución.

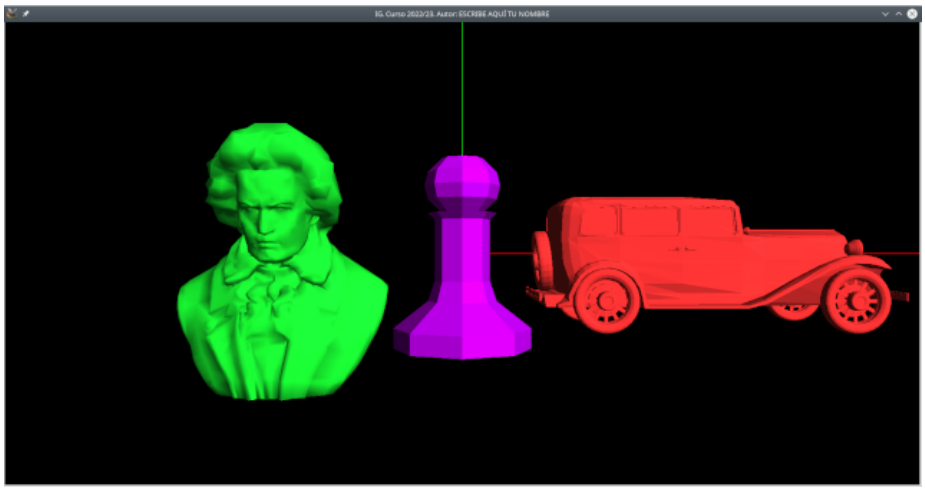


Figura 1: Escena creada en la práctica 2

Desarrollo

Representación de la malla de triángulos

Crea estructuras de datos para representar mallas de triángulos que te permitan identificar la malla con una variable (para facilitar el pasarla como argumento a la función de dibujo de mallas que crearás mas adelante en esta práctica). La malla debe contener al menos los vértices, los triángulos y las normales de vértice y de triángulo.

Si quieres hacer el código orientado a objetos puedes crear una clase malla virtual heredando de Objeto3D.

Lectura de ply

Vamos a incluir funcionalidad para que se puedan leer modelos de un archivo en nuestras mallas. Si has creado una clase para representar mallas puedes crear una subclase y programar un constructor que cree la malla a partir del contenido de un archivo que le pasas como argumento.

Las mallas las leeremos de archivos PLY. PLY es un formato para el almacenamiento de modelos poligonales en fichero. Las siglas proceden de *Polygon File Format*, ha sido diseñado por la Universidad de Stanford.

Un fichero ply puede contener información en formato ASCII o binario. En cualquier caso tendrá una cabecera en ASCII que indica el tipo de datos que contiene. La cabecera determina además como se estructura la información, la geometría que contiene, tipos de datos etc.

El formato permite guardar vértices, polígonos y atributos de vértices (normales,

coordenadas de textura,..).

Es posible descargar modelos 3D en formato PLY de diferentes web (p.e. en 3dvia o en Robin).

Hay muchas funciones publicadas para leer archivos PLY. En el código de prácticas tienes incluido el lector de PLYs de Carlos Ureña. Consta de dos archivos:

- file_ply_stl.h : declaración de las funciones ply::read y ply::read_vertices.
- file_ply_stl.cc : implementación.

Para leer un archivo PLY basta con llamar a la función `ply::read`, tal como se muestra en el siguiente ejemplo:

```
#include <vector>
#include "file_ply_stl.h"

...

// coordenadas de vértices
std::vector<float> vertices_ply ;

// índices de vertices de triángulos
std::vector<int> caras_ply ;

...

ply::read( "nombre-archivo", vertices_ply, caras_ply );
...
```

Tras la llamada se obtienen en `vertices_ply` las coordenadas de los vértices ($3n$ flotantes en total, si hay n vértices en el archivo) y `en caras_ply` los índices de vértices de los triángulos ($3m$ enteros, si hay m triángulos en el archivo).

Cálculo de normales

Para visualizar las mallas con iluminación necesitamos calcularle las normales. Añadiremos una función para calcular las normales de cara y de vértice.

Esta funcionalidad puede ser un método que puedes llamar después desde el constructor después de cargar el modelo PLY.

Para calcular la normal de una cara triangular constituida por vértices P_0, P_1 y P_2 ordenados en sentido antihorario, calculamos el producto vectorial de los vectores $\overrightarrow{(P_0, P_1)}$ y $\overrightarrow{(P_0, P_2)}$ (ecuación [eqN0]) y dividimos el resultado por su módulo para obtener un vector perpendicular y de módulo unidad (ecuación [eqN]) .

$$\vec{N_0} = (\overrightarrow{(P_0, P_1)} \times \overrightarrow{(P_0, P_2)})$$

$$\vec{N} = \frac{\vec{N_0}}{modulo(\vec{N_0})}$$

Las normales de los vértices se pueden calcular sumando las normales de las caras que comparten el vértice, y normalizando el vector resultante (observa que el resultado **NO es la media de los vectores normales**, ya que la media no será un vector normalizado).

Dado que en nuestra estructura de datos tenemos enlaces `Cara` → `Vértice`, debemos hacer la suma de las normales de los vértices iterando en la lista de caras:

1. Inicializar normales de todos los vértices a $(0, 0, 0)$
2. Para cada cara:
   - Sumar su normal a sus tres vértices
3. Para cada vértice:
   - Normalizar su normal

Tanto al calcular las normales de la cara como las de los vértices se debe comprobar que el módulo del vector es mayor que cero antes de hacer la división por el módulo.

Dibujo con sombreado plano y suave

Para dibujar el modelo con sombreado plano (cálculo de iluminación por caras) usamos:

```
glShadeModel(GL_FLAT);
```

antes de comenzar a dibujar el modelo. La iluminación de la cara se calcula con la normal que se haya pasado a OpenGL antes del último vértice de la cara. Por tanto, tenemos que dar la normal de cada cara (usando `glNormal3f(nv,ny,nz)`) antes de que se haya enviado el último vértice de la cara.

Para dibujar con sombreado suave (cálculo de iluminación por vértice) usamos:

```
glShadeModel(GL_SMOOTH);
```

y damos la normal de cada vértice justo antes de enviar el vértice.

Para simplificar el código puedes hacer dos métodos de dibujo de las mallas, una para cada modo de sombreado.

Creación de la escena

El programa final debe crear una escena mostrando los elementos creados(ver Fig. 1.1).

Temporización

Se recomienda realizar esta práctica en dos sesiones.

Prácticas de Informática Gráfica

Grupos C y D (Curso 2024/2025)

Germán Arroyo y Juan Carlos Torres

2024-06-06

Práctica 3: Modelos jerárquicos

Objetivos

- Aprender a diseñar e implemetar modelos jerárquicos de objetos articulados.
- Aprender a crear el grafo de escena.
- Aprender el fucionamiento de la pila de transformaciones.
- Aprender a modificar interactivamente parámetros del modelo.
- Aprender a implementar animaciones sencillas.

Funcionalidad a desarrollar

- Diseñar el grafo de un modelo articulado con al menos tres grados de libertad.
- Crear las estructuras de datos para representarlo.
- Crear el código de visualización.
- Añadir código para modificar los parámetros de las articulaciones con teclado.
- Añadir código para generar una animación del modelo.

Desarrollo

En esta práctica se debe diseñar un modelo jerárquico con al menos tres articulaciones (que se deben controlar con giros y desplazamientos). Se puede tomar como ejemplo el diseño de una grúa semejante a la de la Fig. 1.1, que tiene tres grados de libertad: ángulo de giro del brazo, desplazamiento del gancho en el brazo y altura del gancho.

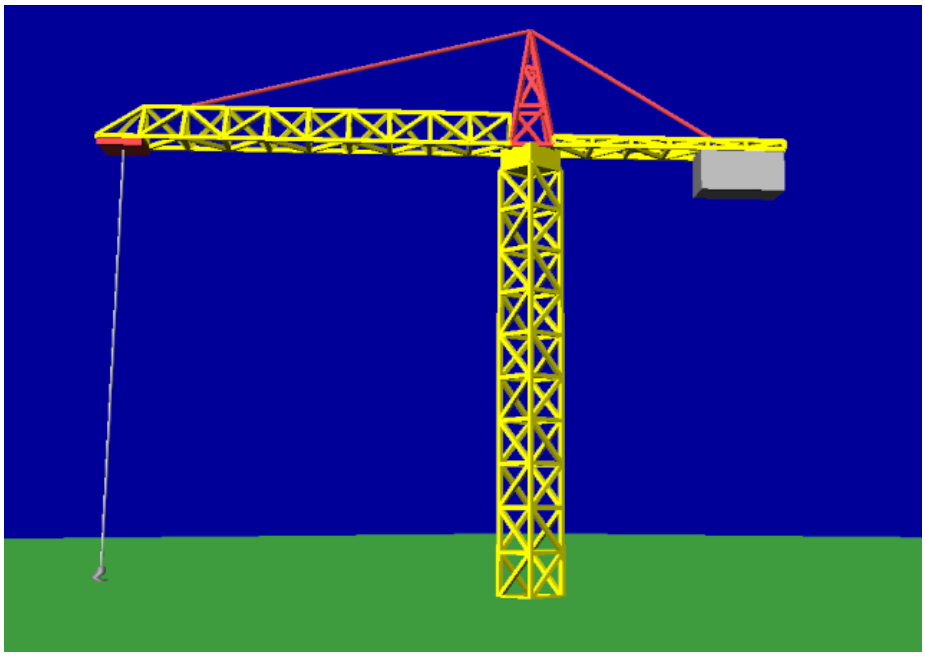


Figura 1: Ejemplo de modelo jerárquico

Nodos terminales

El modelo puede incluir como nodos terminales:

- Objetos predefinidos en glut y glu.
- Objetos creados en la práctica 1.
- Mallas indexadas leídas con el código de la práctica 2 (Puedes descargar modelos de internet, en este caso deberás incluirlos en tu entrega).
- Objetos de revolución creados con el código de la práctica 2, y otros modelos procedurales que construyeras en la práctica.

Diseño del grafo de escena

El diseño del modelo se debe materializar en un grafo de escena que se entregará en un archivo PDF.

El grafo debe incluir todas las transformaciones, indicaciones sobre los parámetros o grados de libertad, y referencias a los objetos usados como nodos terminales.

Crear las estructuras de datos para representar el modelo

Se deben definir las estructuras de datos necesarias para almacenar el modelo (puede ser una subclase de `Objeto3D`). El modelo debe incluir los parámetros asociados a la construcción del modelo (medidas de elementos, posicionamiento, etc.) y los parámetros que se vayan a modificar en tiempo de ejecución, así como métodos para modificar estos últimos.

Crear el código de visualización

Implementar el método `draw` del objeto de forma que se haga un recorrido un preorden del grafo de escena, haciendo uso de la pila de transformaciones de OpenGL.

Para poder verificar el código de forma incremental se recomienda implementar el grafo de forma ascendente comprobando después de la creación de cada nodo que el resultado es el previsto en el diseño del grafo.

Modificar los parámetros de las articulaciones con el teclado

Edita el fichero `entradaTeclado.c` para añadir ordenes que modifiquen los parámetros del modelo. Para cada uno de los grados de libertad incluye dos opciones, una para aumentarlo y otra para decrementarlo. Utiliza las teclas **C**, **V**, **B**, **M** y **N**, mayusculas para aumentar y minusculas para disminuir. Puedes tener presente los límites de cada movimiento.

Ejemplo de codificación:

```
case 'B':
  grua.angY += 1;
  if( grua.angY>360 ) grua.angY -= 360;
  break;
case 'b':
  grua.angY -= 1;
  if( grua.angY<0 ) grua.angY += 360;
  break;
```

Generar una animación del modelo

Para animar el modelo utiliza la función de fondo definida en la plantilla. Esta función se ejecuta periódicamente (cada 30 ms). En ella puedes realizar la actualización de cada parámetro en cada iteración.

Puedes incluir un modo adicional en el programa en el que todos los parámetros se animen (se modifique su valor en cada iteración). Haz que se entre y se salga de este modo pulsando la tecla **A**.

Alternativamente al uso del modo animación puedes programar una velocidad de cambio de cada parámetro que programarás incrementando en cada una

de ellas al parámetro correspondiente en cada fotograma. Si optas por esta opción deberás incluir la opción de aumentar y disminuir las velocidades de los parámetros desde teclado (por ejemplo pulsando las teclas **F**, **G**, **H**, **K**, **L**).

Temporización

Se recomienda realizar esta práctica en dos sesiones.

Prácticas de Informática Gráfica

Grupos C y D (Curso 2024/2025)

Germán Arroyo y Juan Carlos Torres

2024-06-06

Práctica 4: Materiales, fuentes de luz y texturas

Objetivos

- Saber definir fuentes de luz y materiales en OpenGL
- Saber generar y representar coordenadas de textura en mallas de triángulos
- Saber visualizar modelos con textura
- Entender el funcionamiento de modelo de iluminación local

Funcionalidad a desarrollar

- Modificar la representación de las mallas para añadir el material.
- Modificar la representación del cubo para incorporar texturas, dibujando un dado.
- Modificar la representación de las mallas para incorporar coordenadas de textura y texturas.
- Crear una escena con un dado y tres copias del mismo objeto con diferentes materiales (al menos uno con textura) y dos fuentes de luz.

Desarrollo

Añadir materiales y textura a las mallas

Añadiremos variables a la clase malla para almacenar las propiedades de material incluyendo al menos: reflectividad difusa y especular. Crearemos métodos para asignarlos y modificaremos el método de dibujo para usar el material asignado. Si no se incluye reflectividad ambiente se usará para ella el valor de la reflectividad difusa.

Comprueba que funciona correctamente modificando las reflectividades de los objetos de la escena.

Asignar una textura leída de un archivo a un objeto. Necesitamos añadir código para leer las imágenes que se usarán de textura. Para ello usaremos una función del lectura de imágenes JPEG.

Descarga de Prado el fichero ZIP con archivos adicionales para esta práctica y descomprimelo. Verás los archivos `lector-jpg.cpp` y `lector-jpg.h`. Añádelos a tu programa.

Esta función utiliza la librería `libjpeg-dev`, debes instalarla en tu ordenador y añadir `-l jpeg` en tu fichero `Makefile`.

Para leer imágenes utiliza

```
unsigned char* LeerArchivoJPEG(const char *nombre_arch,
                               unsigned &ancho, unsigned &alto)
```

que recibe el nombre del archivo de la imagen a leer, y devuelve la imagen como puntero a un array de `unsigned char` (y las dimensiones de la imagen).

Añade a tu clase `Objeto3D` un método para asignar la textura al objeto leyéndola de un archivo. Añade también una variable para almacenar el identificador de la textura:

```
GLuint texId;
```

Esta operación se debe realizar cuando el contexto del OpenGL ya se haya iniciado (no se puede hacer en el constructor).

Textura del dado

Crea un objeto dado. Asígnale como textura la imagen `dado.jpg` que encontrarás en el zip que has descargado. Haz la lectura de la imagen y la asignación de la textura. Observa la imagen y determina las coordenadas de textura que debes usar para cada vértice.

Para que se visualice el cubo con su textura aplicada, primero debes cargar la textura en OpenGL (esto transfiere la textura de memoria RAM a memoria VRAM de la tarjeta gráfica):

```
glGenTextures(1, texId);
glBindTexture(GL_TEXTURE_2D, texId);

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);
```

donde `image` es el array que contiene la imagen leída. En este punto, ya se puede liberar la memoria RAM usada para ese array, ya que el contenido se encuentra en la VRAM de la tarjeta.

Este proceso se debe realizar **una sola vez** (al asignarle la textura) para evitar consumir la VRAM de la tarjeta haciendo múltiples copias de nuestra textura (lo que haría que cada vez vaya más lenta nuestra aplicación, hasta incluso poder llegar a colgar el sistema) . El identificador de textura debe ser una variable del objeto.

Para dibujar el objeto será necesario cargar su textura y activar el procesamiento de las texturas:

```
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, texId);
```

Después de dibujar el objeto deberemos desactivar el procesamiento de las texturas.

```
glDisable(GL_TEXTURE_2D);
```

El resultado de visualizar el dado debe ser parecido a la Fig. 1.1.

Añadir coordenadas de textura a las mallas. Se añadirá a las mallas un vector de coordenadas de textura, que contendrá n_v pares de valores coordenadas de textura (dos valores de tipo `float`), siendo n_v el número de vértices. Este vector se usará para asociar coordenadas de textura a cada vértice. En las mallas que no tengan o no necesiten coordenadas de textura, dicho vector estará vacío (0 elementos) o será un puntero nulo y el identificador de textura, heredado de `Objeto3D`, será cero.

Calculo de coordenadas de textura de las mallas. Añade un método a las mallas para calcular las coordenadas de textura proceduralmente, creando un método que las calcule a partir de las coordenadas de los vértices. Puedes utilizar proyecciones paralelas a un plano, coordenadas esféricas o cilíndricas. Para ello tendrás que calcular el centro geométrico de la malla y su caja envolvente.

Dibujo con textura. Modifica el método de dibujo de las mallas para que se utilice la textura cuando tengan textura asignada.

Creación de la escena

Crea una escena con al menos una malla con textura, el dado y tres copias del mismo objeto con diferentes materiales.

Temporización

Se recomienda realizar esta práctica en dos sesiones.

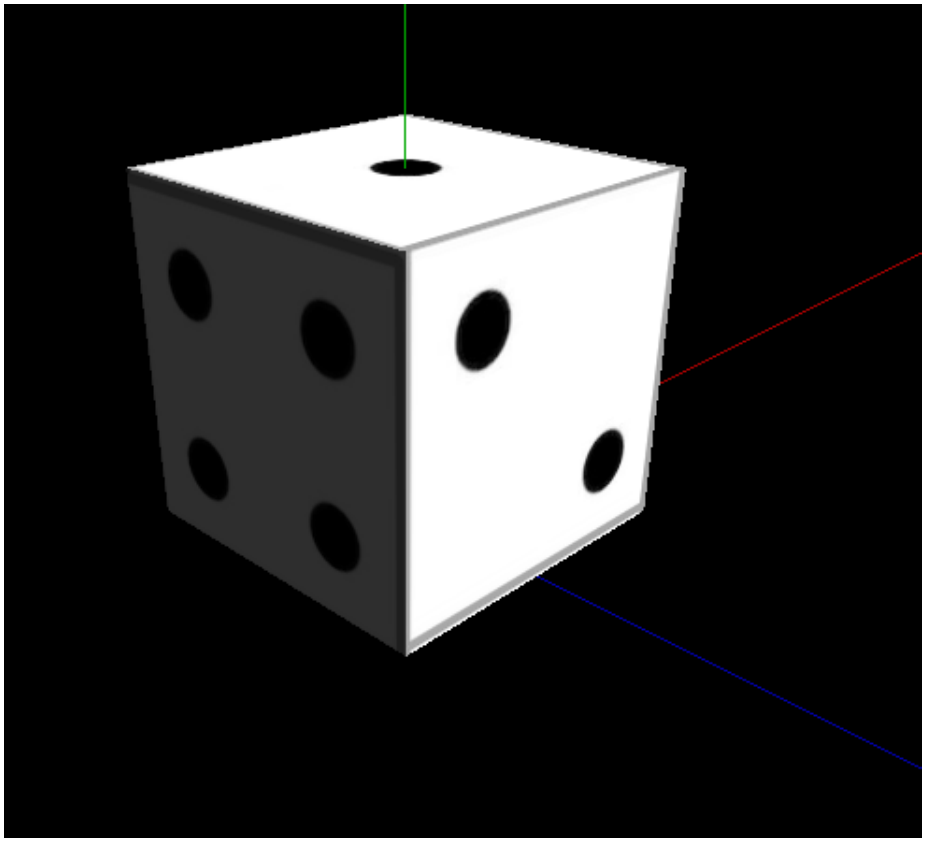


Figura 1: [fig:dado]Imagen del dado

Prácticas de Informática Gráfica

Grupos C y D (Curso 2024/2025)

Germán Arroyo y Juan Carlos Torres

2024-06-06

Práctica 5: Interacción

Objetivos

- Saber desarrollar aplicaciones gráficas interactivas sencillas, gestionando los eventos de entrada de teclado y ratón usando glut.
- Saber implementar operaciones de selección de objetos de la escena.
- Saber controlar interactivamente la cámara usando el ratón.

Funcionalidad a desarrollar

1. Seleccionar objetos de la escena con el ratón.

Desarrollo

Seleccionar objetos de la escena usando el ratón

Se incluirán en la escena los objetos generados en las prácticas 2, 3 y 4, y cualquier otro objeto que se desee. Para realizar la selección usaremos un código de color para cada objeto seleccionable. Se trata de crear una función de dibujado distinta para cuando queremos seleccionar. Está función, que será llamada cuando el usuario hace una operación de selección (por ejemplo click izquierdo), para la escena "para seleccionar" en el buffer trasero utilizando identificadores como colores. Para terminar la selección se leerá el color del píxel donde el usuario ha hecho click. Si no se hace un intercambio de buffers, el usuario no verá esa imagen con falso color.

Para no duplicar el método de dibujo, y evitar inconsistencia a la hora de seleccionar, crearemos una función de dibujo de la escena, que contendrá todo el código del *callback* `Dibuja`, salvo la llamada a `glutPostRedisplay`. Cambiaremos el código de `Dibuja` para llamar a esta función

```
void Dibuja()
{
  dibujoEscena();
  glutSwapBuffers();
}
```

Crearemos una función para codificar los identificadores de los objetos como colores (ten en cuenta que tendrás que almacenar los identificadores de los objetos). Por ejemplo, si tienes objetos articulados (no mas de 255) y quieres seleccionar sus componentes:

```
void ColorSeleccion( int i, int componente)
{
  unsigned char r = (i & 0xFF);
  unsigned char g = (componente & 0xFF);
  glColor3ub(r,g,0);
}
```

Es necesario que no se modifique el color asignado al pixel, por eso damos los colores como `unsigned byte` con `glColor3ub`, es decir, como enteros de 0 a 255 (256 valores). Plantéate si sabrías cambiar el código si tuvieras más de 255 modelos distintos, aunque no es necesario que lo realices para esta práctica. ¿Sabrías explicar por qué no puedes tener 256 modelos?

Además tendremos que desactivar el `GL_DITHER`, `GL_LIGHTING` y `GL_TEXTURE`.

En `dibujoEscena` asigna a cada objeto su material y su color de selección. Puedes controlar si se usa uno u otro para dibujar activando o desactivando el cálculo de iluminación (asumiendo que no lo cambias durante el dibujo). Otra opción es usar un parámetro en `dibujaEscena` para controlar el uso de uno u otro.

El proceso de selección se inicia cuando se pulsa el botón izquierdo del ratón, ejecutará las siguientes acciones:

- Desctivar el cálulo de iluminación.
- Llamar a la función `dibujoEscena`.
- Leer el color del pixel (x,y).
- Decodificar el identificador
- Lanzar un evento de redibujado de la escena.

Para leer el color del pixel podemos usar la función `glReadPixels`:

```
void glReadPixels(GLint x, GLint y, GLsizei width, GLsizei height,
                  GLenum format, GLenum type, GLvoid *pixels);
```

donde

`x` e `y` representa la esquina inferior izquierda del cuadrado a leer (en nuestro caso las coordenadas (x, y) del *pick*).

`width` y `height` son el ancho y alto del área a leer ((1., 1.) en nuestro caso).

`format` es el tipo de dato a leer (coincide con el format del buffer, `GL_RGB` o `GL_RGBA`).

`type` es el tipo de dato almacenado en cada pixel, según hayamos definido el `glColor` (p.ej. `GL_UNSIGNED_BYTE` de 0 a 255, o `GL_FLOAT` de 0.0 a 1.0).

`pixels` es el array donde guardaremos los pixels que leamos. Es el resultado de la función.

Por ejemplo:

```
unsigned char data[4];
...
glReadPixels(x,viewport[3]-y,1,1,GL_RGBA,GL_UNSIGNED_BYTE,data);
```

El proceso de decodificación dependerá de como se hayan codificado los identificadores, teniendo en cuenta que el último parámetro recibe el color del píxel. Por ejemplo, en el caso descrito anteriormente:

```
*i=data[0];
*componente=data[1];
```

En resumen, el proceso de selección puede realizarse en un función del tipo:

```
int pick(int x, int y,int * i)
{

   GLint viewport[4];
   unsigned char data[4];

   glGetIntegerv (GL_VIEWPORT, viewport);
   glDisable(GL_DITHER);
   glDisable(GL_LIGHTING);
   dibujoEscena();
   glEnable(GL_LIGHTING);
   glEnable(GL_DITHER);
   glFlush();
   glFinish();

   glReadPixels(x,viewport[3]-y,1,1,GL_RGBA,GL_UNSIGNED_BYTE,data);

   *i=data[0];

   glutPostRedisplay();
   return *i;
}
```

Temporización

Se recomienda realizar esta práctica en dos sesiones.

Prácticas de Informática Gráfica

Grupos C y D (Curso 2024/2025)

Germán Arroyo y Juan Carlos Torres

2024-06-06

Práctica 6: Proyecto

Objetivos

El objetivo de esta práctica es diseñar y programar una aplicación gráfica interactiva, que incluya la mayor parte de los conceptos vistos en la asignatura.

Desarrollo

Se deberá realizar un documento de diseño de la aplicación su objetivo, funcionalidad y los aspectos mas relevantes del diseño. Este documento se entregará junto con el código desarrollado.

Se podrá integrar el código realizado en las prácticas previas.

Temporización

Esta práctica requiere de varias sesiones para terminarla, se recomiendan unas 3 sesiones mínimo.