Prácticas de Informática Gráfica

Grado en Informática y Matemáticas. Curso 2017-18.

UNIVERSIDAD
DE GRANADA

ETSI Informática y de Telecomunicación.
Departamento de Lenguajes y Sistemas Informáticos.

Índice general

Práctica 1

Visualización de modelos simples

1.1. Objetivos

Con esta práctica se quiere que el alumno aprenda:

- A crear estructuras de datos que permitan representar objetos 3D sencillos
- A utilizar las primitivas de dibujo de OpenGL para dibujar los objetos

1.2. Desarrollo

Para el desarrollo de esta práctica se entrega el esqueleto de una aplicación gráfica basada en eventos, mediante GLUT, y con la parte gráfica realizada por OpenGL. Para facilitar su uso, la aplicación permite abrir una ventana, mostrar unos ejes y mover una cámara básica.

El alumno deberá crear y visualizar un tetraedro y un cubo. Para ello, creará las estructuras de datos que permitan representarlos mediante sus vértices y caras. Usando dicha información y las primitivas de dibujo de OpenGL los visualizará con los siguientes modos:

- Puntos: se visualiza un punto en la posición de cada vértice del modelo.
- Alambre: se viualiza como un segmento cada arista del modelo.
- Sólido: se visualizan los triángulos rellenos todos de un mismo color (plano).
- Ajedrez: se visualizan los triángulos de dos colores alternos: los pares de un color y los impares de otro, donde la paridad está determinada por la posición del triángulo en la tabla de triángulos.

Las primitivas se enviarán a OpenGL usando la función `glDrawElements` en todos los modos de dibujo. Excepto en el modo ajedrez, se puede hacer una única llamada a esta función. Para el modo ajedrez, se pueden usar dos llamadas.

1.3. Evaluación

La evaluación de la práctica se hará mediante la entrega de las prácticas (via la plataforma PRADO), seguida de un sesión de evaluación en el laboratorio, en la cual se harán modificaciones sobre el código del alumno y el nuevo código se subirá a la plataforma PRADO.

- La nota máxima será de 10 puntos, si el alumno implementa todos los requerimientos descritos en este guión, y además hace visualización usando el cauce gráfico programable.
- Si el alumno implementa la visualización usando exclusivamente el cauce de la funcionalidad fija, la nota máxima será de 7 puntos.

Las modificaciones que se pidan durante la sesión de evaluación serán evaluadas entre 0 y la nota máxima descrita aquí arriba.

1.4. Teclas a usar

El programa permitirá pulsar las siguientes teclas:

- **tecla m/M**: cambia el modo de visualización activo (pasa al siguiente, o del último al primero)
- **tecla p/P**: cambia la práctica activa (pasa a la siguiente, o de la última a la primera).
- **tecla o/O**: cambia el objeto activo dentro de la práctica (pasa al siguiente, o del último al primero)

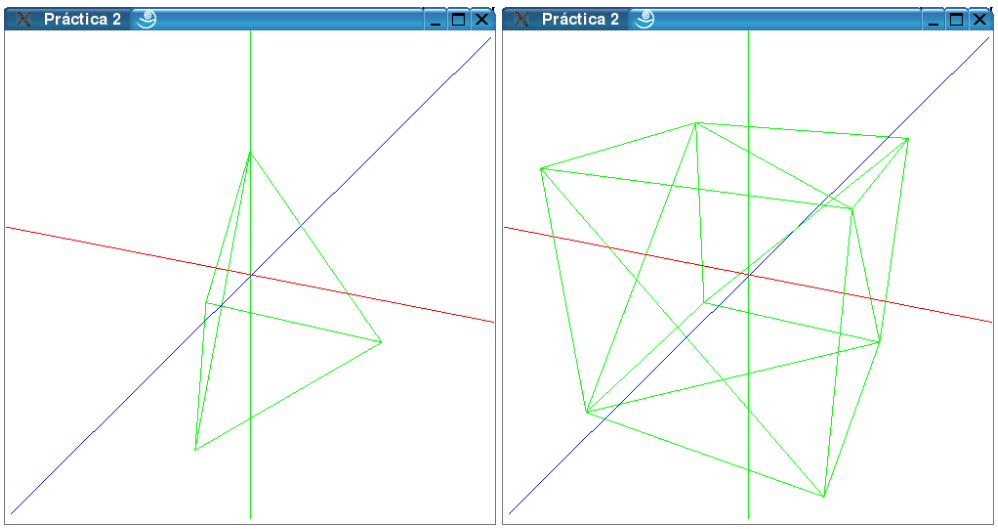
Además de estas teclas (que se deben de implementar), la plantilla que se proporciona incopora otras teclas, válidas para todas las prácticas. En concreto, son las siguientes:

- **tecla q/Q o ESC**: terminar el programa
- **teclas de cursos**: rotaciones del objeto entorno al origen.
- **teclas +/-, av.pág/re.pág.**: aumentar/disminuir la distancia de la camara al origen (zoom).

1.5. Implementación

Una vez descomprimido el archivo `tgz` en una carpeta vacía, se crearán estas subcarpetas:

- `objs`: carpeta vacía donde se crearán los archivos `.o` al compilar.
- `plys`: archivos ply de ejemplo para la práctica 2, proporcionados por el profesor.



(a) Tetraedro (b) Cubo

Figura 1.1: Tetraedro y cubo visualizados en modo alambre.

- `imgs`: imágenes de textura para la práctica 4, proporcionadas por el profesor.
- `include`: archivos de cabecera de los módulos auxiliares (p.ej.: manejo de tuplas de valores reales para coordenadas y colores).
- `srcs`: archivos fuente C/C++ de los módulos auxiliares (p.e.: lectura de plys, lectura de jpgs, shaders, etc...).
- `srcs-alum`: archivos fuente del programa principal, y de cada una de las prácticas (todos ellos son a completar o extender por el alumno). Asimismo, aqui se incluirán archivos `.ply`, imágenes (`.jpg`), o de otros tipos, distintos de los proporcionados por el profesor, y que el alumno use en sus prácticas.

Para realizar las prácticas es necesario trabajar en la carpeta `srcs-alum`. En esa carpeta se debe completar y extender el código que se proporciona en el archivo `practica1.cpp`

Para compilar el código, basta con teclear `make` (estando en la carpeta `srcs-alum`), esta orden leerá el archivos `makefile` y se encargará de compilar, enlazar y ejecutar el código, incluyendo los módulos auxiliares disponibles en la carpeta `srcs` (cabeceras en `include`). Si no hay errores, se producirá en esa carpeta un ejecutable de nombre `prac`

No se debe de modificar en ningún caso el código de los archivos en las carpetas `srcs` ni `include`. Tampoco se debe añadir ningún archivo en esas carpetas. La revisión de las prácticas para evaluación se hará con el contenido no modifcado de dichas carpetas.

El archivo `makefile` que hay en `srcs-alum` se debe de modificar, pero exclusivamente para añadir nombre de unidades de compilación en la definición de `units_alu`. Se deben añadir los nombres de las unidades (archivos `.cpp`) que estén en `srcs_alum` y que se quieren enlazar para crear el ejecutable.

La implementación requiere completar las siguientes funciones (en `practica1.cpp`):

- Función **`P1_Inicializar`**, para crear las tablas de vértices y caras que se requieren para la práctica. Esta función se invoca desde **`main`**`.cpp` una única vez al inicio del programa, cuando ya se ha creado la ventana he inicializado OpenGL.
- Función **`P1_DibujarObjetos`**, para dibujar las mallas, usando el parámetro `cv`, que contiene la variable que determina el tipo o modo de visualización de primitivas. Esta función se invoca desde **`main`**`.cpp` cada vez que se recibe el evento de redibujado.
- Función **`P1_FGE_PulsarTeclaNormal`**, está función se invoca desde **`main`**`.cpp` cuando se pulsa una tecla normal, la práctica 1 está activa, y la tecla no es procesada en el **`main`**`.cpp`. Sirve para cambiar entre la visualización del tetraedro y el cubo (cambiar el valor de la variable `objeto_activo`) cuando se pulsan alguna tecla. Debe devolver **`true`** para indicar que la tecla pulsada corresponde al cambio de objeto activo, y **`false`** para indicar que la tecla no corresponde a esta práctica.

En el archivo **`main`**`.cpp` es necesario gestionar los eventos de teclado que permiten cambiar el valor de la variable global `modoVis`, que determina el modo de visualización actual.

1.5.1. Contexto y modos de visualización

En el archivo `practicas.hpp` (dentro de `srcs-alum`) se declara la clase **`ContextoVis`**, que contiene, como variables de instancia, distintos parámetros y variables de estado usados durante la visualización de objetos y escenarios en las prácticas. Inicialmente (para esta práctica 1), contiene únicamente el modo de visualización (puntos, alambre, sólido, etc...), en concreto está en la variable de instancia `modoVis`, que es un valor de un tipo enumerado **`ModoVis`**, tipo que también se declara

en ese archivo de cabecera.

Las declaraciones son como se indica aquí:

```
// tipo enumerado para los modos de visualización:
typedef enum
      { modoPuntos, modoAlambre, modoSolido, modoAjedrez } ModoVis ;
// numero de modos distintos
const int numModosVisu = 4 ;
// clase para los distintos parámetros de la visualización
class ContextoVis
{
   public:
      ModoVis modoVis ;  // modo de visualización activo actualmente
} ;
```

Más adelante se definirán nuevos modos de visualización y otros parámetros en la clase **ContextoVis**.

1.5.2. Clase abstracta para objetos gráficos 3d.

La implementación de los diversos tipos de objetos 3D a visualizar en las prácticas se hará mediante la declaración de clases derivadas de una clase base, llamada **Objeto3D**, con un método virtual llamado **visualizarGL**, con una declaración como esta:

```
class Objeto3D
{
   protected:
      std::string nombre_obj ;  // nombre asignado al objeto
   public:
      // visualizar el objeto con OpenGL
      virtual void visualizarGL( ContextoVis & cv ) = 0 ;
      // devuelve el nombre del objeto
      std::string nombre() ;
} ;
```

Cada clase concreta proveerá su propio método **visualizarGL**. Estos métodos tienen siempre un parámetro de tipo **ContextoVis**, que contendrá el modo de visualización que se debe usar (entre otras cosas).

Cualquier tipo de objeto que pueda ser visualizado en pantalla con OpenGL se implementará con una clase derivada de **Objeto3D**, que contendrá una implementación concreta del método virtual **visualizarGL**. El parámetro `modoVis` (dentro de `cv`) servirá para distinguir el modo de visualización que se requiere.

La declaración de las clases **Objeto3D** y **ContextoVis** se hará en un archivo de cabecera llamado **Objeto3D**`.hpp`. La implementación se incluirá en **Objeto3D**`.cpp`, ese archivo incluirá únicamente la implementación del método **nombre** (que simplemente devuelve `nombre_obj`, y se puede usar para depurar). Es necesario añadir al archivo `makefile` el nombre **Objeto3D** (en `units_loc`), para lograr que el nuevo archivo se compile y enlace con el resto.

1.5.3. Clase para mallas indexadas.

Las mallas indexadas son mallas de triángulos modeladas con una tabla de coordenadas de vértices y una tabla de caras (que contiene ternas de valores enteros). Se pueden visualizar con OpenGL en modo inmediato usando la instrucción **`glDrawElements`** o bien **`glBegin`**/**`glEnd`** (mucho menos eficiente). Para implementar este tipo de mallas crearemos una clase (**`MallaInd`**), derivada de **`Objeto3D`** y que contiene:

- Como variables de instancia privadas, la tabla de coordenadas de vértices y la tabla de caras. La primera puede ser un vector stl con entradas de tipo **`Tupla3f`**, y la segunda un vector stl con entradas tipo **`Tupla3i`**.
- Como método público virtual, el método **`visualizarGL`**, que visualiza la malla teniendo en cuenta el parámetro modo, y usando las dos tablas descritas arriba.

El esquema puede ser como sigue:

```
#include "Objeto3D.hpp"

class MallaInd : public Objeto3D
{
   protected:
    // declarar aquí tablas de vértices y caras
    // ....
   public:
      virtual void visualizarGL( ContextoVis & cv ) ;
    // .....
} ;
```

La declaración de esta clase se puede poner en un archivo de nombre **`MallaInd.`**`hpp`, y su implementación en **`MallaInd.`**`cpp`. Es necesario añadir al archivo `makefile` el nombre **`MallaInd`** (en `units_loc`), para lograr que este archivo se compile y enlace con el resto.

Las tablas de vértices y caras se pueden implementar con arrays de C clásicos que contienen flotantes o enteros. No obstante, se recomienda usar vectores STL de tuplas de flotantes o enteros, ya que esto facilitará la manipulación posterior. En este guión, se describen más adelante los tipos que se proporcionan para tuplas de flotantes o enteros (tipos **`Tupla3f`** y **`Tupla3i`**).

1.5.4. Programación del cauce gráfico

Se tiene la opción de usar programación del cauce gráfico para realizar la visualización. Esta programación permitirá visualizar las primitivas de esta primera práctica usando para ello un *shader program* distinto del proporcionado en la funcionalidad fija de OpenGL.

El código fuente de este shader puede coincidir con el fuente sencillo visto en las transparencias de teoría para un *fragment shader* y un *vertex shader* básicos. También se pueden usar las funciones que hemos visto para cargar, compilar y enlazar los programas, que ya están disponibles en la unidad de compilación `shaders` que hay en las carpetas `srcs` (`shaders.cpp`) e `include` (`shaders.hpp`).

La implementación de esta funcionalidad requiere modificar **`main.`**`cpp` para incluir una variable global (de tipo **`GLuint`**) con el identificador del programa. Esta variable se usará para activar dicho programa siempre antes de visualizar.

Los dos archivos `.glsl` requeridos deben de estar en `srcs-alum`, y se deben entregar junto con el resto de fuentes de este directorio.

1.5.5. Clases para los objetos de la práctica 1

Los objetos cubo y tetraedro se implementarán usando dos clases derivadas de **`MallaInd`**, cada una de ellas definirá un nuevo constructor que construirá las dos tablas correspondientes a cada tipo de objeto. Estas clases se pueden declarar e implementar en un par de archivos nuevos, o se puede hacer en `practica1.hpp/.cpp`. En cualquier caso, en el archivo `practica1.cpp` habrá dos variables globales nuevas, una será una instancia del cubo y otra una instancia del tetraedro. El esquema para la clase Cubo (p.ej.) puede ser este:

```
class Cubo : public MallaInd
{
   public:
      Cubo() ;    // crea las tablas del cubo, y le da nombre.
};
class Tetraedro : public MallaInd
{
   public:
      Tetraedro() ;    // crea las tablas del cubo, y le da nombre.
};
```

En la función **`P1_Inicializar`** se crearán las instancias del cubo y el tetraedro. En la función **`P1_DibujarObjetos`** se visualizará el cubo o el tetraedro.

1.5.6. Clases para tuplas de valores enteros o reales con 2,3 o 4 valores

Haciendo *include* de `tuplasg.hpp`, están disponibles estos tipos de datos (clases):

```
// adecuadas para coordenadas de puntos, vectores o normales en 3D
// también para colores (R,G,B)
Tupla3f   t1 ;   // tuplas de tres valores tipo float
Tupla3d   t2 ;   // tuplas de tres valores tipo double

// adecuadas para la tabla de caras en mallas indexadas
Tupla3i   t3 ;   // tuplas de tres valores tipo int
Tupla3u   t4 ;   // tuplas de tres valores tipo unsigned

// adecuadas para puntos o vectores en coordenadas homogéneas
// también para colores (R,G,B,A)
Tupla4f   t5 ;   // tuplas de cuatro valores tipo float
Tupla4d   t6 ;   // tuplas de cuatro valores tipo double

// adecuadas para puntos o vectores en 2D, y coordenadas de textura
Tupla2f   t7 ;   // tuplas de dos valores tipo float
Tupla2d   t8 ;   // tuplas de dos valores tipo double
```

Este trozo código válido ilustra las distintas opciones, para creación, consulta y modificación de tuplas:

```
float       arr3f[3] = { 1.0, 2.0, 3.0 } ;
unsigned    arr3i[3] = { 1, 2, 3 } ;
```

```
// declaraciones e inicializaciones de tuplas
Tupla3f  a( 1.0, 2.0, 3.0 ), b, c(arr3f) ;  // b indeterminado
Tupla3i  d( 1, 2, 3 ), e, f(arr3i) ;        // e indeterminado

// accesos de solo lectura, usando su posición o índice en la tupla (0,1,2,...),
// o bien varias constantes predefinidias para coordenadas (X,Y,Z) o colores (R,G,B):
float x1 = a(0), y1 = a(1), z1 = a(2),    //
      x2 = a(X), y2 = a(Y), z2 = a(Z),    // apropiado para coordenadas
      re = c(R), gr = c(G), bl = c(B) ;   // apropidado para colores

// conversiones a punteros
float *       p1 = a ; // conv. a puntero de lectura/escritura
const float * p2 = b ; // conv. a puntero de solo lectura

// accesos de escritura
a(0) = x1 ;  c(G) = gr ;

// escritura en un 'ostream' (cout) (se escribe como: (1.0,2.0,3.0)
cout << "la tupla 'a' vale: " << a << endl ;
```

En C++ se pueden sobrecargar los operadores binarios y unarios usuales (`+`, `-`, etc...) para operar sobre las tuplas de valores reales:

```
// declaraciones de tuplas y de valores escalares
Tupla3f  a,b,c ;
float    s,l ;

// operadores binarios y unarios de asignación/suma/resta/negación
a = b ;
a = b+c ;
a = b-c ;
a = -b  ;

// multiplicación y división por un escalar
a = 3.0f*b ;     // por la izquierda
a = b*4.56f ;    // por la derecha
a = b/34.1f ;    // mult. por el inverso

// otras operaciones
s = a.dot(b)       ; // producto escalar (usando método dot)
s = a|b            ; // producto escalar (usando operador binario barra )
a = b.cross(c)     ; // producto vectorial (solo para tuplas de 3 valores)
l = a.lengthSq()   ; // calcular módulo o longitud al cuadrado
a = b.normalized(); // hacer a= copia normalizada de b (a=b/modulo de b) (b no cambia)
```

1.6. Instrucciones para subir los archivos

Para entregar la práctica se creará y se subirá un único archivo `.zip`, de nombre igual a `P1.zip` siguiendo estas indicaciones:

- Hacer un zip con todos los fuentes de la carpeta `srcs-alum`, incluyendo **`main`**`.cpp` o cualquier otro, así como los shaders si los hay (archivos `.glsl`). El `zip` debe hacerse directamente en `srcs-alum`, y no puede tener carpetas dentro de él, solo puede contener directamente los archivos indicados.