Dpto. de Lenguajes y Sistemas Informáticos
Escuela Técnica Superior de Ingenierías Informática y Telecomunicación

Prácticas de Informática Gráfica
Grupo C
Curso 2022/23

Germán Arroyo
Juan Carlos Torres

Realizado a partir de material docente creado por:
G. Arroyo, P. Cano, A. López, L. López, D. Martín,
F.J. Melero, C. Romo, J.C. Torres, C. Ureña

Índice general

Introducción

Las prácticas de Informática Gráfica son una parte esencial de la asignatura. Con ellas se pretende aclarar los conceptos que se ven en la asignatura y entender el funcionamiento y los principios de diseño de los sistemas gráficos interactivos.

Objetivos

- Saber diseñar y utilizar las estructuras de datos más adecuadas para representar mallas poligonales, modelos jerárquicos y modelos geométricos en general.
- Saber utilizar y representar transformaciones geométricas.
- Conocer la funcionalidad básica de una biblioteca de programación gráfica.
- Saber diseñar e implementar programas gráficos interactivos, estructurando de forma eficiente la gestión de eventos para garantizar la accesibilidad y la usabilidad.
- Conocer los fundamentos de la visualización, los modelos de iluminación, y entender y poder configurar los parámetros de materiales y luces.
- Conocer los fundamentos de la animación por ordenador.

OpenGL

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

Permite manejar:

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

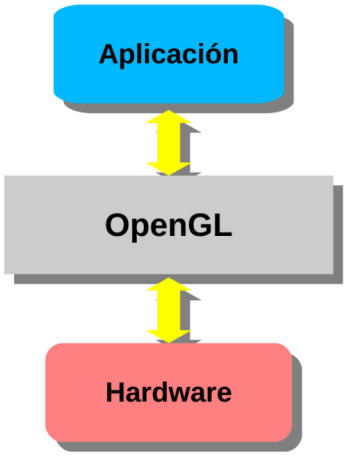

Figura 1: OpenGL es una API para la comunicación con el hardware gráfico

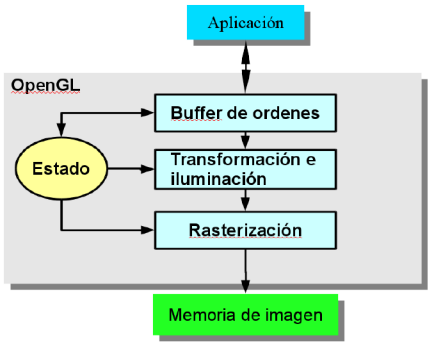

Figura 2: Esquema de funcionamiento de OpenGL

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

A partir de la versión 3.1 hay dos modos de funcionamiento: el modo de compatibilidad y el "core profile". El "core profile" es mas reciente y más potente, pero también mas complejo. En este seminario trabajaremos con la versión clásica de OpenGL, conocida también como "compatibility mode".

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. Estas operaciones no se realizan necesariamente en este orden, con frecuencia el calculo 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

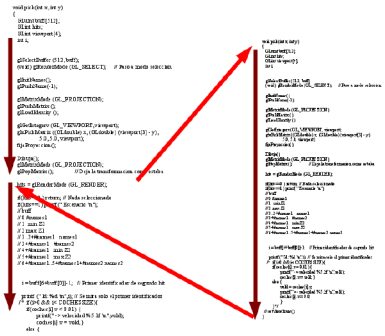

Figura 3: Secuencia de ejecución en un paradigma no orientado a eventos

en GPU acelera enormemente el dibujo ya que las GPU son procesadores paralelos (tipo SIMD). En su estructura mas 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.

OpenGL tiene una librería de utilidades asociada (GLU), que se diseño para implementar funciones que no se esperaba que se pudiesen realizar por hardware como la teselación de polígonos, la creación de texturas o el dibujo de superficies.

Estructura de un programa

Para poder utilizar OpenGL el programa debe inicializarlo y crear al menos una ventana de dibujo. Estas operaciones (que pueden depender bastante del sistema operativo y de gestión de ventanas) no las realiza OpenGL. Para realizarlas se pueden usar la mayor parte de las librerías de gestión de interfaces de usuario (como QT o FLTK).

Una alternativa simple es usar GLUT, que es un toolkit sencillo diseñado específicamente para OpenGL- GLUT incluye funciones para interaccionar con el sistema de gestión de ventanas. Es portable e independiente de la plataforma. Permite crear ventanas gráficas y procesar eventos de entrada. Además incluye funciones para dibujar objetos poliédricos simples.

En cualquier caso un programa en OpenGL suele desarrollarse utilizando el paradigma de programación dirigida por eventos (salvo programas que solo dibujen sin ninguna posibilidad de interacción).

En un paradigma de programación no dirigida por eventos el orden en que se ejecutan las instrucciones lo fija el programador (ver figura 3).

En el paradigma de programación dirigida por evento el orden de ejecución de las operaciones depende de los eventos que se produzcan durante la ejecución del programa. Los

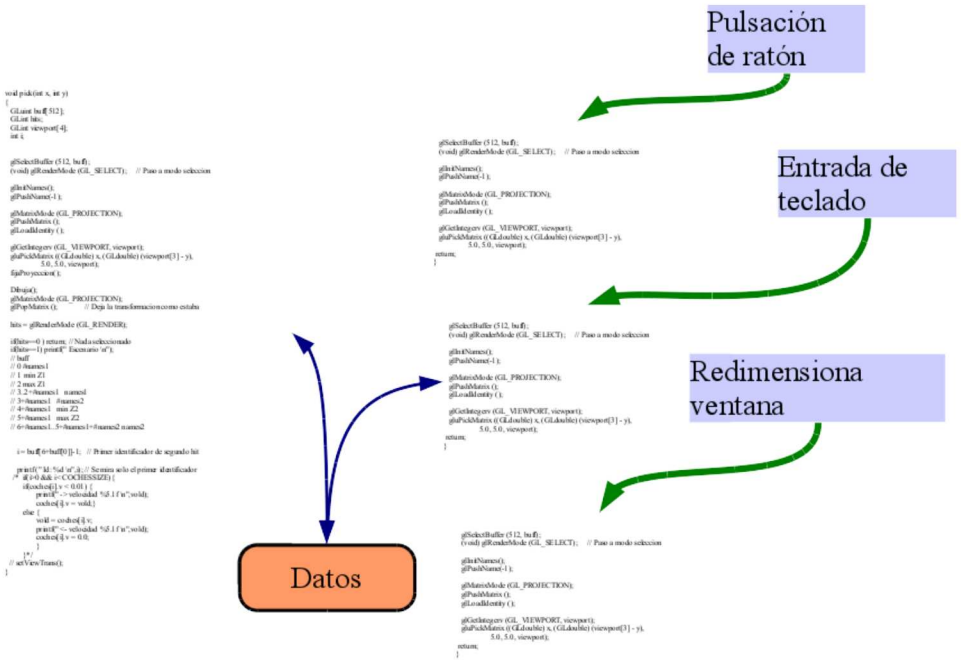

Figura 4: Estructura de un programa en un paradigma orientado a eventos

eventos pueden ser externos, generados por el usuario (como acciones en dispositivos de entrada) o generados por el propio programa. El código se estructura en un conjunto de procedimientos (llamados usualmente callback), cada callback se ejecuta cuando se produce un evento determinado (ver figura 4).

En el ejemplo de la figura 4 hay tres eventos. La asociación de cada evento con el callback correspondiente se realiza en el programa principal, concretamente, usando GLUT se haría con las llamadas:

```
glutReshapeFunc( inicializaVentana );
glutKeyboardFunc(letra);
glutMouseFunc( clickRaton );
```

en el que inicializaVentana es la función que se ejecutará cuando se redimensione la ventana, letra el callback de teclado y clickRaton el callback de pulsación de botones del ratón.

Entre los eventos que podemos procesar en una aplicación que use GLUT se encuentran:

- Cambio de tamaño de ventana
- Entrada de letras
- Entrada de caracteres especiales (teclas de función, cursor, escape, ...)
- Pulsación de ratón
- Movimiento de ratón

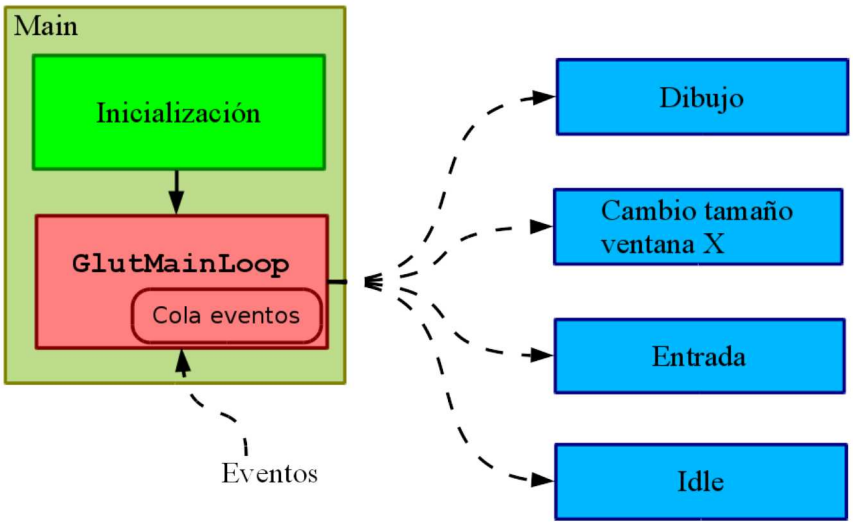

Figura 5: Estructura de un programa en un paradigma orientado a eventos

- Ausencia de eventos pendientes de procesar (el callback se ejecuta si no hay eventos pendientes)
- Eventos cronometrados
- Necesidad de redibujar (se genera llamando a la función glutPostRedisplay)

Dado que el orden de ejecución no está prefijado, es necesario guardar el estado del programa en variables globales. Del mismo modo, cualquier paso de información entre callback se debe realizar usando variables globales.

El programa principal debe inicializar GLUT, crear la ventana de dibujo, conectar los callback y lanzar el gestor de eventos de GLUT (llamando a glutMainLoop). A continuación se muestra un ejemplo de programa principal simple

```
int main( int argc, char *argv[] ) {
// Inicializa glut y OpenGL
glutInit( &argc, argv );
// Crea una ventana X para la salida grafica
glutInitWindowPosition( 600, 300 );
glutInitWindowSize( 300, 300 );
glutInitDisplayMode( GLUT_RGBA | GLUT_DOUBLE );
glutCreateWindow("FGGC: cubo");
//Inicializa las funciones de dibujo y cambio de ta-
maño de la ventana X
glutDisplayFunc( Dibuja );
glutReshapeFunc( Ventana );
// FUNCIONES DE INTERACCION
CreaMenu();
glutKeyboardFunc(letra);
```

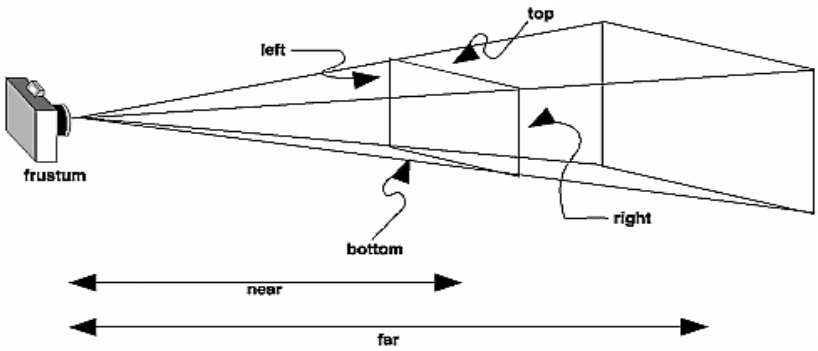

Figura 6: Proyección perspectiva

```
glutSpecialFunc(especial);
glutIdleFunc(idle);
glutMainLoop();
return 0;
}
```

Dibujo con OpenGL

Para dibujar una escena 3D con OpenGL es necesario:

- Definir la cámara
- Crear fuente de luz
- Asignar propiedades de material
- Asignar normales
- Crear geometría

Los cuatro primeras acciones modifican el estado de OpenGL. Cuando se dibuja geometría se hace utilizando las normales, materiales, fuentes de luz y cámara previamente establecidas.

Cámara

Para definir la cámara es necesario especificar la proyección (equivalente a la focal de una cámara fotográfica) y la posición y orientación de la cámara. La posición y orientación se indican usando transformaciones geométricas que se explican en la sección siguiente.

La proyección puede ser paralela o perspectiva. Para utilizar una proyección perspectiva usamos la función glFrustrum y para una paralela la función glOrtho.

Para ambas funciones se debe indicar los límites de la zona que será visible desde la cámara (pirámide de visión), ver figuras 6y7.

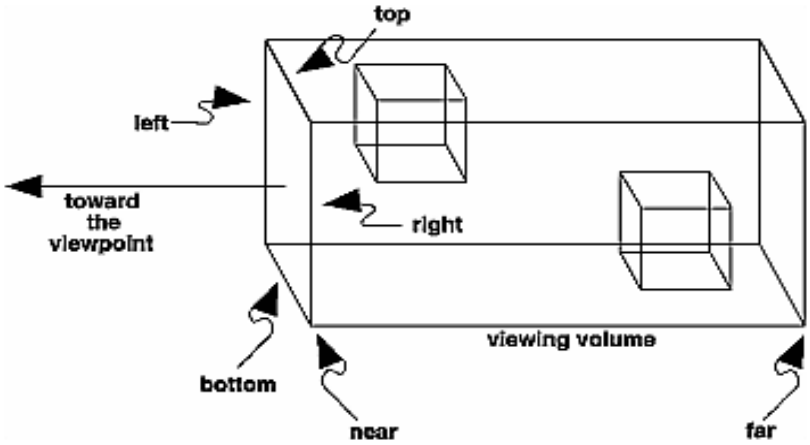

Figura 7: Proyección paralela

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

Modelo de iluminación

El modelo utilizado por defecto en OpenGL es el de Lambert, que es un modelo simple que permite calcular iluminación difusa y especular de forma local. OpenGL calcula la contribución al color de la reflexión difusa, especular y de la radiación de fondo en el ambiente y las suma para determinar el color con el que se dibuja:

$$Color_s = Obj_{amb} I_{amb} + Obj_{dif} I_{dif} + Obj_{esp} I_{esp}$$

$Color_s$es el color calculado para el punto y Obj es el color asignado al objeto (usando la función glMaterial). En el color del objeto se especifica de forma independiente el color con el que se ven la reflexión difusa, especular y ambiente. R_{amb}es la intensidad ambiente, I_{dif} e I_{esp} son las intensidades reflejados por el objeto que se calculan de acuerdo con el modelo de Lambert

$$I_{dif} = R_{dif} cos(\alpha) = R_{dif} max(\vec{L}\vec{N}, 0)$$
$$I_{esp} = R_{esp} max((\vec{H}\vec{N})^n, 0)$$

R_{dif} y R_{esp} son la intensidad difusa y especular de la fuente de luz. A cada fuente de luz se le asigna su contribución al cálculo de cada componente independientemente. Esta división es un tanto extraña, pero permite controlar de forma flexible que fuentes de luz se utilizan para calcular la iluminación ambiente, difusa y especular. Tanto los colores como las intensidades se especifican como cuartetos (RGBα), con valores de componentes normalizados entre 0 y 1.

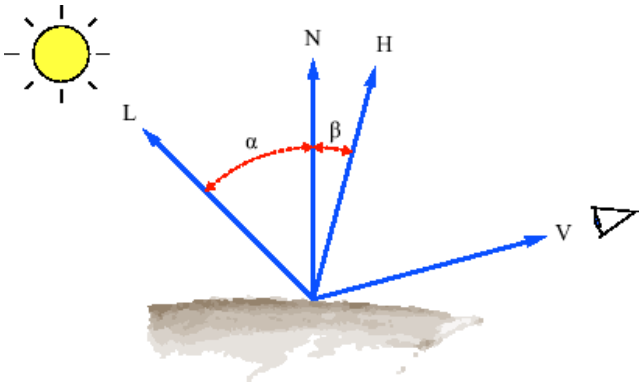

Figura 8: Cálculo de iluminación

Creación de fuentes de luz

Las fuentes de luz tienen identificadores de la forma GL_LIGHTn con n al menos entre 0 y 7. Solo la luz 0 está inicializada. Para modificar o asignar los parámetros usamos la función glLightfv, indicando el identificador de la luz, el parámetro a modificar y el valor asignado. Para usar una luz debemos dar sus intensidades y posición. Las posiciones se dan como cuartetos en coordenadas homogéneas. La cuarta componente puede ser 1 (para una luz en el punto indicado por las tres primeras componentes o 0 para una luz direccional que incide en la dirección dada por las tres primeras componentes. Las siguientes instrucciones modifican los parámetros de la luz 0:

```
GLfloat pos[4] ={10.0, 40.0, 10.0, 1.0 };
GLfloat inten[4] ={0.4, 0.4, 0.4, 1.0 };
GLfloat light_ambient[ ] = { 0.2, 0.2, 0.2, 1.0};
GLfloat light_specular[ ] = { 1.0, 1.0, 1.0, 1.0 };
glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
glLightfv( GL_LIGHT0, GL_DIFFUSE, inten);
glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
glLightfv( GL_LIGHT0, GL_POSITION, pos );
```

En necesario activar la iluminación y encender las luces que se quiera utilizar:

```
glEnable( GL_LIGHTING );
glEnable( GL_LIGHT0 );
```

Propiedades de material

La asignación de propiedades a los materiales se realiza de forma similar a la asignación de intensidades a las fuentes de luz. la función usada es glMaterial, cuyo primer argumento es la superficie a la que debe asignarse, el segundo es el parámetro y el tercero es el valor. Como ejemplo, las siguientes líneas asignan color ambiente y difuso a ambos lados de las caras y color especular a la cara delantera:

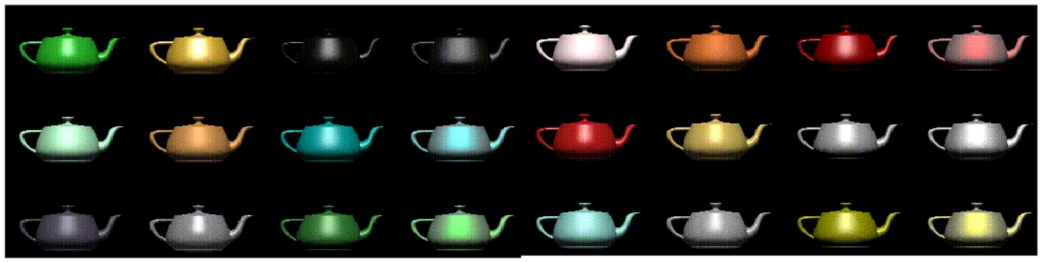

Figura 9: Ejemplo de materiales

```
float color[4]={0.2,0.7,1,1}; // R,G,B, Alfa
GLfloat mat_specular[ ] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat low_shininess[ ] = { 5.0 };
glMaterialfv( GL_FRONT_AND_BACK,GL_AMBIENT_AND_DIFFUSE, co-
lor );
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
glMaterialfv(GL_FRONT, GL_SHININESS, low_shininess);
```

Creación de geometría

OpenGL no almacena el modelo geométrico, es necesario almacenar el modelo en el programa, y dibujarlo desde el callback de redibujado. Para dibujar una primitiva se utilizan las funciones glBegin para indicar que se va a comenzar a dibujar una primitiva. El argumento indica la primitiva que queremos dibujar. A continuación se pasa a OpenGL los datos de la primitiva (como mínimo sus vértices). Al acabar de dibujar se debe llamar a glEnd. En un bloque glBegin..glEnd se pueden dibujar varias primitivas del mismo tipo. El código siguiente dibuja un cuadrilatero:

```
glBegin( GL_QUADS ); {
    glVertex3f( x, 0, 0 );
    glVertex3f( x, y, 0 );
    glVertex3f( x, y, z );
    glVertex3f( x, 0, z );
    }
glEnd();
```

y el siguiente dibuja dos:

```
glBegin( GL_QUADS ); {
    glVertex3f( x, 0, 0 );
    glVertex3f( x, y, 0 );
```

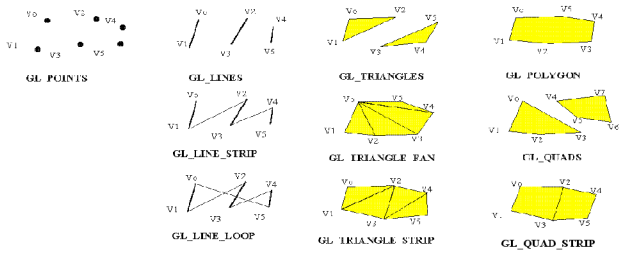

Figura 10: Primitivas básicas en OpenGL

```
        glVertex3f( x, y, z );
        glVertex3f( x, 0, z );
        glVertex3f( 2*x, 0, 0 );
        glVertex3f( x, 2*y, 0 );
        glVertex3f( x, y, 2*z );
        glVertex3f( 2*x, 0, 2*z );
        }

    glEnd();
```

OpenGL permite dibujar puntos, líneas, triángulos, cuadrilateros y polígonos convexos. Algunas de estos elementos se pueden dibujar con más de una primitiva. por ejemplo, los triángulos se pueden dibujar sueltos, formando un abanico o formando una tira. La figura 10muestra el resultado que se obtendría con las diferentes promitivas pasando los mismos seis vértices.

Las librerías auxiliares GLU y GLUT permiten dibujar objetos mas complejos llamado a las primitivas de OpenGL.

Normales

Para visualizar un modelo 3D es necesario calcular el color con que se debe mostrar en función de la iluminación. Tal como se mostró en la sección , el cálculo de la iluminación depende de la normal a las superficies.

La normal a un polígono convexo (y como caso particular a triángulos y cuadrilateros) se puede calcular realizando el producto vectorial de los vectores definidos por dos de sus aristas, tal como se muestra en la figura 11.

OpenGL puede utilizar dos modos de sombreado, en uno de ellos (sombreado plano) realiza el cálculo de iluminación una sola vez para cada polígono, en el otro (sombreado

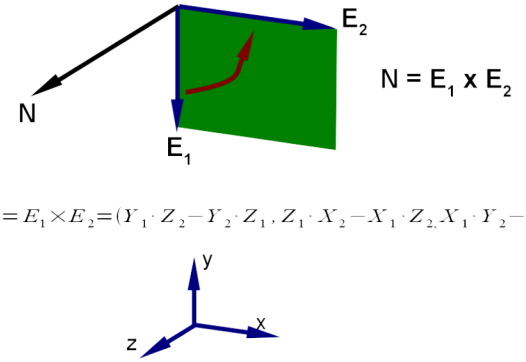

Figura 11: Calculo de normales

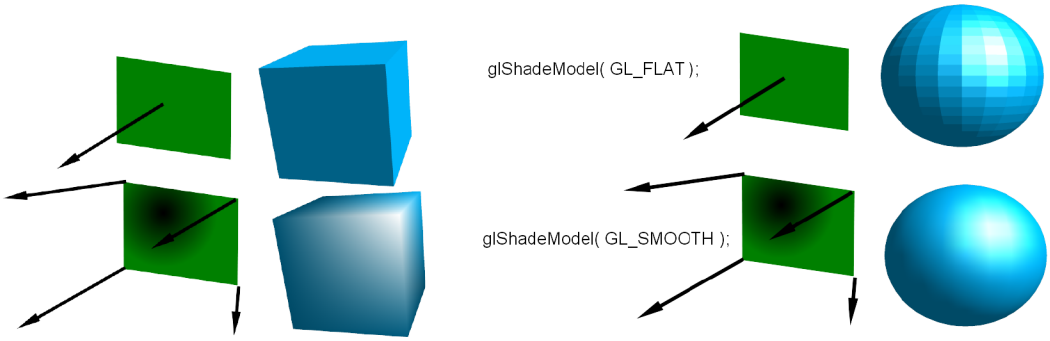

Figura 12: Modos de sombreado

suave) lo realiza una vez en cada vértice del polígono. En este último caso, el color en el polígono es una interpolación de los colores en los vértices.

El modo de sombreado se selecciona con la función glShadeModel (ver figura 12).

En cualquier caso, es necesario pasar las normales a OpenGL usando la función glNormal. Cuando se utiliza sombreado plano es necesario pasar una normal por cara:

```
glShadeModel( GL_FLAT );
glBegin( GL_QUADS );{

    glNormal3f(1, 0, 0 );
    glVertex3f( x, 0, 0 );
    .....
    glVertex3f( x, 0, z ); }

glEnd();
```

Si utilizamos sombreado suave debemos pasar una normal para cada vértice:

```
glShadeModel( GL_SMOOTH );
glBegin( GL_QUADS );{
```

```
        glNormal3f(1, 0, 0 );
        glVertex3f( x, 0, 0 );
        .....
        glNormal3f(1, 0, 0 );
        glVertex3f( x, 0, z ); }
    glEnd();
```

Transformaciones

Para construir modelos complejos es usual realizar un diseño ascendente, partiendo de componentes simples que se colocan posteriormente en la posición y orientación correcta en la escena.

Para aplicar una transformación geométrica a una primitiva debemos llamar a la función que la crea:

```
glTranslatef( X, Y, Z );
glRotatef(ang, X, Y, Z); // ang es el ángulo en gra-
dos.
// (X,Y,Z) es el eje de rotación
glScalef( X, Y, Z );
```

La transformación se queda almacenada en el estado de OpenGL y se aplicará a todas las primitivas que se dibujen a partir de ese momento. Por ejemplo, las lineas de código

```
glRotatef(90,0,1,0);
glutSolidTorus(0.5,3,24,32);
```

hacen que se dibuje un toro rotado 90º respecto al eje Y.

Las transformaciones geométricas se almacenan como matrices 4x4 (en el módulo 2 se estudian con más detalle la transformaciones geométricas). OpenGL guarda dos transformaciones geométricas diferentes: transformación de modelado y transformación de proyección. La segunda contiene las transformaciones necesarias para pasar del sistema de coordenadas usado en la escena 3D a la ventana de visualización 2D, y se aplican después de las transformaciones de modelado. Podemos decidir en cual de las dos transformaciones se guardan las transformaciones que creamos con la función glMatrixMode:

```
glMatrixMode( GL_MODELVIEW );
glMatrixMode( GL_PROJECTION);
```

Las funciones glFrustrum glOrtho crean transformaciones geométricas de proyección y se deben llamar estando en modo GL_PROJECTION.

Para cada una de las transformaciones (MODELVIEW y PROJECTION) OpenGL guarda una pila. Las transformaciones que se aplican se componen con al transformación que se encuentra en la cabecera de la pila. Podemos eliminar la transformación que se encuentra en la cabecera cargando la transformación identidad en ella

```
glLoadIdentity();
```

También podemos apilar y desapilar las transformaciones de la pila con las funciones

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

Desarrollo

Las prácticas se explicarán al comienzo de la primera sesión dedicada a cada una de ellas, a las que se debe asistir habiendo leido el guión correspondiente. En el guión de cada práctica se especifica sus fechas de realización y la fecha límite de entrega.

Evaluación

La evaluación de las prácticas se realizará en base a las entregas y sus correspondientes defensas, en las que se plantearán cuestiones, problemas o modificaciones sobre el código entregado, para evaluar la comprensión de los conceptos. En el guión de cada práctica se especifica la evaluación de las entregas.

La defensa de las prácticas se realizará una vez terminado el plazo de entrega. Las fechas de defensa se anunciarán con una antelación mínima de 10 días. Se realizarán al menos dos defensas en el curso.

Bibliografía

Manual de referencia de **OpenGL ver. 2.1**, incluyendo las llamadas de **GLU**:
`https://www.glprogramming.com/blue/index.html`

Manual de referencia de **OpenGL ver. 4.5**, incluyendo las llamadas de **GLU**:
`http://www.opengl.org/sdk/docs/man/`

Manual de referencia de la API de **GLUT**, versión 3:
`http://www.opengl.org/resources/libraries/glut/spec3/spec3.html`
`http://www.opengl.org/resources/libraries/glut/glut-3.spec.pdf`

Página web en sourceforge de **freeglut**, implementación open-source de la API de GLUT:
`http://freeglut.sourceforge.net/`

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:

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 práctcas).

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 respoden a eventos de ratón (Práctica 5).

visual.c: Contiene las funciones de proyección transformación de visualización, y el call-back 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 *entradaTclado.c*.

El código base se puede compilar usando el 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,X,y,Y o bien las teclas de movimiento del curso) y alejarla o acercarla (usando d y D o las teclas de avance y retroceso de página).

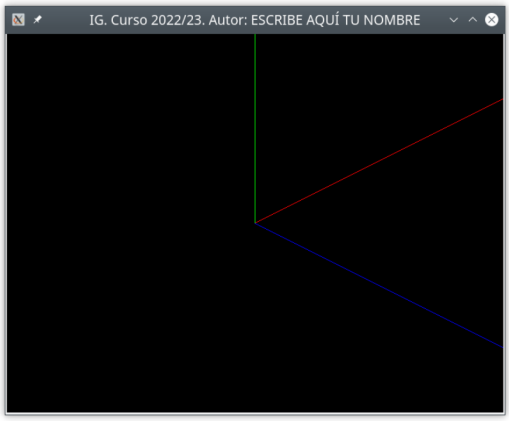

Figura 13: Ejecución del esqueleto de la práctica 1

Funcionalidad a desarrollar

Se deberá crear y visualizar una pirámide de base cuadrada y un cubo (ver Fig. 14), 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.

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
} ;
```

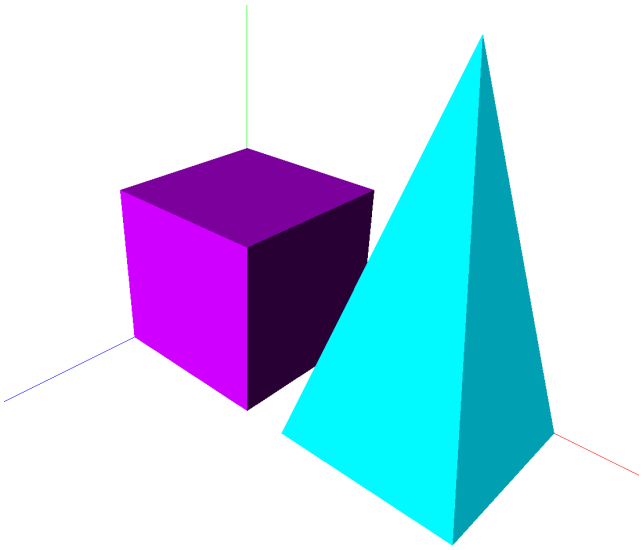

Figura 14: Escena creada en la práctica 1

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 un clase *Cubo*, con un constructor al que le puedas pasar el tamaño *Cubo*(*float lado*) en *modelo* e implementa su método *draw*. Puedes utilizar triángulos o cuadrilateros para dibujarlo. Llama al constructor del cubo desde *initModel* y al su método *draw* en la función *Dibuja* para que se dibuje en cada 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 recuerda 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. 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 inicializala 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*.

Evaluación

La evaluación de la práctica, sobre 10 puntos, se hará del modo siguiente:

- Hasta dos puntos por cada figura correcta (geometría y normales).
- Un punto por cada modo de visualización correcto.
- Un punto por utilizar colores diferentes en los dos modelos.
- Hasta dos puntos por crear figuras adicionales.

Temporización

Esta práctica se debe realizará en una sesión de prácticas:

Grupo C1 Jueves 22/09/22

Grupo C2 Miércoles 21/09/22