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

Prácticas de Informática Gráfica
Grupo C
Curso 2023/24

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

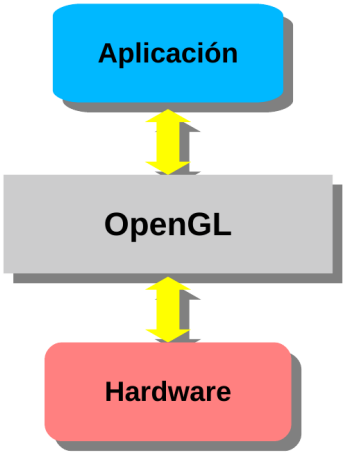

Figura 1: OpenGL es una API para la comunicación con el hardware gráfico

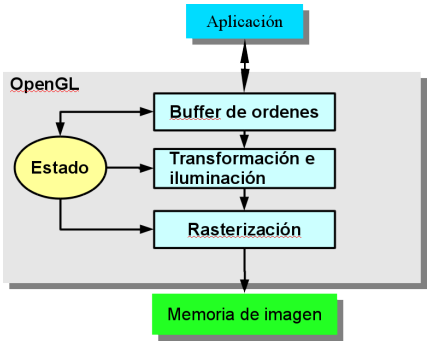

Figura 2: Esquema de funcionamiento de OpenGL

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

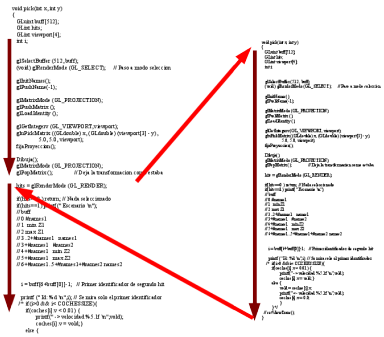

Figura 3: Secuencia de ejecución en un paradigma no orientado a eventos

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

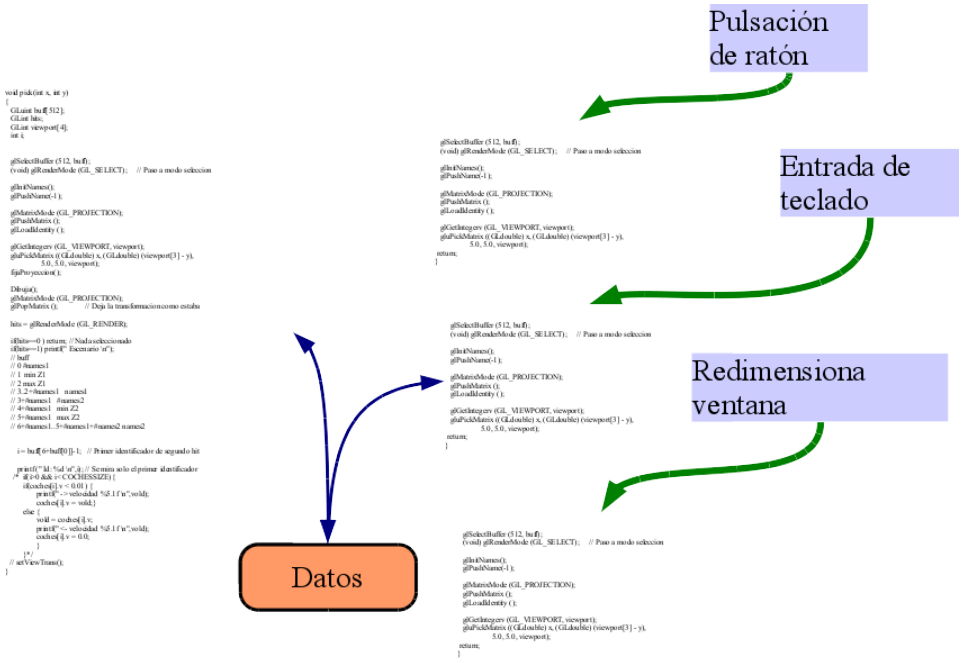

Figura 4: Estructura de un programa en un paradigma orientado a eventos

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

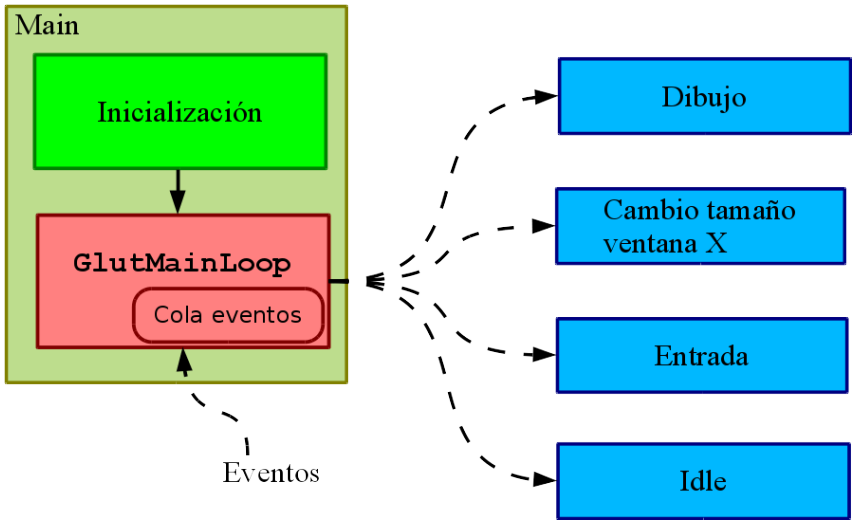

Figura 5: Estructura de un programa en un paradigma orientado a eventos

- Movimiento de ratón
- 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(\textquotedbl{}FGGC:~cubo\textquotedbl{});
//~Inicializa~las~funciones~de~dibujo~y~cambio~de~tamanyo~de~la~ventana
glutDisplayFunc(~Dibuja~);
glutReshapeFunc(~Ventana~);
//~FUNCIONES~DE~INTERACCION
CreaMenu();~~~
glutKeyboardFunc(letra);
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

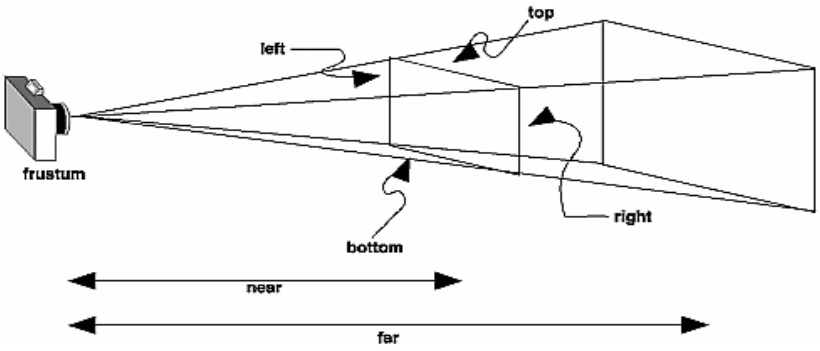

Figura 6: Proyección perspectiva

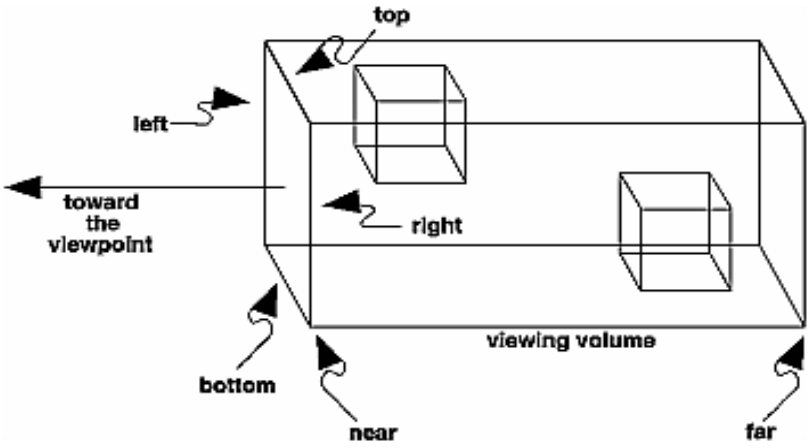

Figura 7: Proyección paralela

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.

```
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:

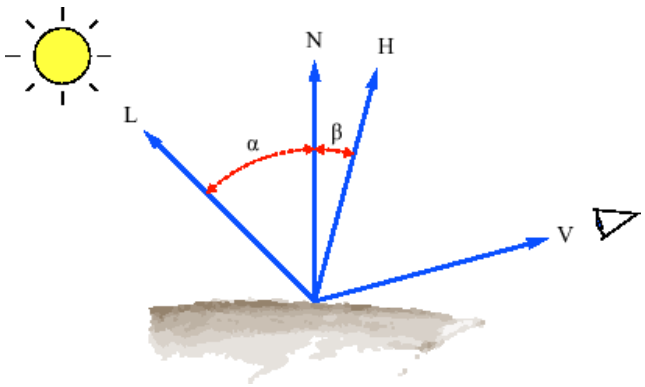

Figura 8: Cálculo de iluminación

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

$Color_s$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.

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:

```
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,colo

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

Figura 9: Ejemplo de materiales

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~);~~~~~~~~

                glVertex3f(~x,~y,~z~);~~~~~~~~

                glVertex3f(~x,~0,~z~);

                glVertex3f(~2{*}x,~0,~0~);~~~~~~~~

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

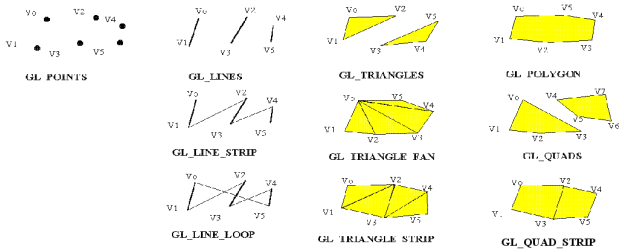

Figura 10: Primitivas básicas en OpenGL

```
        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 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~);
```

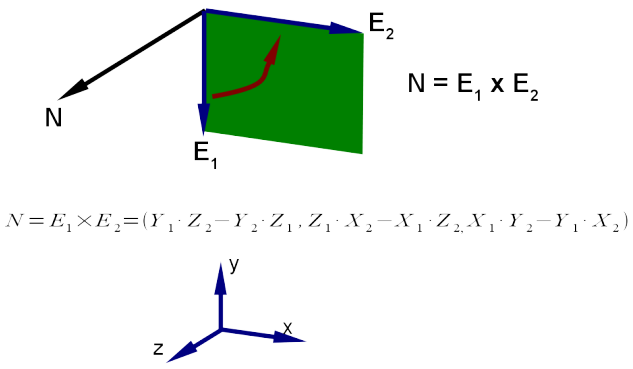

Figura 11: Calculo de normales

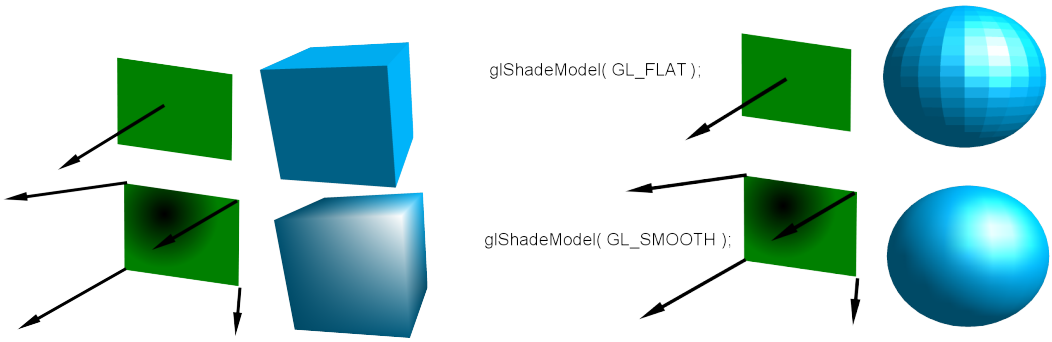

Figura 12: Modos de sombreado

```
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~angulo~en~grados.~

//~(X,Y,Z)~es~el~eje~de~rotacion

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).

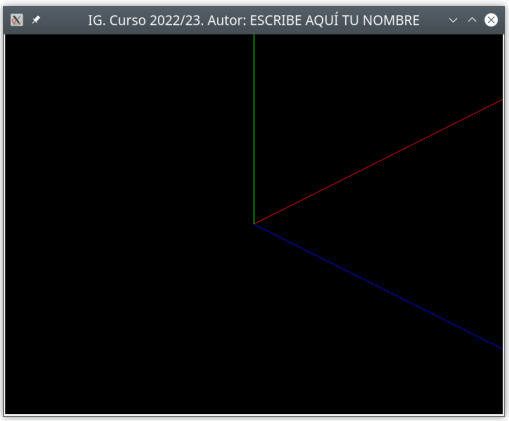

Figura 13: Ejecución del esqueleto de la práctica 1

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
} ;
```

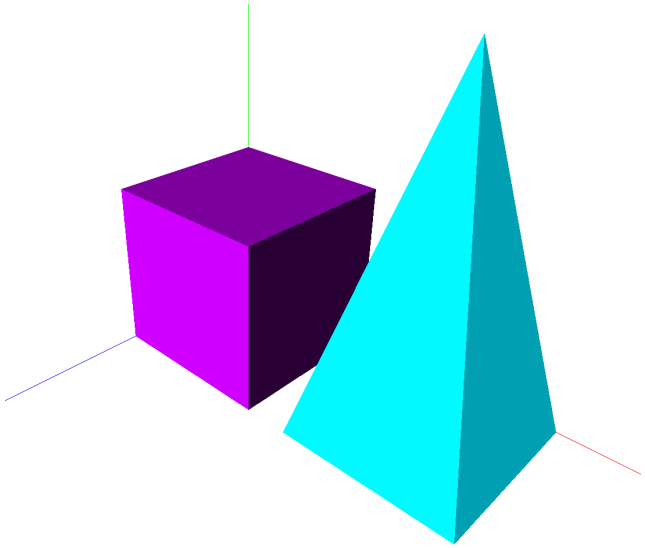

Figura 14: Escena creada en la práctica 1

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*) e implementa su método *draw*. Puedes utilizar triángulos o cuadrilateros para dibujarlo. Crea un cubo y llama a 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:

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
- Saber crear mallas de objetos de revolución a partir de la poligonal del perfil

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.
- **Superficies de revolución**. Se creará una subclase de malla de triángulo para representar superficies de revolución.
- **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 una superficie de revolución y dos ply uno dibujado en modo FLAT y el otro en modo SMOOTH.

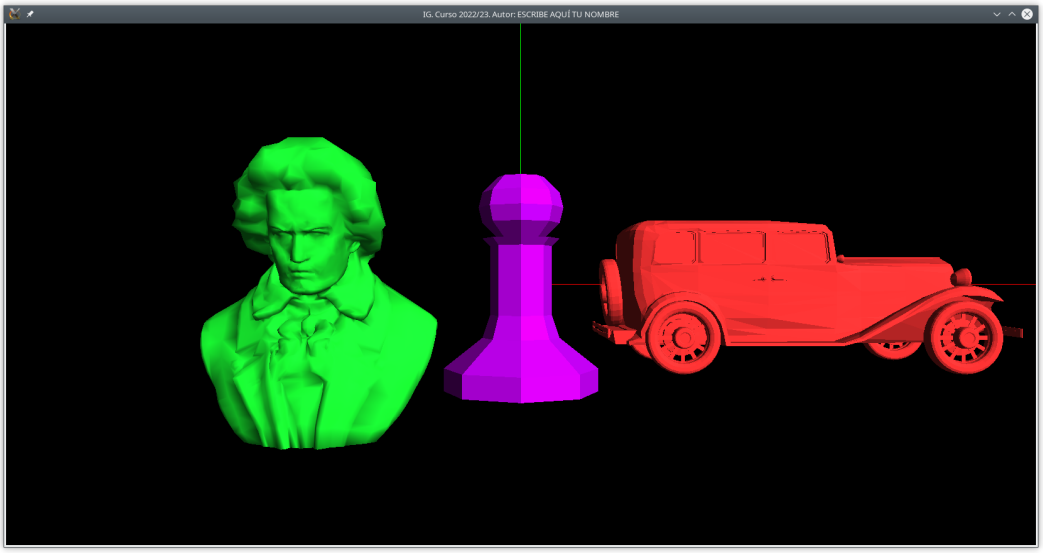

Figura 15: Escena creada en la práctica 2

La Figura 15 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.

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 ascci o binario. En cualquier

caso tendrá una cabecera en ascci 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"

...
std::vector<float> vertices_ply ; // coordenadas de vertices
std::vector<int>   caras_ply ;    // indices de vertices de triangulos

...
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 *encaras_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 1) y dividimos el resultado por su módulo para obtener un vector perpendicular y de módulo unidad (ecuación 2) .

$$\vec{N_0} = (\overrightarrow{(P_0,P_1)} x \overrightarrow{(P_0,P_2)}) \tag{1}$$

$$\vec{N} = \frac{\vec{N_0}}{modulo(\vec{N_0})} \tag{2}$$

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:

```
Inicializar normales de todos los vertice a (0,0,0)
Para cada cara
  sumar su normal a sus tres vertices
Para cada vertice
  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.

Superficies de revolución

Se desarrollará un método para la generación procedural de una malla obtenida por revolución de un perfil poligonal alrededor del eje Y. Dicho algoritmo tiene como parámetros de entrada el perfil, y el número de copias del mismo que servirán para crear el objeto.

Puedes crear una subclase de Malla para la que este método sea el constructor.

El perfil de entrada es una secuencia de *m* vértices en 3D en el plano XY (su coordenada z es cero). El perfil inicial se puede leer de un fichero PLY cuyo contenido sólo ha de tener las coordenadas de los vértices. Este fichero PLY puede escribirse manualmente, o bien se puede usar el que se proporciona en el material de la práctica. Para leer el perfil puedes usar la función *ply* :: *read_vertices*

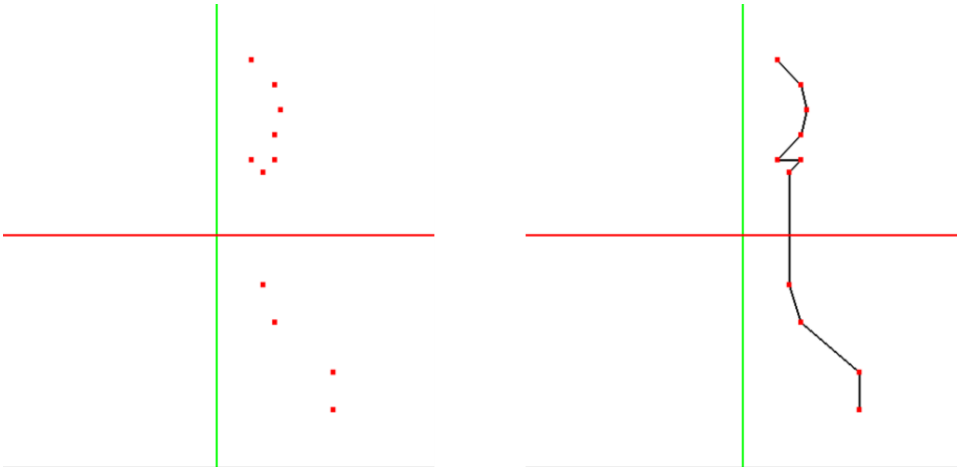

Figura 16: Ejemplo de perfil (puntos a la izquierda, poligonal a la derecha).

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

...
std::vector<float> vertices_ply ; // Coordenadas de vertices
...
ply::read_vertices( "nombre-archivo", vertices_ply );

...
```

Algoritmo de creación

Para la creación de un objeto de revolución, lo más fácil es crear en primer lugar la lista de vértices (partiendo del perfil original) y en segundo lugar la lista de triángulos. Para simplificar la creación de la lista de triángulos se pueden duplicar los vértices del perfil original (de hecho hacerlo simplificará la aplicación de una textura en la práctica 4).

Supongamos que el perfil tiene m vértices, que nombramos como $(p_0,..,P_{m-1})$ Asumimos que los vértices se dan de abajo hacia arriba (si cambiase el sentido cambiaría la orientación de las caras). Normalmente los vértices tienen coordenadas X estrictamente mayor que cero, pero el algoritmo funciona bien si algún vértice tiene coordenada X igual a cero, en este caso se crearían triángulos degenerados (sin área), que OpenGL ignora durante la rasterización. La figura 16 muestra un ejemplo de perfil.

De ese perfil original haremos n replicas rotadas (a cada una de ellas la llamamos una instancia del perfil). El valor de n debe ser mayor que tres. Cada uno de estas instancias del perfil forma un ángulo de $2\pi/(n-1)$ radianes con la siguiente o anterior.

Las instancias del perfil se numeran desde 0 hasta $n-1$, por tanto, la i-ésima instancia forma un ángulo de $2\pi i/(n-1)$ radianes con la instancia número 0. Las instancias0 y $n-1$ tienen sus vértices en la mismas posiciones que el perfil original. Se van a crear nm vértices

en total. Dichos vértices se insertarán en la lista de vértices por instancias del perfil (es decir, todos los vértices de una misma instancia aparecen consecutivos), además las instancias se almacenan en orden (empezando en la instancia 0 hasta la $n-1$).

Por tanto,en la lista final de vértices, el j-ésimo vértice de la i-ésima instancia tendrá un índice en la lista igual a $im+j$ (donde i va desde 0 hasta $n-1$ y j va desde 0 hasta $m-1$).

Para crear los vértices, por tanto, bastará con hacer un bucle doble que recorre todos los pares (i, j) y en cada uno de ellos crea el vértice correspondiente y lo inserta al final de la lista de vértices.

El pseudo-código, por tanto, para la creación de la lista de vértices será como sigue: Partimos de la lista de vértices vacía.

```
Para cada i desde 0 hasta n-1 (ambos incluidos)
  Para cada j desde 0 hasta m-1 (ambos incluidos)
   q = Rotacion de P_j  2iPI/(n-1) radianes respecto al eje Y
   Agregar q al final de la lista de vertices.
```

Para crear la lista de caras basta hacer un bucle con un índice i que recorre las instancias. Para cada instancia recorremos sus vértices con otro bucle excepto el último de ellos. Por cada vértice visitado se insertan en la lista de caras dos triángulos adyacentes (que comparten la arista diagonal).

Es decir, pseudo-código para la lista de triángulos visita todos los vértices (excepto el último de cada instancia y los de la última instancia), y crea dos triángulos nuevos que son adyacentes a ese vértice:

```
Partimos de la lista de triangulos vacia
Para cada i desde 0 hasta n-2 (ambos incluidos)
  Para cada j desde 0 hasta m-2 (ambos incluidos)
    Sea k = i m + j
      Agregar triangulo formado por los indices k, k+m y k+m+1
      Agregar triangulo formado por los indices k, k+m+1 y k+1
```

Una vez creada la malla debemos calcular las normales. Podemos usar la misma función usada para las mallas creadas a partir de archivos PLY.

Creación de la escena

El programa final debe crear una escena como la de la figura 15.

Evaluación

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

- Dos puntos por la estructura de datos.
- Dos puntos por la creación de mallas a partir de archivos PLY.

- Dos puntos por la creación de mallas de superficies de revolución.
- Dos puntos por el cálculo de normales.
- Dos puntos por la visualización en modos FLAT y SMOOTH.
- Hasta dos puntos por crear figuras adicionales (p.e. barrido lineal).

Temporización

Esta práctica se debe realizará en tres sesiones de prácticas:

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 3 grados de libertad distintos (al menos deben aparecer giros y desplazamientos). Se puede tomar como ejemplo el diseño de una grúa semejante a la de ejemplo (ver figura 17) que tiene al menos tres grados de libertad: ángulo de giro de la torre, giro del brazo y altura del gancho.

Nodos terminales

El modelo puede incluir como nodos terminales:

- Objetos predefinidos en glut y glu

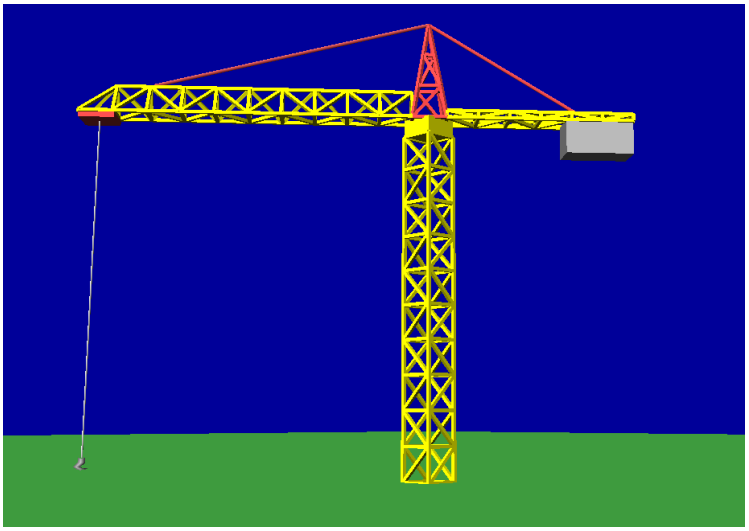

Figura 17: Ejemplo de modelo jerárquico

- 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.
- Las piezas usadas para construir la grua de la figura 17.

Las piezas de la grua se muestran en la figura 18. Se han contruido proceduralmente con funciones definidas en el módulo *estructura* que puedes descargar de Prado.

La definición de las funciones y el significado de sus parámetros se pueden consultar en el archivo *estructura.h*.

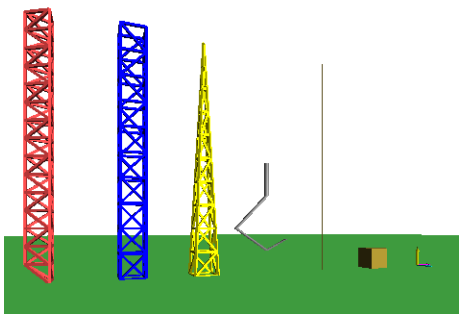

Figura 18: Estructuras empleadas para el ejemplo de las gruas.

Diseño del grafo de escena

El diseño del modelo se debe materializar en un grafo de escena (tipo PHIGS) según la notación vista en teoría, en un archivo PDF (que tendrás que entregar junto con el código de la práctica).

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 par 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.

Añadir código para modificar los parámetros de las articulaciones con 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. Ten 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;
```

Añadir código para 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.

Debes 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 (pulsando las teclas F,G,H,K,L).

Evaluación

- Grafo: claro, bien etiquetado,con transformaciones geométricas adecuadas, parámetros de transformaciones y croquis de submodelos (4 puntos)
- Modelo creado con articulaciones correctas (4 puntos).
- Modificación interactiva de parámetros (1 punto).
- Animación (1 punto).
- Modificación interactiva de velocidades de cambio de parámetros (2 puntos)

Temporización

Esta práctica se debe realizará en tres sesiones de prácticas:

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 y crear un método para asignar el material a las mallas.
- Modificar la representación del cubo para incorporar texturas, dibujando un dado.
- Modificar la representación de las mallas de revolución para incorporar coordenadas de textura y texturas.
- Crear una escena con una lata y tres copias del mismo objeto con diferentes materiales 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 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 $-ljpeg$ en tu 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 cubobjetola textura debes cargar la textura en OpenGL:

```
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.

Este proceso se debe realizar una sola vez (al asignarle la textura). 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:

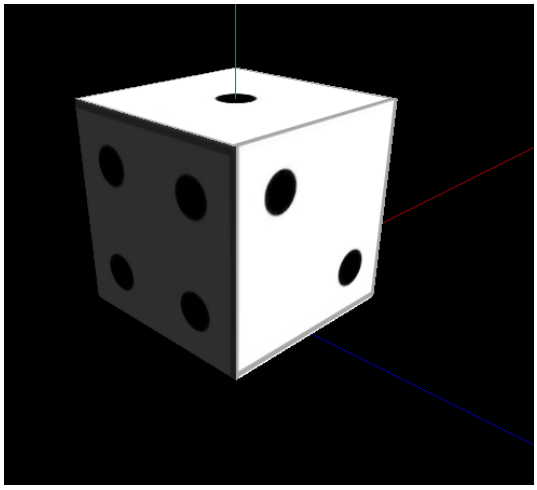

Figura 19: Imagen del dado

```
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 figura 19.

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 floats), siendo n_v es 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 de revolución

Añade un método a las mallas de revolución para calcular las coordenadas de textura de sus vértices para aplicar una textura que se corresponda con una proyección cilíndrica de la superficie (ver figura 20). Es decir calculando las coordenadas de textura haciendo que la u evolucione en el sentido angular y la v siguiendo el perfil, teniendo en cuenta que las coordenadas deben estar entre 0 y 1.

La componente u de la coordenada de textura (que recorre la imagen en sentido horizontal) se puede calcular a partir del ángulo:

$$u_i = \alpha_i/360 \tag{3}$$

Figura 20: Ejemplo de textura de la lata

siendo α el ángulo del perfil en grados. Este valor de u es el mismo para todos los vertices del perfil.

La componente *v* se calcula en función del desplazamiento en el perfil. Para ello, calculamos la distancia, d_i de cada vértice al inicio del perfil (este valor es el mismo para todos los vértices que se obtienen girando el mismo vértice del perfil):

$$d_i = \sum_{k=0}^{k=i-1} distancia(V_k, V_k + 1) \tag{4}$$

donde V_k es el vértice k-esimo del perfil.

Y calculamos la coordenada *v* de textura como

$$v_i = \frac{d_i}{d_{max}} \tag{5}$$

siendo d_max la longitud total del perfil (el valor de d_i del último punto del perfil).

Dibujo con textura

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

Visualización de una lata

Construye la superfice lateral de una lata, usando el poerfil $lata - pcue.pl$). Busca una imagen apropiada en internet, recortala para dejar solo la parte de la etiqueta lateral y escalalá para que tenga un tamaño potencia de 2 (puedes hacer estas operaciones con cualquier programa de dibujo o de manipulación de imágenes como Gwenview).

Creación de la escena

Crea una escena con al menos una lata, el dado y tres copias del mismo objeto con diferentes materiales y dos fuentes de luz de diferente tipo, color y posición.

Hacer que cada luz se encienda o se apague pulsando el número correspondiente (1 o 2).

Evaluación

- Representación y visualización de materiales (2 puntos).
- Aplicar texturas a mallas de revolución (3 puntos).
- Aplicar texturas al cubo (3 punto).
- Añadir fuentes de luz (1 punto).
- Composición de la escena (1 puntos).
- Añadir la tapa y la base a la lata (2 puntos).

Temporización

Esta práctica se debe realizará en tres sesiones de prácticas:

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

Teniendo en cuenta la reducción de horas lectivas de este curso parte de la práctica será optativa

1. opcionalAñadir vistas con proyección en planta, alzado y perfil.
2. ObligatoriaMover la cámara usando el ratón.
3. ObligatoriaSeleccionar objetos de la escena con el ratón.
4. opcional Añadir un menú de acciones que permita cambiar el material de los objetos seleccionados.

Desarrollo

Añadir vistas con proyección en planta, alzado y perfil opcional

Se incluirá en la escena un mecanismo para cambiar la proyección ofreciendo las vistas clásicas de frente, alzado y perfil de la escena. Al menos una de ellas deberá tener proyección ortogonal y al menos otra proyección en perspectiva. Se activarán con las teclas F1, F2 y F3.

Mover la cámara usando el ratón y el teclado en modo primera persona

Con las teclas A,S,D y W se moverá la cámara activa por la escena (W: avanzar, S: retroceder, A: desplazamiento a la izquierda, D: desplazamiento a la derecha, R: reiniciar posición), conservando la dirección en la que se está mirando. Para ello será necesario modificar únicamente el punto VRP o eye (esto equivale a modificar la traslación al sistema de coordenadas de la cámara). Para realizar este desplazamiento es necesario calcular la dirección en la que está mirando la cámara.

Por otro lado, girar la cámara a derecha o izquierda, arriba o abajo se realizará siguiendo los movimientos del ratón con el botón central pulsado. Esto implica modificar el VPN o el lookAt o el giro de la transformación de visualización.

Para controlar la cámara con el ratón es necesario hacer que los cambios de posición del ratón afecten a la posición de la cámara, y en glut eso se hace indicando las funciones que queremos que procesen los eventos de ratón. En el programa principal del código de partida aparece:

```
glutMouseFunc( clickRaton );
glutMotionFunc( ratonMovido );
```

Estos callback están declarados en *mouse.c*.

La función clickRaton será llamada cuando se actúe sobre algún botón del ratón. La función ratonMovido cuando se mueva el ratón manteniendo pulsado algún botón.

El cambio de orientación de la cámara se gestionará en cada llamada a ratonMovido, que solo recibe la posición del cursor, por tanto debemos almacenar el estado de los botones del ratón cada vez que se llama a clickRaton, para que solo se mueva la cámara cuando este pulsado el botón central. La información de los botones se recibe de glut cuando se llama al callback:

```
void clickRaton( int boton, int estado, int x, int y );
```

por tanto bastará con analizar los valores de boton y estado, y almacenar información que nos permita saber si el botón central está pulsado y la posición en la que se encontraba el cursor cuando se pulsó

```
if ( boton == GLUT_MIDDLE_BUTTON )
{
if ( estado == GLUT_DOWN ){
// Se entra en el estado "moviendo camara"
  }
else{
// Se sale del estado "moviendo camara"
  }
}
```

En la función ratonMovido comprobaremos si el botón central está pulsado, en cuyo caso actualizaremos la posición de la cámara a partir del desplazamiento del cursor

```
void ratonMovido( int x, int y )
```

```
{
......
if (MOVIENDO_CAMARA) {
//Girar la camara usando segun el vector (x-xant,y-yant);
  xant=x;
  yant=y;
  }
...
glutPostRedisplay();
```

Para girar la cámara se debe recalcular el valor de sus parámetros (VPN o lookAt) en función del incremento de x e y recibido. Si hasta ahora en la práctica la transformación de visualización se hacía, por ejemplo, en

```
void transformacionVisualizacion ()
{
  glTranslatef (0, 0, -D);

  glRotatef (view_rotx, 1.0, 0.0, 0.0);
  glRotatef (view_roty, 0.0, 1.0, 0.0);

  // glTranslatef(-x_camara,-y_camara,-z_camara);
}
```

ahora habrá que hacer cambiar *view_rotx* y *view_roty*.

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, y cuando el usuario hace clic, se pinta la escena "para seleccionar" en el buffer trasero y se lee el color del pixel donde el usuario ha hecho click. 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 *unsignedbyte* con *glColor3ub*, es decir, como enteros de 0 a 255.

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, *y* es la esquina inferior izquierda del cuadrado a leer (en nuestro caso el x,y, del pick).

width, *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 pixel. 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;
}
```

Añadir menú de acciones que permitan cambiar el objeto seleccionado opcional

Se incluirá un menú de glut que permitirá al menos asignar el color, de entre varios predefinidos, al objeto seleccionado.

Opcionalmente se podrá usar el menú para para transformar un objeto o para actuar sobre la articulación asociada a un componente de un objeto articulado. En este caso se puede además hacer que la transformación o el parámetro de la articulación se modifique con el ratón.

Evaluación

- opcional Añadir vistas de planta, alzado y perfil (2 puntos)

- Mover la cámara y rotarla con el ratón (~~3~~ 5 puntos)
- Selección de objetos(~~3~~ 5 puntos)
- opcional Cambio de material al seleccionar (2 puntos)
- opcional Transformación de objetos y cambio de parámetros de articulaciones (4 puntos)

Las partes opcionales solo se evaluan si se ha realizado la parte obligatoria

Temporización

Esta práctica se debe realizará en ~~tres~~ dos sesiones de prácticas: