

# Apuntes-de-OpenGL.pdf



CAZZ



Informática Gráfica



3º Grado en Ingeniería Informática



Escuela Técnica Superior de Ingenierías Informática y de  
Telecomunicación  
Universidad de Granada



**El más PRO del lugar  
puedes ser Tú.**

¿Quieres eliminar toda la publi  
de tus apuntes?



¡Hazte PRO!

4,95€ / mes



**WUOLAH**



**El más PRO del lugar puedes ser Tú.**



**¿Quieres eliminar toda la publi de tus apuntes?**



**¡Fuera Publi!**  
Concéntrate al máximo



**Apuntes a full.**  
Sin publi y sin gastar coins

Para los amantes de la inmediatez, para los que no desperdician ni un solo segundo de su tiempo o para los que dejan todo para el último día.

**Quiero ser PRO**

4,95 / mes

# Apuntes de OpenGL y GLUT

*Cristina Cañero Morales*

Introducción .....	2
Librerías añadidas .....	2
De vértices a píxeles .....	2
Formato de las funciones .....	3
Funciones básicas para definir objetos .....	3
Primitivas geométricas básicas .....	3
Primitivas de objetos predefinidos .....	3
Color .....	4
Ejemplo de primitivas geométricas .....	4
Definiendo una escena .....	4
Rotación .....	5
Translación .....	5
Escalado .....	5
Multiplicación por cualquier matriz .....	5
Guardando matrices en la pila .....	5
Ejemplos de transformaciones geométricas .....	5
Definición y colocación de la cámara .....	6
Posición de la cámara .....	6
Tipo de proyección .....	7
Proyección ortogonal .....	7
Proyección perspectiva .....	7
Ejemplo de definición de la posición de la cámara y el tipo de proyección .....	8
Definición del viewport .....	9
Manteniendo la proporción de los objetos .....	9
Ejemplo de preservación de la proporción .....	9
Ocultaciones .....	10
Algoritmo de las caras de detrás .....	10
Algoritmo del Z-buffer .....	10
Combinando ambos algoritmos .....	11
Luces y materiales .....	12
Métodos de sombreado .....	13
Especificación de las normales .....	13
Fuentes de luz .....	13
Materiales .....	15
Estructura de un programa usando la GLUT .....	16
Ejemplo de control de eventos .....	16
Control del mouse .....	17
Ejemplo de control de mouse .....	18
Definición de menús .....	19
Ejemplo de creación de menús .....	19
Configuración del Visual C++ para usar la GLUT .....	21
Creación del ejecutable con Visual C++ 6.0 .....	21

## Introducción

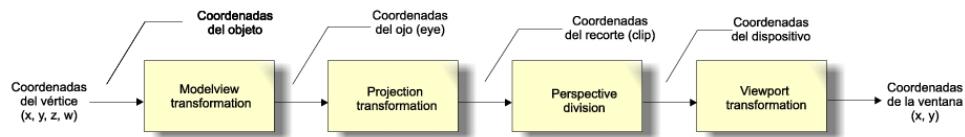
OpenGL es una **librería** de funciones que nos permiten visualizar gráficos 3D en nuestra aplicación. La gran ventaja de esta librería es que nos aísla del hardware disponible; por tanto, si nos ponemos una tarjeta aceleradora, no hará falta que cambiemos nuestro programa para aprovechar la potencia de la tarjeta.

## Librerías añadidas

En estas prácticas, no sólo vamos a trabajar con la librería OpenGL. Trabajaremos también con la **GLU** y la **GLUT**. La librería GLU contiene funciones gráficas de más alto nivel, que permiten realizar operaciones más complejas (una cosa así como el ensamblador y el C). En cambio, la librería GLUT es un paquete auxiliar para construir aplicaciones de ventanas, además de incluir algunas primitivas geométricas auxiliares. La gran ventaja de este paquete es que el mismo código nos servirá en Windows™ y en Linux™, además de simplificar mucho el código fuente del programa, como veremos en los ejemplos.

## De vértices a píxeles

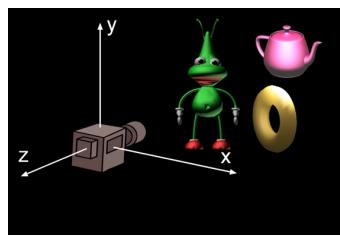
Desde que le damos a OpenGL unas coordenadas 3D hasta que aparecen en la pantalla, estas coordenadas son modificadas de la siguiente manera:



Las coordenadas 3D del vértice son coordenadas homegéneas, esto es,  $(x, y, z, w)$ . De todas maneras, en la mayoría de los casos pondremos simplemente  $(x, y, z)$ , puesto que, excepto en casos muy especiales,  $w=1$ . A estas coordenadas las llamaremos **coordenadas del objeto**, puesto que es el sistema de coordenadas en que se definen los objetos. Ahora bien, hay que montar una escena, y podríamos querer situar el objeto en diferentes posiciones, ya sea para poner varios objetos del mismo tipo en la misma escena, como para realizar animaciones. Por otro lado, dependiendo de donde situemos la cámara, veremos los objetos dispuestos de una u otra manera. Es por ello que existen las **coordenadas del ojo**. Para obtener las coordenadas del ojo a partir de las coordenadas del objeto, se realiza una transformación de modelo y vista, que consiste en la siguiente operación:

$$(x, y, z, w)_{\text{ojo}}^T = (x, y, z, w)_{\text{objeto}}^T \cdot M$$

donde **M** recibe el nombre de **matriz Modelview**. En realidad, esta transformación aglutina dos: por un lado, la obtención de las coordenadas de la escena, (o **coordenadas mundo**), y por otro, la transformación que se realiza para situar el punto de vista.



Por tanto, podríamos separar la matriz **M** en dos:

$$M = M_{\text{escena}} \cdot M_{\text{punto\_de\_vista}}$$

Una vez tenemos las coordenadas de la escena definidas respecto a la cámara, se realiza la “foto”, que consiste en multiplicar las coordenadas por una **matriz de proyección**, que denominaremos **P**. Esta matriz realiza una transformación afín del espacio de la escena, de tal manera que el **volumen de visualización** (es decir, la parte de la escena que quedará “fotografiada”) se deforma hasta convertirse en un cubo que va de  $(-1, -1, -1)$  a  $(1, 1, 1)$ . Por tanto, para seleccionar la parte de la escena que caerá en el

volumen de visualización, bastará con ver si queda dentro de estos límites. Es por esto que a estas coordenadas se las denomina **coordenadas de recorte o clipping**:

$$(x, y, z, w)^T_{\text{clipping}} = (x, y, z, w)^T_{\text{ojo}} \cdot P = (x, y, z, w)^T_{\text{objeto}} \cdot M \cdot P$$

Después, una vez realizado el recorte, se realiza la transformación de **perspective division**, que da como resultado **coordenadas del dispositivo (coordenadas de ventana-mundo)**.

Finalmente, se pasa de coordenadas de ventana-mundo a **coordenadas de viewport (coordenadas de ventana-vista)**, mediante la llamada **transformación del viewport**.

Resumiendo, para obtener una vista de una escena en nuestra pantalla tenemos que definir:

- La **matriz** del Modelview **M**, que define la colocación de los objetos en la escena y el punto de vista.
- La **matriz** de Proyección **P**, que define el tipo de proyección.
- La posición del **viewport** en coordenadas de pantalla.

Las matrices **M** y **P** por defecto son iguales a la identidad, mientras que la posición del viewport es en la esquina superior izquierda y ocupando toda la pantalla (en este caso, la ventana de ejecución).

### Formato de las funciones

Los nombres de las funciones en estas librerías siguen la siguiente convención:

{gl, glu, glut} <un nombre> [{d, f, u, ... etc}] [v]

El prefijo **gl** indica que se trata de una función de la librería de **OpenGL**, el prefijo **glu** de una función de la librería **GLU**, y el prefijo **glut** es para las funciones de la **GLUT**.

Los sufijos pueden aparecer o no, depende. Si los hay, es que la función puede estar en varias versiones, dependiendo del tipo de datos de los parámetros de ésta. Así, acabará con un sufijo **d** si recibe parámetros **double**, y con un sufijo **fv** si recibe parámetros de tipo **\*float** (es decir, punteros a float). En estas prácticas, por defecto referenciaremos las funciones con sufijo **f** y **fv**, aunque hayan más versiones.

Ejemplos de funciones: **gluPerspective**, **glColor3f**, **glutSwapBuffers**, **glMaterialf**, **glMaterialfv**... etc.

### Funciones básicas para definir objetos

#### Primitivas geométricas básicas

Podemos dibujar puntos, líneas y polígonos. Para definir un punto en el espacio 3D, usaremos la función: **glVertex3f(x, y, z)**. Estos puntos se unen formando estructuras, como líneas y polígonos. La siguiente tabla muestra alguna de las opciones que tenemos:

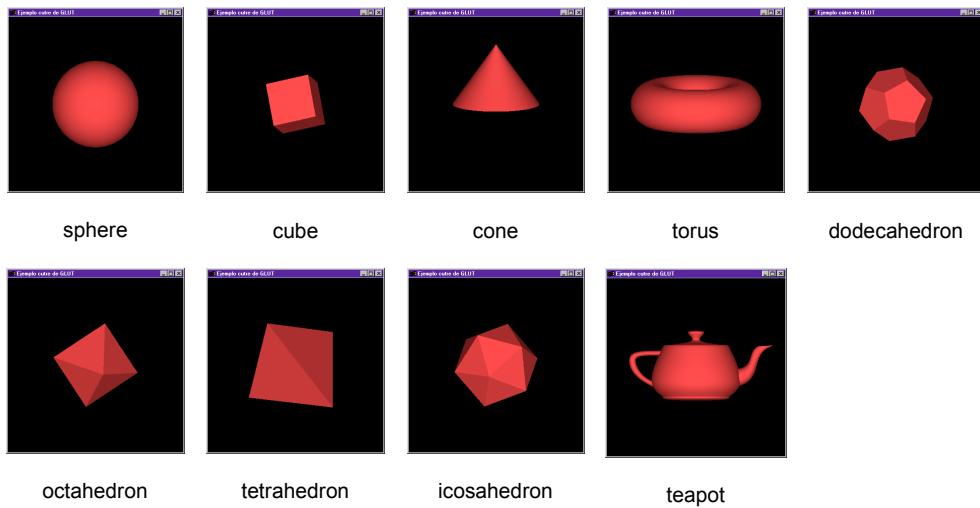
Modo	Descripción
GL_POINTS	Puntos individuales aislados
GL_LINES	Cada par de puntos corresponde a una recta
GL_LINE_STRIP	Segmentos de recta conectados
GL_LINE_LOOP	Segmentos de recta conectados y cerrados
GL_TRIANGLES	Cada tres puntos un triángulo
GL_QUADS	Cada cuatro puntos un cuadrado
GL_POLYGON	Un polígono

#### Primitivas de objetos predefinidos

Hay algunos objetos que vamos a renderizar muy a menudo, y que por tanto, ya vienen definidos. Así, disponemos de las siguientes funciones:

- **glutWireSphere(radius, slices, stacks)**, **glutSolidSphere(radius, slices, stacks)**
- **glutWireCube(size)**, **glutSolidCube(size)**
- **glutWireCone(base, height, slices, stacks)**, **glutSolidCone(base, height, slices, stacks)**
- **glutWireDodecahedron(void)**, **glutSolidDodecahedron(void)**
- **glutWireOctahedron(void)**, **glutSolidOctahedron(void)**
- **glutWireTetrahedron(void)**, **glutSolidTetrahedron(void)**

- glutWireCosahedron(void), glutSolidCosahedron(void)
- glutWireTeapot(void), glutSolidTeapot(void)



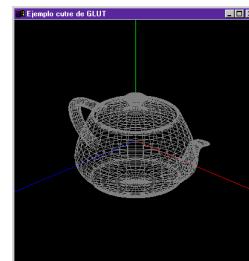
### Color

Por defecto, el trazado de las líneas y puntos es blanco, pero podemos cambiarlo. Para hacer esto, usaremos la función **glColor3f(r, g, b)**. El valor de r, g y b debe estar entre 0 y 1 (y no entre 0 y 255!).

### Ejemplo de primitivas geométricas

Vamos a dibujar una tetera de alambres en gris, con los ejes de coordenadas en los siguientes colores: rojo para **x**, verde para **y**, y azul para **z**.

```
glColor3f(0.5f, 0.5f, 0.5f);
glutWireTeapot(0.5);
glBegin(GL_LINES);
	glColor3f(1.0f, 0.0f, 0.0f);
	glVertex3f(0.0f, 0.0f, 0.0f);
	glVertex3f(2.0f, 0.0f, 0.0f);
	glColor3f(0.0f, 1.0f, 0.0f);
	glVertex3f(0.0f, 0.0f, 0.0f);
	glVertex3f(0.0f, 2.0f, 0.0f);
	glColor3f(0.0f, 0.0f, 1.0f);
	glVertex3f(0.0f, 0.0f, 0.0f);
	glVertex3f(0.0f, 0.0f, 2.0f);
glEnd();
```



### Definiendo una escena

Hasta ahora hemos construido nuestra escena mediante la especificación directa de las coordenadas 3D, o bien utilizando las primitivas de objetos. Pero, ¿cómo podríamos dibujar, por ejemplo, dos teteras?. Cuando utilizamos **glVertex3f**, o llamamos a la función **glWireTeapot**, estamos definiendo un objeto en **coordenadas del objeto**.

Estas coordenadas son multiplicadas por una matriz, denominada matriz **ModelView**. Esta matriz es de 4x4 y contiene las transformaciones necesarias de translación, rotación, escalado, ... etc. que definen la localización de nuestro objeto en el **mundo**. Por tanto, si llamamos a la función **glWireTeapot** con dos matrices **ModelView** diferentes, aparecerán en nuestra escena en dos sitios distintos.

OpenGL puede trabajar con varias matrices. Para poder modificar la matriz **ModelView**, tenemos que decirle a OpenGL que queremos trabajar con ella, porque si no el resultado puede ser impredecible, ya que no sabremos qué matriz estamos modificando. Esto lo haremos haciendo una llamada a **glMatrixMode(GL\_MODELVIEW)**.

# Estudiar sin publi es posible.

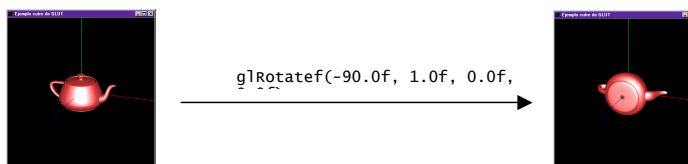
Compra Wuolah Coins y que nada te distraiga durante el estudio.



Una vez en modo modelview, podemos acumularle operaciones de inicialización a la identidad (llamando a `glLoadIdentity()`), de rotación, de translación, de escalado, etc....

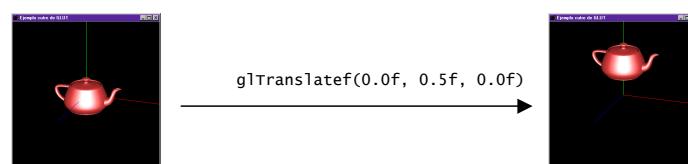
## Rotación

Se define mediante la primitiva `glRotatef(alpha, x, y, z)`. Esta función multiplica la matriz actual por una matriz de rotación de alpha grados respecto al eje (x, y, z).



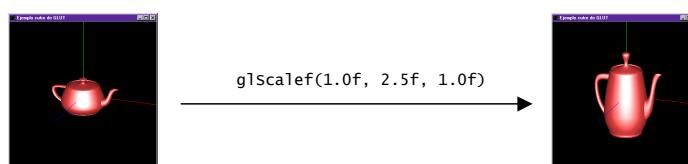
## Translación

Se define mediante la primitiva `glTranslatef(x, y, z)`. Aplica una translación en (x, y, z) sobre la matriz actual.



## Escalado

Se define mediante la primitiva `glScalef(sx, sy, sz)`. Escalado de cada uno de los ejes.



## Multiplicación por cualquier matriz

Si no nos bastara con estas transformaciones, podemos multiplicar la matriz actual por cualquiera. Esto lo hacemos con la función `glMultMatrixf`. En estas prácticas no hará falta usar esta función.

## Guardando matrices en la pila

Dado que las operaciones sobre las matrices son **acumulativas**, es necesario tener una manera de **recuperar** el estado anterior de la matriz.

OpenGL dispone de una pila para cada matriz; para la **matriz ModelView** el tamaño de esta pila es de al menos 32 matrices, mientras que para la **matriz Projection** es de al menos 2.

La función `glPushMatrix` nos permite guardar la matriz **actual** en la pila, mientras que la función `glPopMatrix` coge la matriz que se encuentre en el **top** de la pila y la asigna a la matriz **actual**.

## Ejemplos de transformaciones geométricas

```
ROTATE + TRANSLATE
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
for (i=0; i<10; i++) {
    glPushMatrix();
    glRotatef(i * 360 / 10, 0.0f, 1.0f, 0.0f);
    glTranslatef(0.5f, 0.0f, 0.0f);
    glutSolidTeapot(0.25);
    glPopMatrix();
}
```



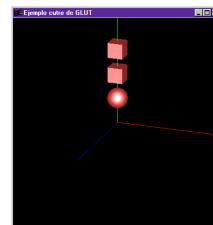
#### TRANSLATE + ROTATE

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
for (i=0; i<10; i++) {
    glPushMatrix();
    glTranslatef(0.5f, 0.0f, 0.0f);
    glRotatef(i * 360 / 10, 0.0f, 1.0f, 0.0f);
    glutSolidTeapot(0.25);
    glPopMatrix();
}
```



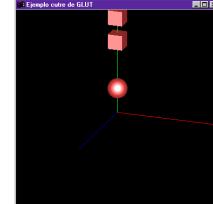
#### JUGANDO CON LA PILA 1

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glPushMatrix();
glTranslatef(0.0f, 0.25f, 0.0f);
glutSolidSphere(0.1, 30, 30);
glPopMatrix();
glPushMatrix();
glTranslatef(0.0f, 0.5f, 0.0f);
glutSolidCube(0.15);
glTranslatef(0.0f, 0.25f, 0.0f);
glutSolidCube(0.15);
glPopMatrix();
```



#### JUGANDO CON LA PILA 2

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glPushMatrix();
glTranslatef(0.0f, 0.25f, 0.0f);
glutSolidSphere(0.1, 30, 30);
glTranslatef(0.0f, 0.5f, 0.0f);
glutSolidCube(0.15);
glTranslatef(0.0f, 0.25f, 0.0f);
glutSolidCube(0.15);
glPopMatrix();
```



#### **Definición y colocación de la cámara**

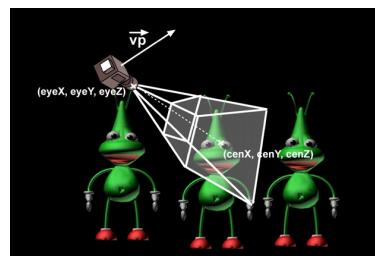
Una vez hemos definido toda nuestra escena en coordenadas mundo, tenemos que "hacerle la foto". Para ello, tenemos que hacer dos cosas: colocar la cámara en el mundo (o sea, en la escena) y definir el tipo de proyección que realizará la cámara.

#### **Posición de la cámara**

Tenemos que definir no sólo la posición de la cámara (o donde está), sino también hacia dónde mira y con qué orientación (no es lo mismo mirar con la cara torcida que recta... aunque veamos lo mismo). Para hacer esto, basta con modificar la matriz **ModelView** para mover **toda** la escena de manera que parezca que hemos movido la cámara. El problema de este sistema es que tenemos que pensar bastante las transformaciones a aplicar. Es por ello que la librería **GLU** viene al rescate con la función **gluLookAt**. Su sintaxis es la siguiente:

```
void gluLookAt( eyeX, eyeY, eyeZ, cenX, cenY, cenZ, vp_X, vp_Y, vp_Z);
```

donde **eye** corresponde a la posición de la cámara, **cen** corresponde al punto hacia donde mira la cámara y **vp** es un vector que define la orientación de la cámara. No podemos llamar a **gluLookAt** en cualquier momento, puesto que tiene **postmultiplicar** la matriz **ModelView** (por tanto, conviene llamarla lo primero de todo). El vector **vp** *no puede ser paralelo* al vector formado por **eye** y **cen**, es más, debería serle perpendicular. Si no, el resultado es impredecible.



### Tipo de proyección

A la vez de definir la posición y orientación de la cámara, hay que definir el tipo de "foto" que hace, un poco como si seleccionásemos el objetivo. Esto es lo que nos va a definir la forma del **volumen de visualización**.

La primera elección que tenemos que hacer es si queremos una proyección **ortogonal** o **perspectiva**. Dependiendo del tipo de aplicación, utilizaremos una u otra.

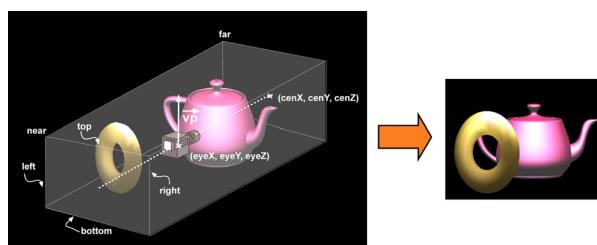
En OpenGL, para definir la proyección, modificaremos la matriz **Projection**. Para hacer esto, pues, tendremos que hacer una llamada a **glMatrixMode(GL\_PROJECTION)**, para indicarle que vamos a modificar esta matriz. A partir de ese momento, todas las funciones de transformación (**glRotatef**, **glTranslatef**, **glScalef**, ...etc) y las de pila (**glPushMatrix** y **glPopMatrix**) se aplican sobre estas matriz.

### Proyección ortogonal

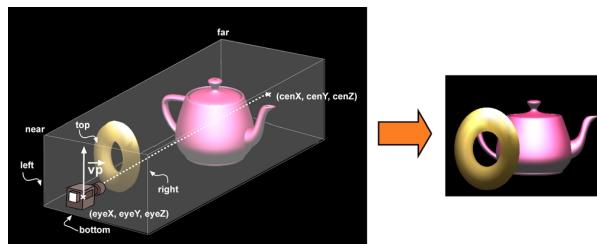
Para el caso de la proyección ortogonal la analogía de la cámara no es tan evidente, porque en realidad estamos definiendo una caja, o **volumen de visualización** alrededor del **eye** de la cámara (esto es, podemos ver detrás). Para hacer esto, llamamos a

**glOrtho(left, right, bottom, top, near, far)**

Como se aprecia en la figura, la "foto" obtenida incluye tanto la tetera como el toroide, puesto que el volumen de visualización tiene un **near** negativo (detrás de la cámara) y por tanto el volumen de visualización lo incluirá.



Esto sería equivalente a tener la cámara situada de la siguiente manera:



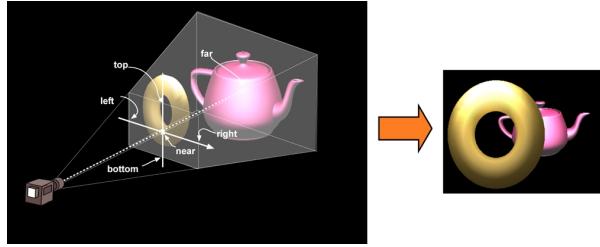
En este caso, la imagen sería la misma, pero **near** y **far** serían ambos positivos, puesto que el volumen se está definiendo por **delante** de la cámara. Si **near** y **far** fueran los valores de la figura anterior, entonces seguramente sólo aparecería en la imagen el toroide, puesto que el volumen de visualización estaría centrado en la cámara.

### Proyección perspectiva

El caso de la proyección perspectiva es más intuitivo, puesto que la analogía de la cámara se cumple bastante bien. Así, una vez hemos situado la cámara en la escena, definimos el **volumen de visualización** mediante la función:

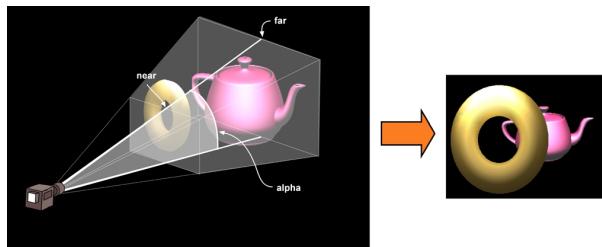
**glFrustum(left, right, bottom, top, near, far)**

Esta función, aunque tiene los mismos parámetros que **glOrtho**, éstos tienen un significado ligeramente distinto. Mientras que **left**, **right**, **bottom** y **top** siguen definiendo el tamaño y forma del plano **near** de proyección, los parámetros **near** y **far** ahora deben cumplir la relación  $0 < \text{near} < \text{far}$ . De hecho, en ningún caso **near** debe tomar el valor 0, puesto que los resultados serán impredecibles. El volumen de visualización generado es como se muestra en la figura:



Fijáos que en la imagen obtenida el toroide sale mucho mayor que la tetera, que está más lejos. Esta es la principal diferencia con la proyección ortogonal, que los rayos proyectivos no son paralelos.

Una manera más simple de definir la proyección perspectiva es mediante la función **gluPerspective(alpha, aspect, near, far)** Veamos esquemáticamente qué quieren decir cada uno de estos parámetros:



En esta función, **alpha** corresponde al ángulo de apertura del campo de visión en la dirección de **y**, **near** la distancia mínima a la que deben estar los objetos para salir en la foto y **far** la distancia máxima (ambas siempre positivas) El parámetro **aspect** define la relación entre el ángulo de visión horizontal y vertical (es la **anchura / altura**), y típicamente debería ser de entre 30 y 35 grados para obtener resultados realistas. La relación entre **far / near** debería ser lo más cercana posible a 1.

#### Ejemplo de definición de la posición de la cámara y el tipo de proyección

```

glMatrixMode(GL_PROJECTION);
glPushMatrix(); /* Guardamos la matriz de proyección */
glLoadIdentity(); /* P = I */
gluPerspective(30.0f, anchura / altura, 1.0, 10.0);

glMatrixMode(GL_MODELVIEW);
glPushMatrix(); /* Guardamos la matriz del modelo */
glLoadIdentity(); /* M = I; */

/* Colocamos la cámara en el punto deseado */
gluLookAt(0, 5, 5, 0, 0, 0, 0, 1, 0);

/* Dibujamos una tetera roja centrada en (-1, 0, 0) */
glTranslatef(-1.0f, 0.0f, 0.0f);
	glColor3f(1.0f, 0.0f, 0.0f);
	glutWireTeapot(0.5f);

glPopMatrix(); /* Recuperamos la antigua matriz del modelo */
glMatrixMode(GL_PROJECTION);
glPopMatrix(); /* Recuperamos la antigua matriz de proyección */

```



# Descarga la APP de Wuolah.

Ya disponible para el móvil y la tablet.

Available on the  
App Store

GET IT ON  
Google Play

## Definición del viewport

La definición del viewport se realiza de manera sencilla mediante la función `glViewport(x, y, cx, cy)`, donde (x, y) es la esquina superior izquierda del viewport en coordenadas pantalla y (cx, cy) corresponden a la anchura y altura del viewport deseado. Por defecto, la GLUT define el viewport ocupando toda la ventana.

### Manteniendo la proporción de los objetos

En la sección anterior hemos introducido el concepto de **aspect**, que corresponde a la relación **anchura / altura**. Cuando definimos el volumen de visualización, es importante que el aspect de éste sea el mismo que el de **viewport** (ventana de visualización).

Así, si definimos un volumen de visualización que es el doble de alto que de ancho (`aspect = 0.5`), y nuestro viewport es cuadrado (`aspect = 1`) la imagen que obtendremos podría tener la siguiente pinta:



Como se puede observar, aunque el volumen de visualización es suficientemente grande para contener todo el objeto, la imagen sale deformada. En cambio, si nuestro volumen de visualización tiene aspect 1, aunque estemos incluyendo más volumen del necesario para incluir nuestro objeto en su totalidad, la imagen obtenida será:



que no ha sufrido ninguna deformación, lo cual es interesante en la mayoría de aplicaciones.

La definición del aspect del volumen de visualización cuando se usa `gluPerspective` es trivial, basta con pasarle el parámetro adecuado. Para el caso de las funciones `glFrustum` y `glOrtho` hay que definir los valores left, right, top y bottom teniendo en cuenta que a relación que tienen que mantener entre ellos tiene que ser la misma que el ancho y alto de nuestro viewport.

### Ejemplo de preservación de la proporción

```
#include <gl/glut.h>

float aspect = 1.0f;
void display(void) {
    glClear(GL_COLOR_BUFFER_BIT);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(20.0f, aspect, 1.0f, 10.0f);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(0.0f, 0.0f, 5.0f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f);
    glutWireTeapot(0.5);
    glFlush();
    glutSwapBuffers();
}
void onSize(int sx, int sy) {
    glviewport(0, 0, sx, sy);
    aspect = (float) sx / (float) sy;
}
void main(void) {
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(400, 400);
    glutInitWindowPosition(100, 100);
    glutCreateWindow("Ejemplo de menus");
    glutDisplayFunc(display);
    glutReshapeFunc(onSize);
    glutMainLoop();
}
```

## Ocultaciones

OpenGL permite utilizar dos métodos de ocultación: el algoritmo de las caras de detrás (Back Face Removal), y el algoritmo del Z-buffer.

### Algoritmo de las caras de detrás

Consiste en ocultar las caras que no se dibujarían porque formarían parte de la parte trasera del objeto. Para decirle a OpenGL que queremos activar este método de ocultación, tenemos que escribir `glEnable(GL_CULL_FACE)`.



**Importante.** OpenGL ignora totalmente la normal que le podamos pasar con la función `glNormalf`, así que esta normal sólo se utiliza a efectos de iluminación. A efectos de ocultación de caras, OpenGL decide si oculta o no una cara a partir del orden de definición de los vértices que la forman. Así, si la proyección sobre del camino que forman los vértices del primero al último es en el sentido de las agujas del reloj, entonces se dice que el orden es **clockwise**. Si es en sentido inverso, se dice que es **counterclockwise**. Por defecto, OpenGL **sólo** renderiza aquellos polígonos cuyos vértices están en orden **counterclockwise** cuando son proyectados sobre la ventana. Pero los objetos definidos por la GLUT no tienen porqué estar definidos **clockwise** (como es el caso de la tetera). Por tanto, antes de renderizarlos usando este algoritmo de ocultación, debemos llamar a `glFrontFace(GL_CW)` o a `glFrontFace(GL_CCW)`.

### Algoritmo del Z-buffer

El algoritmo del Z-buffer es del tipo espacio-imagen. Cada vez que se va a renderizar un pixel, comprueba que no se haya dibujado antes en esa posición un pixel que esté más cerca respecto a la cámara. Este algoritmo funciona bien para cualquier tipo de objetos: cóncavos, convexos, abiertos y cerrados. Para activarlo, hay que hacer una llamada a

`glEnable(GL_DEPTH_TEST)`

Esta llamada le dice a OpenGL que active el test de profundidad. Además, cada vez que se redibuja la escena, a parte de borrar el buffer de color, hay que borrar el buffer de profundidad. Esto se hace con la llamada

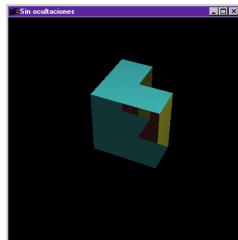
`glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT).`

Por último, pero no menos importante, al inicializar OpenGL se le tiene que decir que cree el buffer de profundidad. Esto se hace al definir el modo de visualización (ver capítulo GLUT).

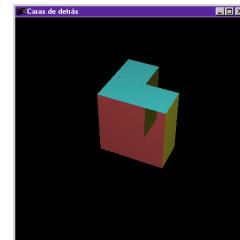


## Combinando ambos algoritmos

El algoritmo de la eliminación de las caras de detrás es suficiente para renderizar para objetos convexos y cerrados, como este octahedro, pero para objetos cóncavos y cerrados no basta. Por ejemplo, un objeto con una cara parcialmente oculta, como el de la figura, no tiene porqué renderizarse bien:

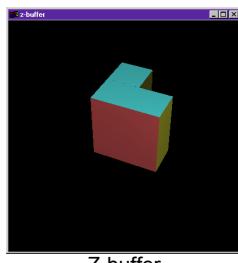


Sin ocultaciones

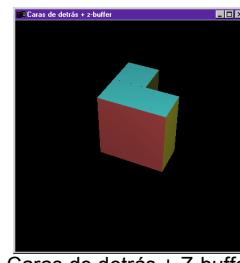


Eliminando caras de detrás

Ahora bien, si sabemos que todos nuestros objetos están **cerrados**, es decir, no se ve el interior, podemos **combinarlo** con el Z-buffer, ya que descartaremos ya para empezar las caras que no vamos a ver, ganando así en **velocidad** de renderización.



Z-buffer



Caras de detrás + Z-buffer

Ahora bien, la combinación de estos dos algoritmos **no** siempre es adecuada. En el caso de la tetera, por ejemplo, puesto que la tapa no está perfectamente ajustada a la tetera, y por tanto, queda un hueco (es decir, se ve el interior) ¡Si utilizamos el algoritmo de las caras de detrás veremos a través! Por tanto, en este caso lo mejor es utilizar sólo el algoritmo del Z-buffer.



Caras de detrás + Z-buffer



Sólo Z-buffer

## Luces y materiales

Para introducir un poquito más de realismo a nuestra escena, podemos substituir los modelos de alambres por modelos de caras planas. Ahora bien, si hacemos esto directamente, el resultado, para el caso de la tetera de **glutSolidTeapot**, es el siguiente:



La verdad es que mucho realismo no tiene. El problema está en que todas las caras se pintan con el mismo color, y por tanto lo que vemos es la "sombra" del objeto.

En el mundo real, en cambio, los objetos están más o menos iluminados dependiendo de su posición y orientación respecto a las distintas fuentes de luz que inciden sobre ellos, así como del tipo de material de que están hechos. Por tanto, si queremos mejorar el realismo, tendremos que modelizar este fenómeno.

OpenGL modeliza dos tipos de reflexión: difusa y especular. La **reflexión difusa** se produce en superficies mates, donde no hay una dirección privilegiada de reflexión. En cambio, la **reflexión especular** en el caso ideal consiste en que la luz se refleja toda en una sola dirección, formando un ángulo con la normal igual al que forma la luz incidente.

Estos tipos de reflexiones dependen tanto del tipo de luz como de la superficie. Así, una bombilla produce más reflejos especulares que un fluorescente, y una taza de porcelana es más especular que una pelota de goma. Por tanto, estas propiedades se aplican tanto a las luces como a los materiales.

Por otro lado, también se modeliza la **luz ambiente**, que es una manera sencilla de modelizar la luz que no incide directamente sobre el objeto, sino que va a parar a él después de reflejarse por toda la escena. De esta manera, se evitan sombras bruscas, que restarían realismo a nuestra renderización. Por homogeneidad, también se puede definir la luz ambiente que emite una fuente, aunque existe una manera más coherente de definirla mediante la función **glLightModelfv**.

Así, en el modo RGBA, que es con el que trabajaremos, el color con que se ve un vértice de un objeto es la suma de la intensidad de **emisión** del material, el producto de la reflectancia ambiente y de la iluminación ambiente, y la contribución de cada luz activa. Cada luz contribuye con una suma de tres términos: **ambiente**, **difusa** y **especular**.

$$I_{\text{vértice}} = I_{\text{emisión}} + R_{\text{ambiente}} \cdot S_{\text{ambiente}} + \sum (k_a \cdot I_a^i + k_d \cdot I_d^i + k_s \cdot I_s^i) \cdot f(\mathbf{d}_i)$$

En realidad, en OpenGL  $k_a = 1.0$ ,  $k_d = 1.0$  y  $k_s = 1.0$ .

La contribución de la luz **ambiente** de una fuente ( $I_a^i$ ) es el producto de la reflectancia ambiente del material y de la intensidad ambiente que emite esa fuente.

La contribución **difusa** de una fuente ( $I_d^i$ ) es el producto de la reflectancia difusa del material, de la intensidad de la luz difusa que emite y del producto escalar entre la normal y el vector de dirección que va de esa fuente al vértice.

La contribución de luz **especular** de una fuente ( $I_s^i$ ) es el producto de la reflectancia del material, de la intensidad de luz especular que emite esa fuente, y del producto escalar entre los vectores vértice-ojo y vértice-luz, elevado a la potencia del brillo (shininess) del material.

Las tres contribuciones de cada fuente luz se ven **atenuadas** por igual basándose en una función  $f(\mathbf{d}_i)$  dependiente de la **distancia**  $\mathbf{d}_i$  desde el vértice a la fuente de luz, el exponente de focalización de esa luz y el ángulo de corte de la fuente de luz.

# Estudiar sin publi es posible.

Compra Wuolah Coins y que nada te distraiga durante el estudio.



Todos los productos escalares son reemplazados por cero si dan como resultado un número negativo. Cuando se quiere trabajar con las caras de detrás, se invierten las normales.

## Métodos de sombreado

OpenGL nos permite elegir entre dos métodos de sombreado, de caras planas y suave. Para elegir el método de sombreado, usaremos la función `glShadeModel` con parámetro `GL_FLAT` para caras planas, y `GL_SMOOTH` para el suave.



## Especificación de las normales

OpenGL, excepto cuando renderiza NURBS y otras funciones gráficas de alto nivel, no genera automáticamente normales, así que tenemos que especificárselas nosotros mismos.

Para especificar la normal actual, basta con hacer una llamada a la función `glNormal3f(nx, ny, nz)`. A partir de ese momento, cualquier vértice que se renderice se supondrá que tiene como normal  $(nx, ny, nz)$ . Esta normal es deseable que esté normalizada.

Con el modo de sombreado suave activo, dependiendo de si definimos la misma normal para toda la cara o para cada vértice, podemos obtener un sombreado de caras planas o de tipo **gouraud**. Para obtener un sombreado de tipo **gouraud**, bastan con definir a cada vértice la normal media de las caras a las que pertenece. Esto es especialmente interesante cuando queremos renderizar superficies curvas, mientras que para renderizar poliedros es más aconsejable asignar una única normal para todos los vértices de una misma cara.

## Fuentes de luz

En OpenGL, todas las fuentes son **puntuales**, no se pueden definir, pues, fuentes distribuidas. OpenGL permite activar un número limitado de luces; como mínimo serán 8, pero este número dependerá de la implementación y lo podemos obtener haciendo una llamada como sigue:

```
int MaxLuces;  
glGetIntegerv(GL_MAX_LIGHTS, &MaxLuces);
```

Así, para activar una luz, podemos hacer una llamada a:

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

La primera llamada le dice a OpenGL que active la iluminación. A partir de este momento las llamadas a `glColor3f` no se tendrán en cuenta, si no sólo los materiales (ver más adelante). En este caso estamos activando la primera luz; para activar la siguiente haríamos la llamada `glEnable(GL_LIGHT1)`. De todas maneras, siempre se cumple que `GL_LIGHTi = GL_LIGHT0 + i`.

Ahora bien, cada luz puede tener distintas características, como la posición, el tipo de luz que emite, el color de la luz, ... etc. Para definir estas propiedades, se utilizan las funciones:

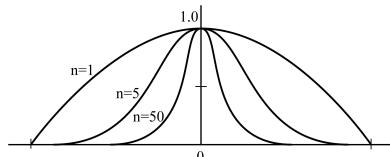
```
void glLightf(int light, int property, float param)  
void glLightfv(int light, int property, float* param)
```

La función `glLightf` se usa para las propiedades que requieren como parámetro un escalar, mientras que `glLightfv` se usa para las propiedades que esperan un vector.



## Propiedades escalares

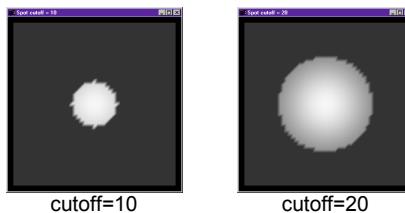
**GL\_SPOT\_EXPONENT**. Define la focalización de la luz. Así, la intensidad de la luz es proporcional al  $\cos^n \phi$ , donde  $\phi$  es el ángulo entre la dirección de la fuente de luz y la dirección de la posición de la luz hasta el vértice iluminado. Esta propiedad se refiere al exponente  $n$ . Por defecto,  $n = 0$ , que corresponde a una intensidad de la luz uniformemente distribuida ( $\cos^0 \phi = 1$ ), pero podemos darle valores en el rango [0, 128].



La función  $\cos^n \phi$  para diferentes valores de  $n$



**GL\_SPOT\_CUTOFF**. Esta propiedad especifica el ángulo de apertura de la luz. Se le pueden dar valores en el rango [0, 90], así como el valor especial 180. Si el ángulo entre la dirección de la luz y la dirección de la posición de la luz hasta el vértice es mayor que este ángulo, la luz se ve completamente enmascarada. Si es menor, la luz se ve controlada por el **spot exponent** y el **attenuation factor**. Por defecto, vale 180, que corresponde a una distribución uniforme de la luz.



**GL\_CONSTANT\_ATTENUATION**, **GL\_LINEAR\_ATTENUATION**, **GL\_QUADRATIC\_ATTENUATION**. El parámetro **param** especifica uno de los tres factores de atenuación, que siempre tiene que ser mayor que 0. Si la luz es posicional (y no direccional), la intensidad de ésta se ve atenuada por la distancia **d** según la siguiente fórmula:

$$I = I_{\text{fuente}} / (A_{\text{constant}} + A_{\text{linear}} \cdot d + A_{\text{quadratic}} \cdot d^2)$$

Por defecto, los valores de atenuación son  $(A_{\text{constant}}, A_{\text{linear}}, A_{\text{quadratic}}) = (1, 0, 0)$ , resultando en una **no** atenuación.

## Propiedades que requieren de un vector

**GL\_AMBIENT**. El parámetro corresponde a un vector de 4 flotantes (RGBA) que especifica el color de la luz ambiente que emite la fuente. Por defecto, vale  $(0, 0, 0, 1)$ , es decir, no emite luz ambiente.

**GL\_DIFFUSE**. El parámetro es el RGBA que especifica el color de la luz difusa que emite la fuente. Por defecto, vale  $(0, 0, 0, 1)$ , excepto cuando la fuente es **GL\_LIGHT0**, que vale  $(1, 1, 1, 1)$ .

**GL\_SPECULAR**. El parámetro es el RGBA de la luz specular que emite la fuente. Por defecto, vale  $(0, 0, 0, 1)$ , excepto cuando la fuente es **GL\_LIGHT0**, que vale  $(1, 1, 1, 1)$ .

**GL\_POSITION**. El parámetro corresponde a un vector de 4 flotantes ( $x, y, z, w$ ) que especifican la posición de la luz. La posición es transformada por la matriz **Modelview**, como si se tratase de un punto cualquiera y almacenada en coordenadas de ojo. Si la componente **w** de la posición es 0.0, la luz se trata como una luz direccional (un ejemplo sería la luz del sol). En ese caso, los cálculos de iluminación toman su dirección pero no su posición, y la atenuación no está habilitada. Si la luz es posicional, es decir, si **w** = 1.0 en el caso normal, se aplican los cálculos basados en la posición y dirección de la luz en coordenadas del ojo, y la atenuación está habilitada. Por defecto, esta propiedad vale  $(0, 0, 1, 0)$ , es decir, la luz es direccional, paralela y en dirección al eje  $-z$ .

**GL\_SPOT\_DIRECTION**. El parámetro corresponde a un vector de 3 flotantes ( $x, y, z$ ) que especifica la dirección de la luz. Esto sólo tiene sentido cuando **GL\_SPOT\_CUTOFF** no es 180, que es su valor por defecto. Por defecto, la dirección es  $(0, 0, -1)$ .

## Materiales

Para definir las propiedades de material de un objeto, usaremos las funciones

```
glMaterialf(int face, int property, float param)
glMaterialfv(int face, int property, float* param)
```

Como con **glLightf** y **glLightfv**, usaremos **glMaterialf** para las propiedades que requieren un parámetro escalar, y **glMaterialfv** para las propiedades que requieren un vector. El parámetro **face** se refiere a la cara, y puede ser **GL\_FRONT** o **GL\_BACK** (cara delantera o trasera). Por defecto, OpenGL sólo se ocupa de las caras delanteras, aunque con la función **glLightModelf** se puede obligar a OpenGL a que se ocupe también de las traseras.

### Propiedades escalares

**GL\_SHININESS**. OpenGL implementa la reflexión especular según una versión simplificada del modelo de Phong. El modelo de Phong utiliza la fórmula:

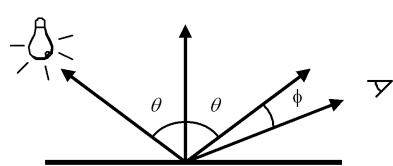
$$I_{\text{especular}} = W(\theta) \cdot \cos^n(\phi)$$

donde  $\theta$  corresponde al ángulo de incidencia, y  $\phi$  al ángulo de visión, es decir, el ángulo mínimo respecto a la reflexión especular para verla (si la reflexión es perfecta,  $\phi$  vale 0) (ver figura).

En cambio, OpenGL utiliza la fórmula siguiente:

$$I_{\text{especular}} = \cos^n(\phi)$$

Es decir, supone que  $W(\theta) = 1$ . La propiedad **GL\_SHININESS** se refiere al **exponente n**, y puede tomar valores en el rango [0, 128]. El valor por defecto es 0.



Ángulo de incidencia  $\theta$  y ángulo de visión  $\phi$



### Propiedades que requieren de un vector

**GL\_AMBIENT**. El parámetro es un vector de 4 flotantes RGBA correspondiente a la intensidad de la luz ambiente que refleja el objeto. Por defecto es (0.2, 0.2, 0.2, 1.0) para ambas caras.

**GL\_DIFFUSE**. El parámetro es un vector de 4 flotantes RGBA correspondiente a la intensidad de la luz difusa que refleja el objeto. Por defecto es (0.8, 0.8, 0.8, 1.0) para ambas caras.

**GL\_SPECULAR**. El parámetro es un vector de 4 flotantes RGBA correspondiente a la intensidad de la luz especular que refleja el objeto. Por defecto es (0.0, 0.0, 0.0, 1.0) para ambas caras.

**GL\_EMISSION** El parámetro es un vector de 4 flotantes RGBA correspondiente a la intensidad de la luz que **emite** el objeto. En efecto, OpenGL permite emular objetos que emiten luz. Ahora bien, para que el efecto de emisión de luz sea completo, se debe colocar una fuente de luz en el objeto. Por defecto es (0.0, 0.0, 0.0, 1.0) para ambas caras.

**GL\_AMBIENT\_AND\_DIFFUSE**. Equivale a hacer una llamada con **GL\_AMBIENT** y luego otra con **GL\_DIFFUSE** con el mismo parámetro.

## Estructura de un programa usando la GLUT

Las aplicaciones que desarrollaremos usando la GLUT se basan en una interficie de ventanas. Este tipo de interfeicies son muy cómodas para el usuario, pero la programación se complica bastante porque no podemos controlar lo que va a hacer el usuario.

Una manera sencilla de programar este tipo de aplicaciones es utilizando un esquema **orientado a eventos**.

Ejemplos de **eventos** pueden ser: que el usuario haga click con el mouse sobre la ventana, que cambie el tamaño de esta, que otra ventana tape a la nuestra y haya que redibujarla... etc.

La ejecución de un programa basado en eventos, pues, se convierte en el siguiente bucle:

```
hacer
    evento = CapturarEvento
    si evento = click entonces llamar a funcion Click
    sino si evento = cambiar_tamaño_ventana entonces llamar a funcion TamañoVentana
    ...
    sino si evento = redibujar entonces llamar a funcion Redibujar
    sino si evento = cerrar_ventana entonces llamar a funcion CerrarVentana
    mientras (ventana abierta)
```

Como vemos, mientras la ventana permanezca abierta, se pregunta cada vez qué evento ha ocurrido. En función del evento, se llamará a una función u otra. Estas funciones reciben el nombre de **callback**.

Por defecto, la GLUT ya tiene definidas callbacks para cada evento posible, pero, evidentemente, nosotros querremos personalizar este comportamiento. Para hacer esto, algunas de las funciones de las que disponemos son:

void glutDisplayFunc( void (*func) (void))	La función <b>func()</b> se llamará cada vez que haya que redibujar la ventana.
void glutReshapeFunc(void (*func) (int width, int height))	La función <b>func(width, height)</b> se llamará cada vez que la ventana cambie de tamaño, y recibirá como parámetros la nueva anchura y altura.

Para más información acerca de estas funciones y de otras, consultad el manual de GLUT que podéis encontrar en la página web de la asignatura.

### Ejemplo de control de eventos

El siguiente programa dibuja una tetera de alambres en 3D.

```
#include <gl/glut.h>
#include <stdio.h>
#include <stdlib.h>

void display(void) {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glutWireTeapot(0.5);
    glFlush();
    glutSwapBuffers();
}
void main(void) {
    glutInitDisplayMode(GLUT_DEPTH | GLUT_DOUBLE | GLUT_RGB);
    glutInitWindowSize(400, 400);
    glutInitWindowPosition(100, 100);
    glutCreateWindow("Ejemplo cutre de GLUT");

    glClearColor(0.0, 0.0, 0.0, 1.0);
    glEnable(GL_DEPTH_TEST); /* No hace falta, pero bueno... */

    glutDisplayFunc(display);
    glutMainLoop();
}
```

# Estudiar sin publi es posible.

Compra Wuolah Coins y que nada te distraiga durante el estudio.



Analicemos punto por punto lo que hace el programa:

- 1) Inicializar el modo de visualización. Esto se hace llamando a la función: **glutInitDisplayMode**. En este caso, estamos activando el buffer de profundidad (GLUT\_DEPTH), el doble buffering (GLUT\_DOUBLE) y la renderización en RGB (GLUT\_RGB).
- 2) Inicializar el tamaño de la ventana. Esto se hace mediante la función: **glutInitWindowSize (anchura, altura)**.
- 3) Inicializar la posición de la ventana en el escritorio. La función correspondiente es: **glutInitWindowPosition(x, y)**
- 4) Crear la ventana. Llamaremos a la función: **glutCreateWindow(char\* titulo)** Esta función muestra una ventana del tamaño y posición definidas y con el título que le hayamos puesto.
- 5) Definir el color de fondo (es decir, el color con que pintar la ventana para borrarla). El formato de la función para hacer esto es **glClearColor(R, G, B, A)**
- 6) Activar el test del buffer de profundidad: **glEnable(GL\_DEPTH\_TEST)**
- 7) Definir el callback para redibujar la ventana. Esto lo hacemos mediante la función **glutDisplayFunc(display)**. Así, la función que se llamará cada vez que haya que redibujar la ventana será la función **display()**:

```
void display(void) {  
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);  
    glutWireTeapot(0.5);  
    glFlush();  
    glutSwapBuffers();  
}
```

Esta función borra primero el buffer de color (es decir, la imagen) y el buffer de profundidad. Luego llama a **glutWireTeapot**, que hace que se dibuje una tetera de alambres. La llamada a **glFlush** vacía el pipeline de renderización y el **glutSwapBuffers** "pega" la imagen en la ventana.

¡Fíjate que le hemos pasado una función como parámetro! Sí, esto se puede hacer en C...

- 8) Finalmente, tenemos que hacer que el bucle de eventos se ponga en marcha. Para ello, llamaremos a la función **glutMainLoop()**.

## Control del mouse

De entre los **callbacks** que nos da la GLUT, son especialmente interesantes para el control del mouse los siguientes:

void <b>glutMouseFunc</b> ( void (*func) (int button, int state, int x, int y))	La función <b>func(button, state, x, y)</b> se llamará tanto cuando se apriete ( <b>state = GLUT_DOWN</b> ) como cuando se suelte ( <b>state = GLUT_UP</b> ) el botón definido por el parámetro <b>button</b> en la posición <b>(x, y)</b> .
void <b>glutMotionFunc</b> (void (*func) (int x, int y))	La función <b>func(x, y)</b> se llamará cuando el mouse se mueva mientras está pulsado uno de sus botones.



### Ejemplo de control de mouse

El siguiente ejemplo muestra una tetera que podremos rotar utilizando el mouse. Mientras que el mouse esté pulsado, según como lo movamos, moveremos la tetera. Prestad especial atención a las funciones onMouse y onMotion, especialmente a la obtención de los ángulos de rotación. Si la cámara estuviera situada en otro sitio, la obtención de los ángulos tendría que variar.

```
#include <gl/glut.h>

float alpha, beta;
int x0, y0;

void display(void) {
    glClear(GL_COLOR_BUFFER_BIT);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(20.0f, 1.0f, 1.0f, 10.0f);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(0.0f, 0.0f, 5.0f,
              0.0f, 0.0f, 0.0f,
              0.0f, 1.0f, 0.0f);
    glRotatef(alpha, 1.0f, 0.0f, 0.0f);
    glRotatef(beta, 0.0f, 1.0f, 0.0f);
    glutWireTeapot(0.5);
    glFlush();
    glutSwapBuffers();
}

void onMouse(int button, int state, int x, int y) {
    if ( (button == GLUT_LEFT_BUTTON) & (state == GLUT_DOWN) ) {
        x0 = x; y0 = y;
    }
}

void onMotion(int x, int y) {
    alpha = (alpha + (y - y0));
    beta = (beta + (x - x0));
    x0 = x; y0 = y;
    glutPostRedisplay();
}

void main(void) {
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(400, 400);
    glutInitWindowPosition(100, 100);
    glutCreateWindow("Ejemplo de menus");
    glutDisplayFunc(display);
    glutMouseFunc(onMouse);
    glutMotionFunc(onMotion);
    glutMainLoop();
}
```



## Definición de menús

La GLUT provee una manera sencilla de definir menús para nuestra aplicación. Veamos algunas de las herramientas de que disponemos para trabajar con menús:

<code>int glutCreateMenu(void (*func) (int value))</code>	Esta función crea un menú (todavía sin opciones), y le asigna la función <b>func(value)</b> . Esta función se llamará cada vez que una de las opciones del menú sea seleccionada por el usuario, y recibirá en <b>value</b> el código identificativo de la opción seleccionada. De esta manera, podremos <i>definir que hace cada opción de menú</i> . Devuelve un identificador de menú, que nos servirá cuando tengamos que referirnos a él.
<code>void glutSetMenu(int menu)</code>	Esta función hace que el menú identificado como <b>menu</b> sea el menú actual. Por defecto, el menú actual es el último que se ha creado.
<code>void glutAddMenuEntry(char *name, int value)</code>	Añade una opción de menú al menú actual. Esta opción se llamará <b>name</b> y se identificará por el número <b>value</b> .
<code>void glutAddSubMenu(char *name, int menu)</code>	Añade una opción de menú que abrirá un submenú (en lugar de ejecutar directamente un comando, como en el caso anterior). La opción se llamará <b>name</b> y el menú que aparecerá será el identificado como <b>menu</b> (ver función <b>glutCreateMenu</b> ).
<code>void glutChangeToMenuItem(int entry, char *name, int value)</code>	Esta función sirve para modificar una opción del menú actual. El parámetro <b>entry</b> nos indica la opción a modificar (por ejemplo, para modificar la primera, <b>entry = 1</b> ), <b>name</b> será el nuevo nombre de la opción y <b>value</b> el nuevo identificador que se le pasará a la función controladora de menú.
<code>void glutAttachMenu(int button)</code>	Esta función hace que el menú actual aparezca cada vez que se pulse el botón del mouse indicado por <b>button</b> ( <code>GLUT_LEFT_BUTTON</code> , <code>GLUT_MIDDLE_BUTTON</code> , <code>GLUT_RIGHT_BUTTON</code> ).

Para más información acerca de estas funciones y de otras, consultad el manual de GLUT que podéis encontrar en la página web de la asignatura.

## Ejemplo de creación de menús

Este programa tendrá un menú con dos submenús que permitirán elegir el color de fondo y el de dibujo. Para activar el menú principal, bastará con hacer click con el botón de la derecha. Fijáos que para redibujar la ventana se llama a **glutPostRedisplay**. Esta función envía un evento de redibujado.

```
#include <gl/glut.h>
int iFondo = 0;
int iDibujo = 3;
typedef enum {FONDO1,FONDO2,FONDO3,FONDO4,DIBUJO1,DIBUJO2,DIBUJO3,DIBUJO4}
opcionesMenu;

void onMenu(int opcion) {
    switch(opcion) {
        case FONDO1:
            iFondo = 0;
            break;
        case FONDO2:
            iFondo = 1;
            break;
        case FONDO3:
            iFondo = 2;
            break;
        case DIBUJO1:
            iDibujo = 3;
            break;
        case DIBUJO2:
            iDibujo = 4;
            break;
        case DIBUJO3:
            iDibujo = 5;
            break;
    }
    glutPostRedisplay();
}
```

```

    }
    glutPostRedisplay();
}

void creacionMenu(void) {
    int menuFondo, menuDibujo, menuPrincipal;

    menuFondo = glutCreateMenu(onMenu);
    glutAddMenuEntry("Negro", FONDO1);
    glutAddMenuEntry("Verde oscuro", FONDO2);
    glutAddMenuEntry("Azul oscuro", FONDO3);

    menuDibujo = glutCreateMenu(onMenu);
    glutAddMenuEntry("Blanco", DIBUJO1);
    glutAddMenuEntry("Verde claro", DIBUJO2);
    glutAddMenuEntry("Azul claro", DIBUJO3);

    menuPrincipal = glutCreateMenu(onMenu);
    glutAddSubMenu("Color de fondo", menuFondo);
    glutAddSubMenu("Color de dibujo", menuDibujo);

    glutAttachMenu(GLUT_RIGHT_BUTTON);
}

void display(void) {
    float colores[6][3] = {
        { 0.00f, 0.00f, 0.00f}, // 0 - negro
        { 0.06f, 0.25f, 0.13f}, // 1 - verde oscuro
        { 0.10f, 0.07f, 0.33f}, // 2 - azul oscuro
        { 1.00f, 1.00f, 1.00f}, // 3 - blanco
        { 0.12f, 0.50f, 0.26f}, // 4 - verde claro
        { 0.20f, 0.14f, 0.66f}, // 5 - azul claro
    };
    glClearColor(colores[iFondo][0], colores[iFondo][1], colores[iFondo][2], 1.0f);
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(colores[iDibujo][0], colores[iDibujo][1], colores[iDibujo][2]);
    glutWireTeapot(0.5);

    glFlush();
    glutSwapBuffers();
}

void main(void) {
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(400, 400);
    glutInitWindowPosition(100, 100);

    glutCreateWindow("Ejemplo de menus");
    glutDisplayFunc(display);
    creacionMenu();

    glutMainLoop();
}

```



# Estudiar sin publi es posible.

Compra Wuolah Coins y que nada te distraiga durante el estudio.



## Configuración del Visual C++ para usar la GLUT

En el laboratorio de prácticas ya estará configurado el Visual para poder compilar programas basados en la GLUT. Pero para trabajar en casa deberéis instalarlo vosotros mismos. Podéis conseguir los ficheros necesarios en la dirección <http://www.cvc.uab.es/shared/teach/a24983/c24983.htm>.

Ficheros necesarios:

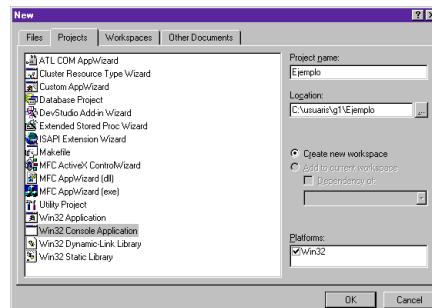
- glut32.dll, que hay que grabarlo en el directorio **C:\WINDOWS\SYSTEM** del vuestro ordenador.
- glut32.lib, que hay que guardarla en el directorio **[PATH del Visual C]\lib**.
- glut.h, que hay que guardarla en **[PATH del visual C]\include\gl**.

**Nota:** típicamente, el path del Visual C es **C:\Archivos de programa\Microsoft Visual Studio\VC98**.

## Creación del ejecutable con Visual C++ 6.0

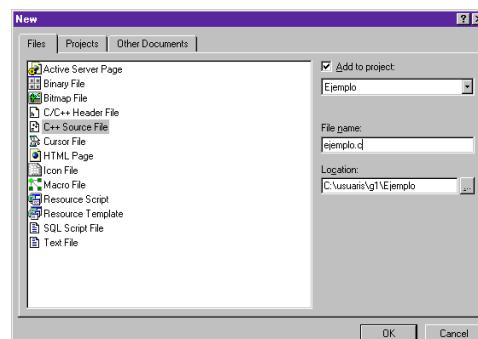
Para compilar un programa como los de los ejemplos en Visual C++ 6.0 (previamente configurado), hacemos lo siguiente:

- 1) Abrimos el Visual C++
- 2) Vamos a la opción de menú “File | New” y seleccionamos “Win32 Console Application” en la pestaña “Projects”. Introducimos el nombre y el path donde queremos que se nos cree el proyecto. Pulsamos **OK**.



Seleccionamos la opción “An empty project” y pulsamos **Finish** y luego **OK**. Esto crea un proyecto vacío.

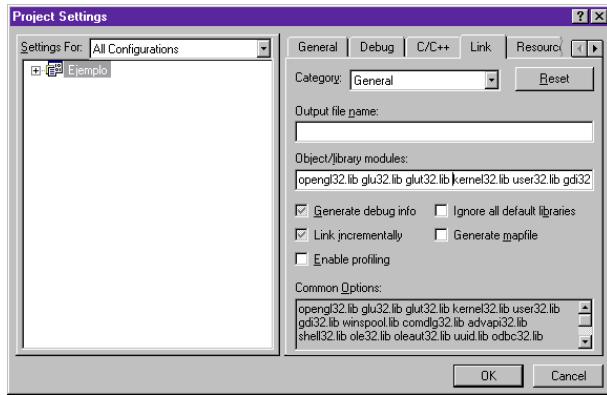
- 3) Seleccionamos la opción de menú “Project | Add to project... | New...”. Seleccionamos en la pestaña de “Files” la opción “C++ Source File”, y le damos como nombre “ejemplo.c” y pulsamos sobre **OK**.



- 4) Escribimos el código del programa.



- 5) Seleccionamos la opción de menú “**Project | Settings**” . Vamos a la pestaña de “**Link**”. Seleccionamos “**All Configurations**” en el cuadro desplegable de “**Settings for**”. Añadimos al cuadro de edición “**Object / library modules**” las siguientes librerías: **opengl32.lib glu32.lib glut.lib** y pulsamos **OK**. Esto le dice al Visual que enlace las librerías de OpenGL, porque si no nos daría errores de link.



- 6) Ahora ya podemos obtener y ejecutar nuestro programa. Para ello, pulsamos sobre el icono: y a la pregunta respondemos “**Sí**”.

Si todo ha ido bien, deberíamos obtener una ventana del estilo siguiente:



**Nota:** por cuestiones de compatibilidad, se abren dos ventanas: una con los gráficos y otra de MS-DOS. Para poder volver a compilar tendremos que **cerrar las dos ventanas**.