# Curso de introducción a OpenGL (v1.1)

*Por Jorge García -aka Bardok (2004)-*

bardok@telefonica.net - shadow@bardok.net

http://www.bardok.net

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-sa/2.0/ or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA

# Índice de contenido


Capítulo 1: Introducción....7
1.1 Propósito de este manual....7
1.2 ¿A quién está dirigido?....7
1.3 Acerca del manual....8
Capítulo 2: OpenGL y GNU/Linux....9
2.1 La librería OpenGL....9
2.2 GNU/Linux y OpenGL....10
2.2.1 ¿Qué es necesario para desarrollar aplicaciones OpenGL?....11
Capítulo 3: Conceptos básicos sobre OpenGL....13
3.1 OpenGL como máquina de estados....13
3.2 El espacio 3D....14
3.2.1 Las transformaciones de los objetos....15
3.2.2 Las coordenadas homogéneas....17
3.3 Sobre GLUT....17
3.4 Las funciones básicas de OpenGL....18
3.4.1 Activación/desactivación de opciones....18
3.4.2 Las matrices y OpenGL....18
3.4.3 El dibujado en OpenGL....19
3.4.4 El color en OpenGL....20
3.4.5 La orientación de las caras en OpenGL....20
3.4.6 Las transformaciones en OpenGL....20
3.4.7 La proyección en OpenGL....20
3.5 Primer contacto con OpenGL....21
3.5.1 Mi primer programa OpenGL....22
3.5.2 Visualizando en perspectiva....24
3.5.3 Ocultación de objetos (Z-Buffer)....26
3.5.4 Jerarquías....28
Capítulo 4: Animaciones....35
4.1 Los vectores para animar figuras....35
4.2 El doble buffering....39
Capítulo 5: Iluminación....43
5.1 El modelo de iluminación en OpenGL....43
5.1.1 Luces....44
5.1.2 Materiales....44
5.1.3 Normales....44
5.2 Utilizando iluminación en OpenGL....45
5.2.1 Luces....45
5.2.2 Materiales....47
5.2.3 Ejemplo: iluminación direccional sobre una superficie....48
5.2.4 Ejemplo: moviendo un foco alrededor de una esfera....50
5.2.5 Ejercicio propuesto....52
Capítulo 6: Texturas....55
6.1 Las coordenadas de textura....55
6.2 Aplicar las texturas....56
6.2.1 Repetición de texturas....58
Capítulo 7: Interacción básica con GLUT....63
7.1 Interacción con el teclado....63

7.2 Redimensionado de la ventana....64
Capítulo 8: Recursos de interés....67
Anexo A: Código de los programas de ejemplo....69
A.a myfirstopenglprogram.c....69
A.b quadortho.c....70
A.c quadpersp.c....71
A.d zbuffer-yes.c....72
A.e simpleguy.c....73
A.f jerarquia-cuadro.c....75
A.g jerarquia-cuadro-recursivo.c....76
A.h sphere-rebotes.cpp....77
A.i sphere-rebotes-multi.cpp....80
A.j normals-perp.c....81
A.k lit-sphere.c....82
A.l l-sphere-rebotes-multi.cpp....84
A.m triang-texture.c....86
A.n normals-perp-texture.c....88
A.o texture-no-clamp.c + texture-yes-clamp.c....89
A.p lit-sphere-keyb.c....91

# Índice de ilustraciones

Figura 2.1. Acceso al hardware a través de OpenGL (GNU/Linux)................11
Figura 3.1. Ejes de coordenadas en OpenGL..................................................14
Figura 3.2. Traslación de un objeto...............................................................15
Figura 3.3. Rotación de un objeto..................................................................16
Figura 3.4. Escalado en el eje X.....................................................................16
Figura 3.5. Vértices de un polígono...............................................................20
Figura 3.6. Volúmen de proyección ortográfico.............................................21
Figura 3.7. Volúmen de visualización en perspectiva....................................21
Figura 3.8. Salida del programa "myfirstopenglprogram".............................24
Figura 3.9. Salida del programa "quadortho"................................................25
Figura 3.10. Salida del programa “quadpersp”.............................................26
Figura 3.11. Dibujado sin Z-Buffer................................................................27
Figura 3.12. Dibujado con Z-Buffer (salida del programa zbuffer-yes)...........28
Figura 3.13. Jerarquía sencilla (salida del programa jerarquia-cuadro.c)......30
Figura 3.14. Jerarquía fractal (programa jerarquia-cuadro-recursivo.c)........30
Figura 3.15. Salida del programa "simpleguy"..............................................32
Figura 3.16. Rotación del brazo (simpleguy).................................................33
Figura 4.1. Representación de la posición y velocidad con vectores.............36
Figura 4.2. Comportamiento del programa "rebotes"....................................36
Figura 4.3. Posición de la cámara en el ejemplo "sphere-rebotes"................38
Figura 4.4. Salida del programa "sphere-rebotes".........................................40
Figura 4.5. Salida del programa "sphere-rebotes-multi"...............................41
Figura 5.1. Efecto de las normales en la iluminación....................................45
Figura 5.2. Tipos de luces..............................................................................46
Figura 5.3. Salida del programa "lit-sphere"..................................................52
Figura 5.4. Salida del programa "l-sphere-rebotes-multi".............................53
Figura 6.1. Coordenadas de textura..............................................................56
Figura 6.2. Mapeado de textura....................................................................56
Figura 6.3. Diferencias entre GL_REPLACE y GL_MODULATE.....................58
Figura 6.4. Texturas para suelo y pared.........................................................59
Figura 6.5. Salida del programa "texture-no-clamp"......................................60
Figura 6.6. Salida del programa "texture-yes-clamp".....................................61

# Índice de tablas

Tabla 3.1. Primitivas de dibujado....................................................................19
Tabla 4.1. Métodos de la clase TSphere...........................................................37

# Capítulo 1: Introducción

*"Cuando no sepas qué poner, mejor no pongas nada..." (Bardok)*

## 1.1 Propósito de este manual

Este manual intenta ser una guía para adentrarse en la programación 3D, con la librería OpenGL, y orientado a la programación bajo GNU/Linux, aunque los conceptos aquí explicados son mayormente válidos para cualquier otra plataforma.

El enfoque que se va a dar es muy práctico, basado casi íntegramente en la realización de ejercicios que permitan al lector obtener los conocimientos básicos para desenvolverse por su cuenta en el mundo de la programación 3D.

## 1.2 ¿A quién está dirigido?

Este manual está dirigido a cualquier persona que tenga interés en adentrarse en el mundo de la programación 3D con OpenGL, y muy especialmente a los asistentes del curso del e-ghost que se celebrará en Julio del año 2004 en la Universidad de Deusto, en la que está físicamente ubicada dicho grupo.

Las personas que intenten hacer algo de utilidad con este manual, deben tener unos conocimientos mínimos de programación en C/C++...

No son necesarios conocimientos de programación específicos bajo GNU/Linux, ya que todo lo necesario para programar bajo esta plataforma se recoge en este manual.

El manual no se recomienda a personas con conocimientos avanzados de OpenGL, ya que no va a contar nada revolucionario acerca de esta tecnología gráfica 3D, si bien, quién sabe, siempre se puede aprender algo nuevo...

## *1.3 Acerca del manual*

El contenido de este manual se divide en secciones que intentan abordar separadamente los distintos puntos por los que un usuario principiante deberá pasar al enfrentarse a la programación con OpenGL bajo GNU/Linux (a partir de este momento, simplemente OpenGL).

En el capítulo 1 comenzaremos con una introducción a la librería OpenGL.

En el capítulo 2 se explicarán las maneras en las que OpenGL aparece en la plataforma GNU/Linux, y cómo deberemos de instalar lo necesario para programar con él.

En el capítulo 3 se comenzará con la programación en OpenGL como tal, viendo los conceptos básicos acerca de transformaciones y visualización propios de esta librería.

En el capítulo 4 se hablará de cómo animar las escenas.

En el capítulo 5 se describen los fundamentos de la iluminación, y cómo se utiliza en OpenGL.

En el capítulo 6 se explica la utilización de texturas en OpenGL, de manera básica.

En el capítulo 7 se explica brevemente cómo interactuar con las ventanas mediante la librería glut.

En el capítulo 8 se recomiendan una serie de lecturas de interés.

# Capítulo 2: OpenGL y GNU/Linux

*“No permitas que los árboles te impidan ver al chalado que te persigue con una motosierra...” (Bardok)*

## *2.1 La librería OpenGL*

OpenGL es un estándar creado por Silicon Graphics en el año 1992 para el diseño de una librería 2D/3D portable [OPENGL]. Actualmente es una de las tecnologías más empleadas en el diseño de aplicaciones 3D, y su última versión, a la fecha de hoy (2004), es la 1.5, y está en proceso la especificación de la versión 2.0, que incluirá muchas mejoras..

Se divide en tres partes funcionales:

- La librería OpenGL, que proporciona todo lo necesario para acceder a las funciones de dibujado de OpenGL.
- La librería GLU (OpenGL Utility Library), una librería de utilidades que proporciona acceso rápido a algunas de las funciones más comunes de OpenGL, a través de la ejecución de comandos de más bajo nivel, pertenecientes a la librería OpenGL propiamente dicha [REDBOOK].
- GLX (OpenGL Extension to the X Window System) proporciona un acceso a OpenGL para poder interactuar con un sistema de ventanas X Window, y está incluido en la propia implementación de OpenGL (su equivalente en Windows es la librería WGL, externa a la implementación de OpenGL).

Además de estas tres librerías, la librería GLUT (OpenGL Utility Toolkit) proporciona una interfaz independiente de plataforma para crear aplicaciones de ventanas totalmente portables [GLUT].

## 2.2 GNU/Linux y OpenGL

OpenGL aparece en GNU/Linux, habitualmente bajo la librería Mesa 3D [MESA], una implementación libre de OpenGL. Esta implementación es, de por sí, una implementación totalmente software de la librería OpenGL (excepto para las tarjetas 3dfx Voodoo1, Voodoo2, Voodoo Rush, Voodoo Banshee, Voodoo3, que tienen acceso al hardware a través del driver Mesa/Glide que proporciona Mesa), por lo que sólo es aconsejable utilizarla en el caso de no disponer de ninguna otra implementación de OpenGL. Actualmente (2004) la versión estable de esta librería es la 6.0.1, que implementa el estándar OpenGL 1.5.

La utilización más habitual de esta librería, no obstante, se realiza a partir de la librería xlibmesa. Esta librería forma parte del sistema Xfree86, y proporciona acceso a aceleración gráfica por hardware, siempre que la tarjeta gráfica y los drivers instalados lo permitan, a través de DRI (Direct Rendering Infraestructure [DRI]). Este es el modo a través del que podemos acceder a aceleración por hardware para tarjetas de vídeo como 3dfx, Intel, Matrox, ATI... DRI está presente en las implementaciones del servidor Xfree86 a partir de la versión 4 del mismo. Más concretamente, la versión 4.3 de Xfree86 trae consigo la librería xlibmesa 4.0.4, que implementa el estándar OpenGL 1.3.

Finalmente, las tarjetas NVIDIA proporcionan su propia implementación de la librería OpenGL, independientemente de la librería Mesa, si bien es cierto que ciertas partes de Mesa (las librerías de utilidades GLU y GLUT) pueden utilizarse junto con la implementación de NVIDIA.

En resumen, podemos utilizar la librería OpenGL de diferentes maneras, según nuestra configuración de hardware:

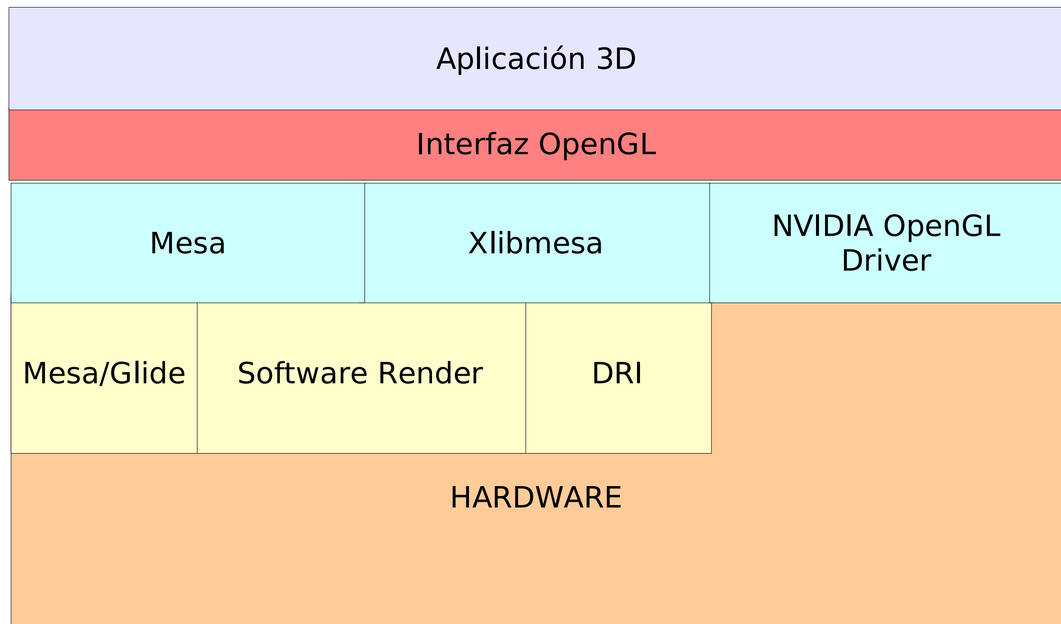


Figura 2.1. Acceso al hardware a través de OpenGL (GNU/Linux)

### 2.2.1 ¿Qué es necesario para desarrollar aplicaciones OpenGL?

A la hora de desarrollar aplicaciones OpenGL bajo GNU/Linux, necesitaremos las siguientes herramientas:

- Los binarios de la librería OpenGL, ya sea Mesa (compilado e instalado a partir del código fuente), xlibmesa (incluido con Xfree86 4 y posteriores) o drivers propios.
- Los paquetes de desarrollo de estas librerías (xlibmesa-dev, en el caso de xlibmesa, glutg3-dev... para drivers propios dependerá del fabricante, y para Mesa, si se ha instalado compilando el código fuente, no debería de haber problemas).
- Un compilador del lenguaje en el que vamos a programar, en este caso, en C, para lo que usaremos el compilador gcc, y g++ si utilizamos C++.
- Un editor de texto, ya sea en modo consola, o bajo el entorno X Windows (a mí me gusta el Kate de KDE, pero cada cual tiene sus preferencias).

Ahora ya estamos listos para comenzar a programar con OpenGL...

# Capítulo 3: Conceptos básicos sobre OpenGL

*“¿Por qué narices habré puesto que después de el nombre de un capítulo, me pida poner una cita de manera automática?” (Bardok)*

## 3.1 OpenGL como máquina de estados

OpenGL es una librería que trabaja como una máquina de estados. Esto es algo que oiremos hasta la saciedad al programar con OpenGL... ¿pero qué significa? La utilización de OpenGL consiste en activar y desactivar opciones, y realizar ciertas acciones, que tendrán como fruto una representación en pantalla (o no) de una serie de datos, dependiendo en el estado en que nos encontremos...

Así, no será lo mismo dibujar un triángulo y activar una textura, que activar una textura y dibujar un triángulo... en OpenGL, el orden de las acciones resulta crítico en la mayoría de las ocasiones... de igual manera, no será lo mismo trasladar y rotar algo, que rotarlo y trasladarlo, como veremos más adelante, en lo referente a las transformaciones...

De este modo, de manera muy abstracta, la manera de dibujar algo en OpenGL suele ser la siguiente:

1. Activar todas las opciones que van a ser persistentes a la escena (ponemos la cámara, activamos la iluminación global...)

2. Activar las opciones que establecen el estado de un objeto específico (su posición en el espacio, su textura...)

3. Dibujar el objeto.
4. Desactivar las opciones propias de ese objeto (volver a la posición original, desactivar su textura)
5. Volver al punto 2 hasta haber dibujado todos los objetos.

Esto, evidentemente, es un esquema sencillo... como se verá más adelante, estas operaciones pueden agruparse en jerarquías, lo que proporciona una gran potencia y flexibilidad a la hora de programar (¿se nota que soy fan de OpenGL? ;-)

## *3.2 El espacio 3D*

Si bien es cierto que OpenGL proporciona acceso a funciones de dibujado 2D, en este curso nos vamos a centrar en el espacio 3D... OpenGL trabaja, a grandes rasgos, en un espacio de tres dimensiones, aunque veremos que realmente, trabaja con coordenadas homogéneas (de cuatro dimensiones). Las tres dimensiones que nos interesan ahora son las especificadas por un sistema 3D ortonormal. Es decir, sus ejes son perpendiculares, y cada unidad en uno de ellos está representada por un vector de módulo 1 (si nos alejamos una unidad, nos alejamos la misma distancia del eje de coordenadas, da igual la dirección).

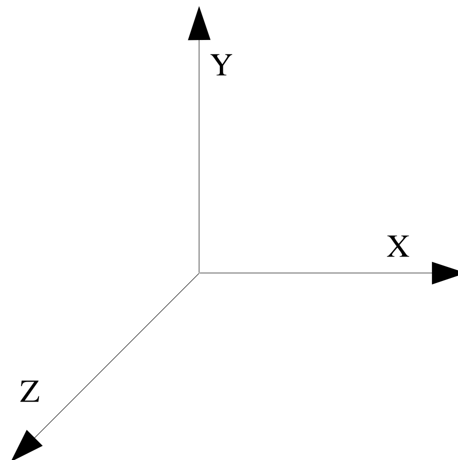


Figura 3.1. Ejes de coordenadas en OpenGL

La cuarta coordenada se utiliza entre otras razones, para representar la perspectiva, pero no nos meteremos con ello en este momento... de este modo, el sistema de coordenadas inicial de un sistema OpenGL puede representarse con esta matriz:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Finalmente, es recomendable desempolvar nuestros algebraicos básicos (vectores, normales, etc.), porque como veremos, nos van a resultar de gran utilidad a la hora de programar en 3D...

La situación de los ejes de coordenadas se refleja en la matriz de transformación. Esta matriz representa la transformación que se aplicará a todos los vértices que se dibujen mientras ella esté activa.

### 3.2.1 Las transformaciones de los objetos

Al oír hablar de programación 3D, habremos oído hablar de las transformaciones de objetos... estas transformaciones son las que van a describir cómo se visualiza un objeto en el espacio, y son de tres tipos:

– Traslación: una traslación es un desplazamiento de un objeto en el espacio... por ejemplo, movemos un objeto a una nueva posición desde la actual

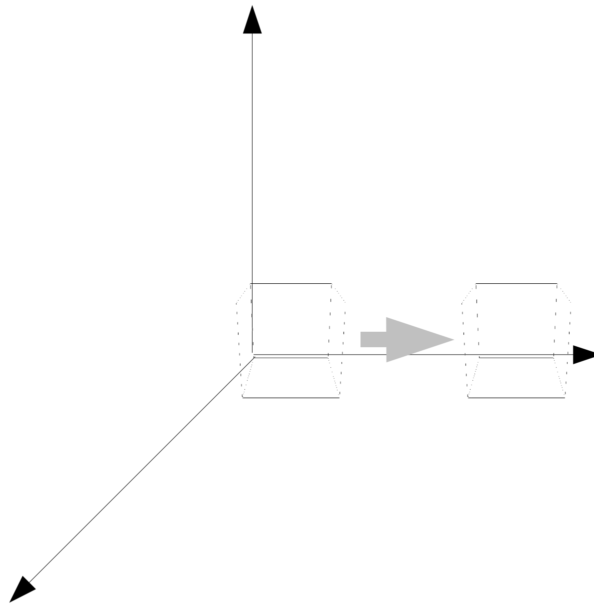

Figura 3.2. Traslación de un objeto

– Rotación: como su nombre indica, un objeto rota alrededor de un eje que pasa por su “centro de giro”

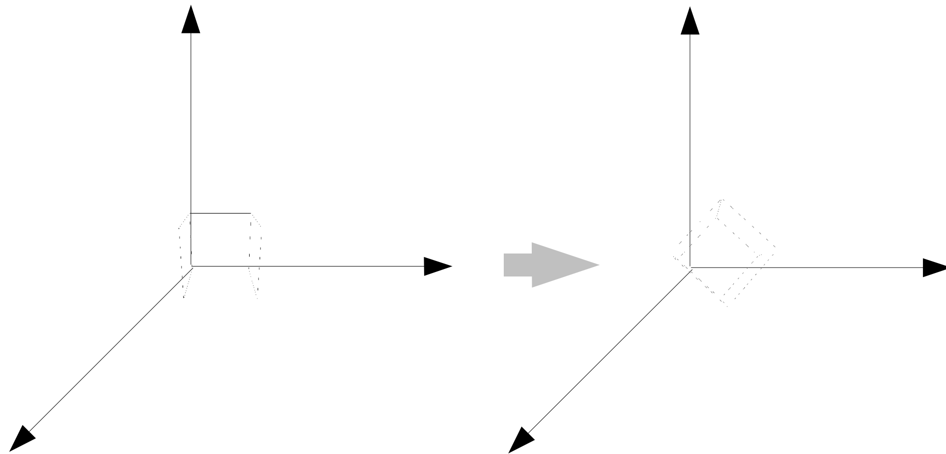

Figura 3.3. Rotación de un objeto

– Escalado: un objeto puede ver afectado el tamaño con que se visualiza por su transformación de escalado

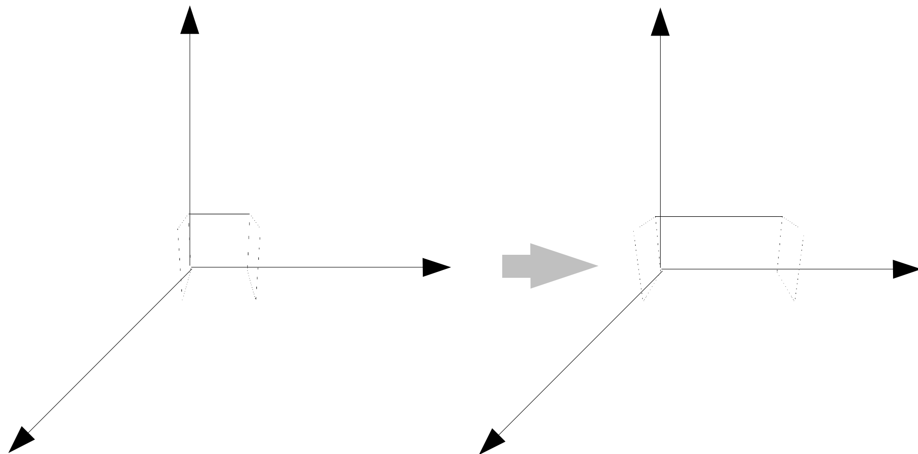

Figura 3.4. Escalado en el eje X

Sin entrar en demasiados detalles, es conveniente saber que toda trasformación construye una matriz de cuatro dimensiones que se multiplicará por la matriz de transformación inicial (aquella que define el eje de coordenadas).

Por ejemplo, al trasladar un objeto dos unidades en el eje X, se genera la siguiente matriz de transformación:

$$\begin{bmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Si aplicamos esta transformación a la matriz original, nos quedará que la nueva matriz de transformación es:

$$\begin{bmatrix}1 & 0 & 0 & 0\\0 & 1 & 0 & 0\\0 & 0 & 1 & 0\\0 & 0 & 0 & 1\end{bmatrix}\times\begin{bmatrix}1 & 0 & 0 & 2\\0 & 1 & 0 & 0\\0 & 0 & 1 & 0\\0 & 0 & 0 & 1\end{bmatrix}=\begin{bmatrix}1 & 0 & 0 & 2\\0 & 1 & 0 & 0\\0 & 0 & 1 & 0\\0 & 0 & 0 & 1\end{bmatrix}$$

Si ahora dibujamos el punto (1,0,0), teniendo en cuenta que la matriz indica un desplazamiento de dos unidades en eje X, el punto debería de dibujarse en la posición (3,0,0). Para esto, se multiplica el punto por la matriz de transformación:

$$\begin{bmatrix}1 & 0 & 0 & 2\\0 & 1 & 0 & 0\\0 & 0 & 1 & 0\\0 & 0 & 0 & 1\end{bmatrix}\times\begin{bmatrix}1\\0\\0\\1\end{bmatrix}=\begin{bmatrix}3\\0\\0\\1\end{bmatrix}$$

Para poder multiplicar el punto (1,0,0) por la matriz, lo hemos convertido a coordenadas homogéneas, y para ello, hemos añadido una última componente con el valor 1.

### 3.2.2 Las coordenadas homogéneas

Cuando pensamos en dibujado 3D, estamos acostumbrados a pensar en espacios de tres coordenadas, o dos, para figuras 2D, pero en OpenGL todo se traduce a coordenadas homogéneas. Las razones son varias (uniformidad de operación con matrices y facilidad de representación de diferentes conceptos relativos a la profundidad), pero no es objeto de este manual discutirlas.

De esta forma, el punto 3D (1,2,3) es traducido por OpenGL al punto (1,2,3,1.0) y el punto 2D (1,2) es traducido a (1,2,0.0,1.0). A grandes rasgos, podemos de decir que un punto (x,y,z,w) en coordenadas homogéneas, es equivalente al punto 3D (x/w,y/w,z/w). Así los puntos (1,2,0,10), (1,2,0,1) y (1,2,0,0.00001) se corresponden con los puntos 2D (0.1,0.2), (1,2) y (10000,20000).

Un apunte interesante es que si la coordenada w es igual a 0, estaríamos hablando, del punto situado en el infinito en la dirección del vector que especifican las otras tres coordenadas (vemos que, si se decrementa la w, el punto 2D o 3D equivalente se aleja). Esto resultará muy útil para entender ciertos aspectos de la iluminación.

## *3.3 Sobre GLUT*

Para los ejemplos de este manual vamos a utilizar la librería de ventanas GLUT, y para ello hemos de tener claros ciertos aspectos de esta librería.

La librería GLUT incluye funciones para la creación de ventanas independiente de plataforma. A medida que vayamos viendo distintos

ejemplos, iremos profundizando en las funciones utilizadas. Inicialmente, sólo vamos a comentar que GLUT funciona generalmente a través de funciones de callback, es decir, como parámetro de una función, se le pasa un puntero a otra función que será llamada por la función principal.

## *3.4 Las funciones básicas de OpenGL*

Para comenzar a utilizar OpenGL hemos de tener claras diversas funciones que vamos a utilizar muy a menudo.

### 3.4.1 Activación/desactivación de opciones

Como se ha comentado, OpenGL es una máquina de estados: activamos y desactivamos opciones que afectan al dibujado. Habitualmente, las opciones se activan y desactivan con glEnable(<OPTION>) y glDisable(<OPTION>).

Por ejemplo glEnable(GL_LIGHTING) activará la iluminación básica de la escena.

### 3.4.2 Las matrices y OpenGL

Anteriormente hemos visto que OpenGL guarda la transformación de los objetos en una matriz. A esta matriz se le denomina matriz de visualización/modelado, porque se emplea para estas dos funciones.

Además de esta transformación, OpenGL posee otra matriz muy importante, que es la matriz de proyección, en la que se guarda la información relativa a la “cámara” a través de la cual vamos a visualizar el mundo.

Al realizar operaciones que modifiquen alguna de estas dos matrices, tendremos que cambiar el “modo de matriz”, para que las operaciones afecten a la matriz que nos interesa.

Para ello utilizaremos las funciones glMatrixMode(GL_MODELVIEW) o glMatrixMode(GL_PROJECTION).

Además, existen dos funciones que permiten guardar y restaurar los valores de la matriz activa en una pila.

La función glPushMatrix() guarda una matriz en la cima de la pila, y glPopMatrix() la saca, y la restaura. Esto lo podemos utilizar para dibujar un objeto y, antes de dibujar el siguiente, restauramos la transformación inicial. Por ejemplo:

```
<transformación común para la escena>

glPushMatrix();

<trasformación propia del elemento 1>

<dibujado del elemento 1>
```

```
glPopMatrix(); // volvemos a la transformación común
glPushMatrix();
<transformación propia del elemento 2>
<dibujado del elemento 2>
glPopMatrix(); // volvemos a la transformación común
...
```

Finalmente, comentar la operación glLoadIdentity(), que carga la matriz identidad como matriz activa.

### 3.4.3 El dibujado en OpenGL

Para dibujar en OpenGL, tenemos que habilitar el modo de dibujado, establecer las opciones de dibujado de cada vértice, y dibujar cada uno de ellos. Al terminar de dibujar una figura, finalizamos el modo de dibujado.

Para comenzar a dibujar, utilizaremos el comando glBegin (<MODO_DE_DIBUJADO>), dónde el modo de dibujado vendrá dado por una constante:

| Parámetro | Descripción |
|---|---|
| GL_POINTS | Se dibujan vertices separados |
| GL_LINES | Cada par de vértices se interpreta como una línea |
| GL_POLYGON | Los vértices describen el contorno de un polígono |
| GL_TRIANGLES | Cada triplete de vértices de interpreta como un triángulo |
| GL_QUADS | Cada cuarteto de vértices se interpreta como un cuadrilátero |
| GL_LINE_STRIP | Líneas conectadas |
| GL_LINE_LOOP | Líneas conectadas, con unión entre el primer y último vértice |
| GL_TRIANGLE_STRIP | Se dibuja un triángulo, y cada nuevo vértice se interpreta con un triángulo entre los dos anteriores vértices y el nuevo |
| GL_TRIANGLE_FAN | Se dibujan triángulos con un vértice común |
| GL_QUAD_STRIP | Igual que el TRIANGLE_STRIP, con cuadriláteros |

Tabla 3.1. Primitivas de dibujado

Entre las funciones que permiten establecer los atributos de cada vértice, están aquellas que nos permiten seleccionar su color (glColor*), normal (glNormal*), coordenadas de textura (glTexCoord*), etc.

Finalmente, las funciones de dibujado de vértices tienen la forma “glVertex*”.

### 3.4.4 El color en OpenGL

OpenGL puede utilizar dos modos de color: color RGBA y color indexado. Nosotros vamos a centrarnos en el color RGBA. Este color recibe este nombre porque se compone de cuatro componentes: Rojo (Red), Verde (Green), Azul (Blue) y canal Alfa, o transparencia.

### 3.4.5 La orientación de las caras en OpenGL

Un polígono tiene dos caras, delantera y trasera. La manera de saber qué cara es la delantera, y cual la trasera, es que, si miramos la delantera, los vértices se habrán dibujado en orden antihorario.

Por ejemplo, tenemos la siguiente figura:

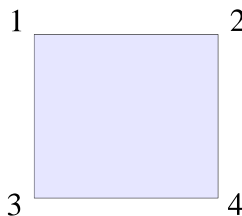


Figura 3.5. Vértices de un polígono

Si dibujamos los vertices en el orden 1,3,4,2, estaremos dibujando la cara delantera mirando hacia nosotros, pero si el orden es, por ejemplo, 1,2,4,3, estaremos mirando la cara trasera del polígono.

### 3.4.6 Las transformaciones en OpenGL

Finalmente, como operaciones básicas, vamos a comentar aquellas que nos van a permitir situar los objetos en el espacio, es decir, aplicarles transformaciones:

- glTranslate*: nos va a permitir trasladar un objeto en el espacio
- glRotate*: nos va a permitir rotar un objeto
- glScale*: nos va a permitir escalar un objeto
- glMultMatrix: multiplica la matriz de transformación actual por una matriz dada. Esto es algo muy utilizado en motores 3D

Es muy importante el orden en el que vayamos a realizar las transformaciones. No es lo mismo trasladar algo y después rotarlo, que primero rotarlo y luego trasladarlo, como podremos comprobar posteriormente...

### 3.4.7 La proyección en OpenGL

En el modo de proyección podemos especificar cómo va a afectar la posición de un objeto a su visualización. Tenemos dos maneras de visualizar el

espacio: con una proyección ortográfica, y con una proyección perspectiva:

- La proyección ortográfica:

  La proyección ortográfica nos permite visualizar todo aquello que se encuentre dentro de un cubo, delimitado por los parámetros de la función glOrto. A la hora de visualizar, la distancia al observador sólo se tiene en cuenta para determinar si el objeto está dentro o fuera del cubo...

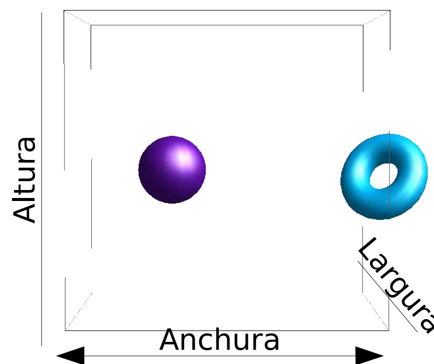


Figura 3.6. Volúmen de proyección ortográfico

- La proyección perspectiva:

  La proyección perspectiva delimita un volúmen de visualización dado por un ángulo de cámara, y una relación alto/ancho. La distancia al observador determinará el tamaño con el que un objeto se visualiza.

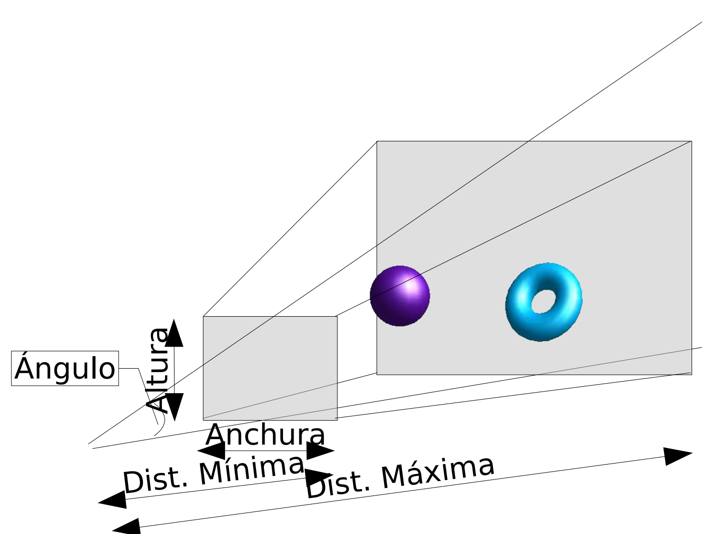


Figura 3.7. Volúmen de visualización en perspectiva

## 3.5 Primer contacto con OpenGL

Bueno, aquí está lo que todo programador desea tocar: el código ;-) vamos a comenzar con unos sencillos programas con los que podremos

comprobar cómo se utilizan los conceptos anteriormente explicados.

### 3.5.1 Mi primer programa OpenGL

En este programa vamos a dibujar un triángulo en pantalla con un vértice de cada color...

Vamos a hacer un único “include”, con la cabecera de GLUT. Generalmente, los archivos de cabecera de OpenGL estarán en el direcctorio “GL” dentro del árbol de archivos de cabecera:

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

Sólo incluímos este archivo, porque ya incluye las cabeceras “gl.h” y “glu.h”. Comenzamos con la función main:

```
int main(int argc, char * argv)
{
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGBA);
    glutInitWindowPosition(20,20);
    glutInitWindowSize(500,500);
    glutCreateWindow(argv[0]);

    glutDisplayFunc(display);
    glutMainLoop();

    return 0;
}
```

Lsa cuatro primeras llamadas crean la ventana con la librería GLUT:

1. Inicializamos los modos con los que se creará la ventana: con un sólo buffer de dibujado, y modelo de color RGBA (rojo, verde, azúl y canal alfa – transparencias).
2. Establecemos el tamaño y posición de la ventana
3. Creamos la ventana

Después establecemos que función indica qué hay que dibujar en la ventana. “display” es un puntero a una función que contiene el código de lo que vamos a dibujar, y llamamos a la función que comienza el dibujado.

Ahora sólo nos queda definir la función “display”:

```
void display(void)
{
    glClearColor(0.0,0.0,0.0,0.0);
    // Color de fondo: negro
    glClear(GL_COLOR_BUFFER_BIT);
    // Boramos la pantalla

    glMatrixMode(GL_PROJECTION);
    // Modo proyección
    glLoadIdentity();
    // Cargamos la matriz identidad

    glOrtho(-1.0,1.0,-1.0,1.0,-1.0,1.0);
    // Proyección ortográfica, dentro del cubo señalado

    glMatrixMode(GL_MODELVIEW);
    // Modo de modelado
```

```
    glBegin(GL_TRIANGLES);
    // Dibujamos un triángulo
      glColor3f(1.0,0.0,0.0);
      // Color del primer vértice: rojo
      glVertex3f(0.0,0.8,0.0);
      // Coordenadas del primer vértice
      glColor3f(0.0,1.0,0.0);
      // Color del segundo vértice: verde
      glVertex3f(-0.6,-0.2,0.0);
      // Coordenadas del  segundo vértice
      glColor3f(0.0,0.0,1.0);
      // Color del tercer vértice: azúl
      glVertex3f(0.6,-0.2,0.0);
      // Coordenadas del  tercer vértice
    glEnd();
    // Terminamos de dibujar

    glFlush();
    // Forzamos el dibujado

    sleep(10);
    // Esperamos 10 segundos
    exit(0);
    // Salimos del programa
}
```

- La función glClearColor especifica cuál va a ser el color con el que se va rellenar el buffer de pantalla cuando se borre
- Con glClear ordenamos a OpenGL que borre los buffers indicados como parámetro. Para borrar la pantalla (el buffer de color) lo indicamos con la constante GL_COLOR_BUFFER_BIT
- Establecemos el volúmen de visualización, en modo proyección y con glOrtho.
- Dibujamos el triángulo

Guardamos el código como "myfirtsopenglprogram.c", y lo compilamos con la siguiente línea de código en una consola:

```
user@machine:$gcc  myfirtsopenglprogram.c -lglut -lGL -lGLU -o
myfirtsopenglprogram
```

Ejecutamos con:

```
user@machine:$./myfirstopenglprogram
```

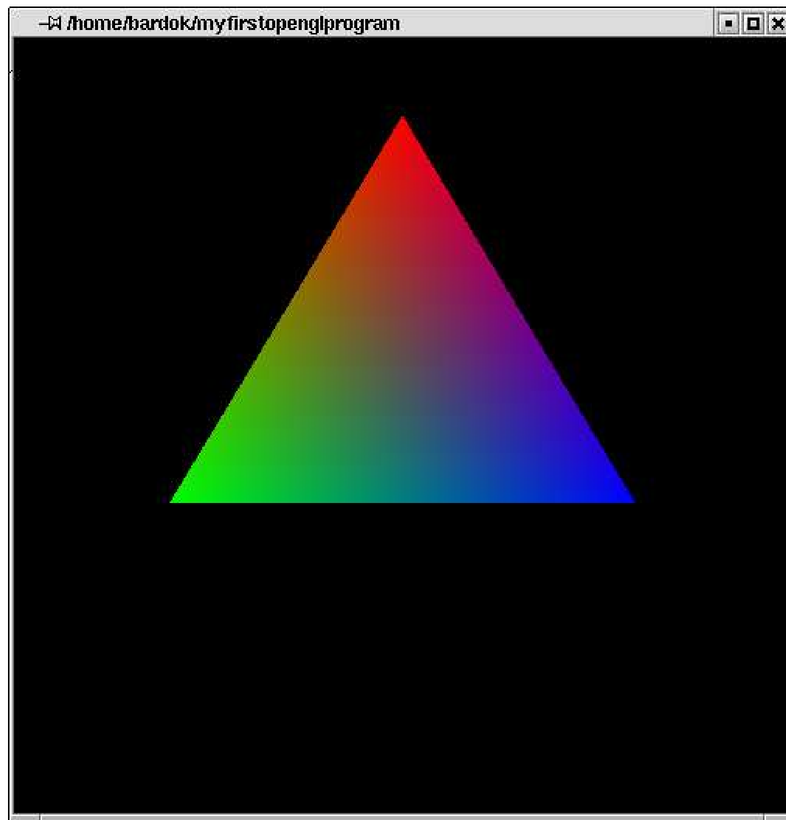

Figura 3.8. Salida del programa "myfirstopenglprogram"

## 3.5.2 Visualizando en perspectiva...

Vamos a dibujar un cuadrado con un programa muy similar al que tenemos en el ejemplo anterior. Para ello vamos a sustituir el código referente al dibujado por el siguiente código:

```
glBegin(GL_QUADS);
   // Dibujamos un cuadrado
   glColor3f(0.0,1.0,1.0);
   // Color para el cuadrado
   glVertex3f(-0.5,0.5,-0.5);
   // Coordenadas del primer vértice (superior-izquierda)
   glVertex3f(-0.5,-0.5,0.5);
   // Coordenadas del segundo vértice (inferior-izquierda)
   glVertex3f(0.5,-0.5,0.5);
   // Coordenadas del primer vértice (inferior-derecha)
   glVertex3f(0.5,0.5,-0.5);
   // Coordenadas del primer vértice (superior-derecha)
glEnd();
// Terminamos de dibujar
```

Guardamos como “quadortho.c”, compilamos el programa, y observamos el resultado:

```
user@machine:$gcc quadortho.c -lglut – lGL -lGLU -o quadortho
user@machine:$./quadortho
```

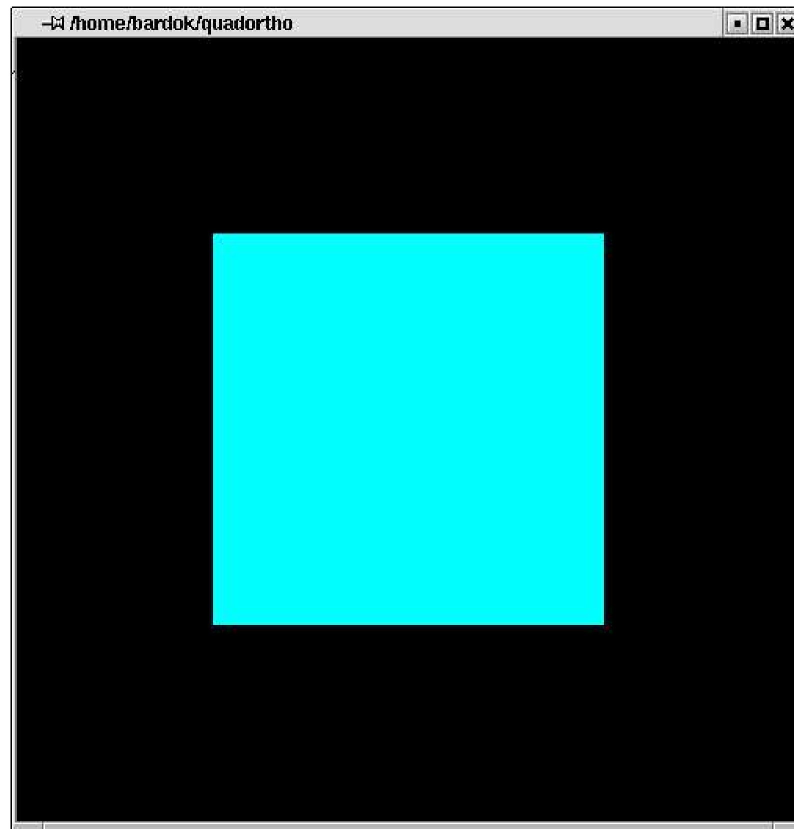


Figura 3.9. Salida del programa "quadortho"

Si estudiamos el código, observamos que la parte superior del cuadrado está más alejada del observador que la inferior, pero como se explicó en el apartado relativo a las proyecciones, esto no se tiene en cuenta para la proyección ortográfica. Si queremos que esto se refleje, tendremos que crear una proyección perspectiva...

Para ello vamos a sustituir la llamada a glOrtho por una llamada a gluPerspective, que establecerá el modo de visualización en perspectiva:

```
gluPerspective(60.0,1.0,1.0,100.0);
// Proyección perspectiva. El ángulo de visualización es de 60
grados, la razón ancho/alto es 1 (son inguales), la distancia
mínima es z=1.0, y la distancia máxima es z=100.0
```

Guardamos como “quadpersp.c”, compilamos y ejecutamos:

```
user@machine:$gcc quadpersp.c -lglut – lGL -lGLU -o  quadpersp
user@machine:$./ quadpersp
```

¡Y nos encontramos con que no se ve nada! Que no cunda el pánico... todo tiene sentido... tal y como está todo planteado, el cuadrado se va a dibujar con centro en el origen de coordenadas... ¿pero dónde está el observador? En la visualización perspectiva, la posición del observador determina cómo

se ven las cosas... vamos a tomar que el observador, ya que no hemos indicado lo contrario, se encuentra en la posición (0,0,0). Vamos a retrasar un poco el cuadrado, para alejarlo del observador, porque sino, se dibuja justo en el mimo lugar que éste, y no puede verse nada... para ello, después de entrar en el modo de modelado/visualización, añadimos:

```
glTranslatef(0.0,0.0,-2.0);
// Alejamos el cuadrado del observador dos unidades en el eje Z
```

Guardamos, compilamos, ejecutamos y...

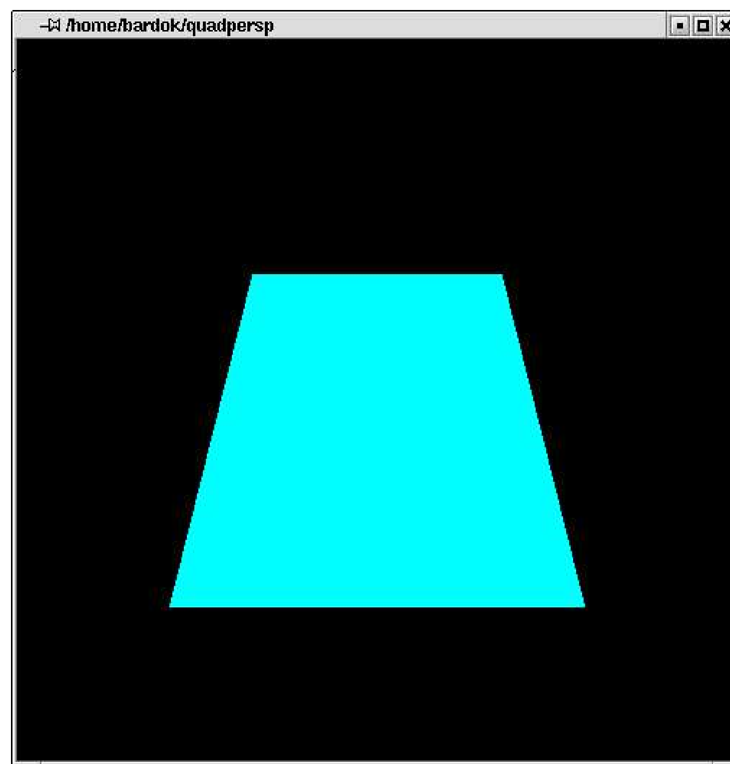


Figura 3.10. Salida del programa “quadpersp”

...¡perfecto! La parte superior del triángulo se dibuja alejada del observador :-D

### 3.5.3 Ocultación de objetos (Z-Buffer)

Si añadimos en el código de visualización un triángulo con las coordenadas (0.0,0.5,0.0), (-0.7,-0.5,0.0),(0.7,-0.5,0.0), después del codigo de dibujado del cuadrado, la salida es la siguiente:

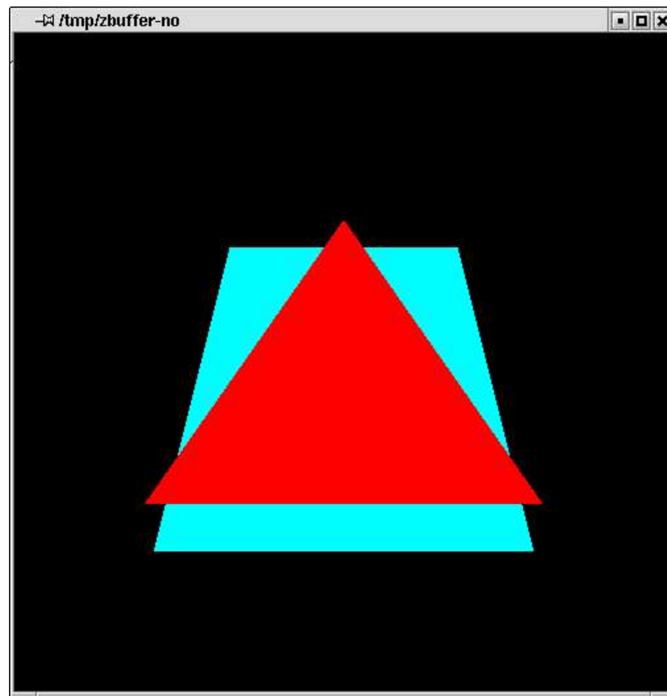


Figura 3.11. Dibujado sin Z-Buffer

Como podemos observar, no se tiene en cuenta que la parte inferior del triángulo debería de estar tapada por el cuadrado, porque su coordenada Z, que indica la profundidad, es mayor (-0.5 para el cuadrado y 0 para el triángulo). Para poder visualizar los objetos con oclusión, tenemos que activar el Z-Buffer.

El Z-Buffer es un array que guarda la profundidad (coordenada Z) para todos los píxels del área de dibujado, de manera que cuando un píxel va a ser dibujado, se mira si su coordenada Z es menor que la coordenada en el Z-Buffer para ese píxel en pantalla. Para activarlo, hemos de añadir las siguientes líneas de código, antes del dibujado:

```
glDepthFunc(GL_LEQUAL);
glEnable(GL_DEPTH_TEST);
glClearDepth(1.0);
```

Con la llamada a glEnable(GL_DEPTH_TEST) habilitamos la comprobación de la profundidad en el dibujado.

Con glDepthFunc(GL_LEQUAL) decimos que el nuevo píxel se dibuje si su coordenada Z es más cercana al punto de visualización, o igual, que la que hay actualmente en Z-Buffer.

Con glClearDepth(1.0) decimos que cada vez que quedamos borrar el buffer de profundidad, se inicialicen sus posiciones al valor 1.0.

Junto con esto, al borrar la pantalla, le indicamos al programa que inicialice el Z-Buffer con:

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Finalmente, sólo nos queda indicar al programa que, cuando se cree la ventana, se reserve espacio para el Z-Buffer. Esto lo indicamos al crear la ventana con:

```
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGBA | GLUT_DEPTH);
```

Tras estos cambios, la salida del programa es la siguiente:

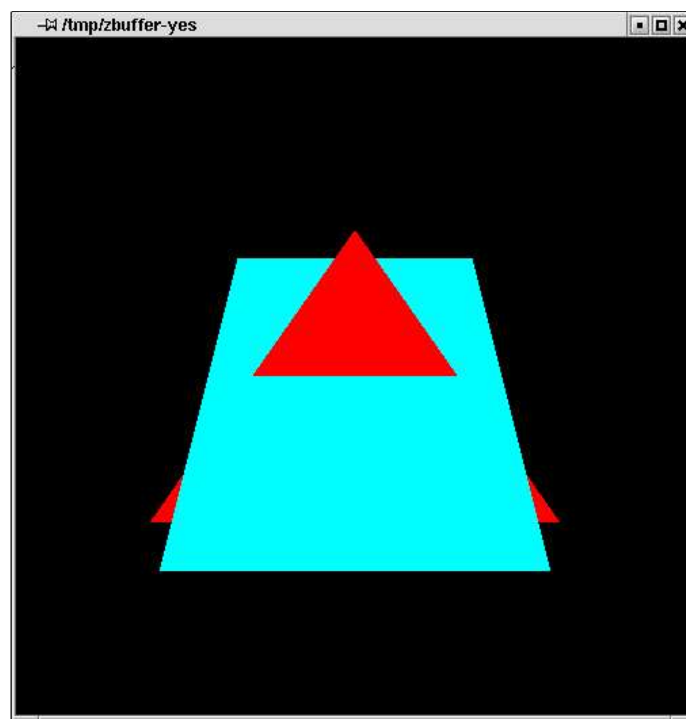


Figura 3.12. Dibujado con Z-Buffer (salida del programa zbuffer-yes)

## 3.5.4 Jerarquías

Ya hemos visto los conceptos básicos para dibujar figuras en OpenGL, ahora vamos a “jugar” un poco con transformaciones y matrices...

Genalmente, los objetos se agrupan en jerarquías. En ellas, existe algún objeto de cuya posición depende la posición del resto de objetos. El ejemplo más claro es el cuerpo humano: de la posición del tronco depende la posición de los brazos, y de la posición de los brazos, la de las manos.

Para representar jerarquías con OpenGL, utilizaremos las pilas de matrices. El procedimiento es el siguiente:

1. Se sitúa el elemento principal
2. Se dibuja el elemento principal
3. Se apila la matriz actual (glPushMatrix)
4. Se sitúa el primer elemento secundario con respecto al primero
5. Se dibuja el primer elemento secundario
6. Vuelta a la situación del elemento principal (glPopMatrix)
7. Se repite para los demás elementos secundarios

Para ver el funcionamiento de las jerarquías vamos a realizar un programa que dibuje un cuadrado (figura principal) y cuatro cuadrados secundarios, dibujados con respecto al principal.

Para ello, definimos, por comodidad, una función "dibujarCuadro" que recibe el tamaño del cuadrado, y lo dibuja centrado:

```
void dibujarCuadro(float tam)
{
    glBegin(GL_QUADS);
    glVertex3f(-tam/2.0,tam/2.0,0.0);
    glVertex3f(-tam/2.0,-tam/2.0,0.0);
    glVertex3f(tam/2.0,-tam/2.0,0.0);
    glVertex3f(tam/2.0,tam/2.0,0.0);
    glEnd();
}
```

Para dibujar la jerarquía, hacemos:

```
glColor4f(1.0,0.0,0.0,1.0);
dibujarCuadro(1.0);
glPushMatrix();
    glTranslatef(0.0, 2.0, 0.0);
    dibujarCuadro(0.5);
glPopMatrix();
glPushMatrix();
    glTranslatef(0.0, -2.0, 0.0);
    dibujarCuadro(0.5);
glPopMatrix();
...
```

Con esto, hemos dibujado una sencilla jerarquía.

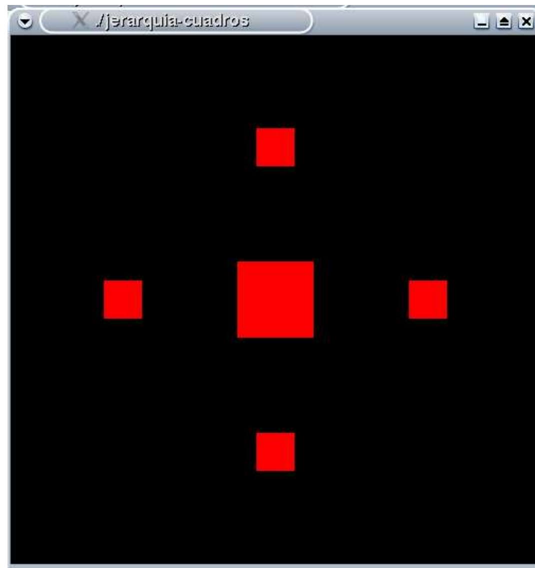


Figura 3.13. Jerarquía sencilla (salida del programa jerarquia-cuadro.c)

Si el procedimiento de dibujado se realiza de manera recursiva, se puede obtener un fractal como el de la figura:

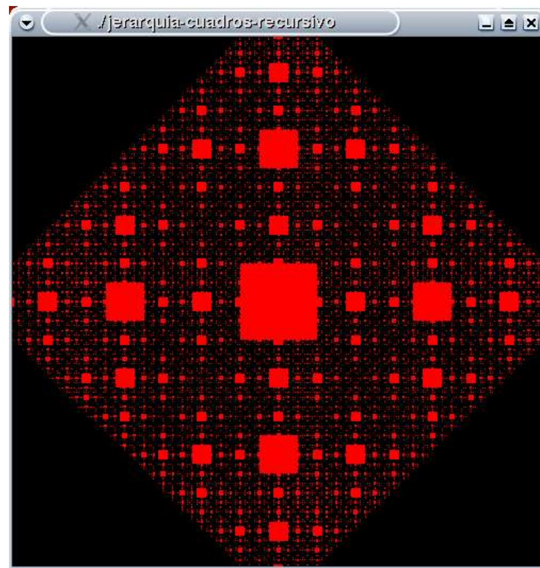


Figura 3.14. Jerarquía fractal (programa jerarquia-cuadro-recursivo.c)

Vamos a dibujar ahora una especie de "figura humanoide", con un cuerpo (un cubo), dos brazos (compuestos por tres partes, brazo, antebrazo y manos, con dos cilindros y un cubo respectivamente) dos piernas (similares a los brazos) y una cabeza (una esfera). Las figuras geométricas las vamos a dibujar mediante funciones de la librería GLUT.

El código para el dibujado es el siguiente:

1. Primero alejamos la cámara

```
glTranslatef(0,0,-16.0);
```

2. Situamos el cuerpo y lo dibujamos. Como queremos dibujar el cuerpo con un cubo, vamos a hacer que en el eje Y sea más grande que en el Z, y en el X un poco más... para ello lo escalamos. Pero este escalado sólo debe de afectar al cuerpo, por lo tanto, después de dibujar el cuerpo, desharemos el escalado, utilizando glPushMatrix y glPopMatrix. Las medidas del cuerpo, las definimos como constantes:

```
glTranslatef(0,BODY_HEIGHT/2,0);
glPushMatrix();
   glScalef(BODY_WIDTH,BODY_HEIGHT,BODY_LENGTH);
   glColor3f(0.0,0.3,0.8);
   glutSolidCube(1);
glPopMatrix();
```

3. A partir del centro del cuerpo, situamos el brazo derecho. Hay que tener en cuenta que un cubo se dibuja partir de su centro. Después vamos a rotar el brazo 30 grados alrededor del eje Z... si lo giramos directamente, estaremos girando el brazo por el centro... pero un brazo gira por su parte superior... para ello, trasladamos el pivote de giro a la parte superior del brazo, lo giramos, y devolvemos el pivote a su sitio, aplicamos el escalado, y ya podemos dibujar el brazo a partir de un cubo. Para dibujar la mano, trasladamos hacia abajo, aplicamos el escalado, y dibujamos. En la mano, no hemos aplicado el pushMatrix, porque su escalado no va a afectar a nada que se dibuje después, pero en caso contrario, deberíamos deshacerlo.

```
glPushMatrix();
   glTranslatef(-(BODY_WIDTH)/2,(BODY_HEIGHT-ARM_HEIGHT)/2,0);
   glTranslatef(0,ARM_LENGTH/2,0);
   glRotatef(-30,0,0,1);
   glTranslatef(0,-ARM_LENGTH/2,0);
   glPushMatrix();
      glScalef(ARM_WIDTH,ARM_HEIGHT,ARM_LENGTH);
      glutSolidCube(1);
   glPopMatrix();
   // ya tenemos el brazo... la mano
   glTranslatef(0,-(ARM_HEIGHT+ARM_WIDTH)/2,0);
   glColor3f(1,0.6,0.6);
   glScalef(ARM_WIDTH,ARM_WIDTH,ARM_LENGTH);
   glutSolidCube(1);
glPopMatrix();
```

4. Dibujamos el resto del cuerpo de manera análoga (ver código fuente en el anexo).

5. Finalmente, dibujamos la cabeza, como una esfera:

```
glColor3f(1,0.6,0.6);
glPushMatrix();
   glTranslatef(0,BODY_HEIGHT/2 + HEAD_RADIUS*3/4,0);
   glutSolidSphere(HEAD_RADIUS,10,10);
glPopMatrix();
```

El resultado del programa es el siguiente:

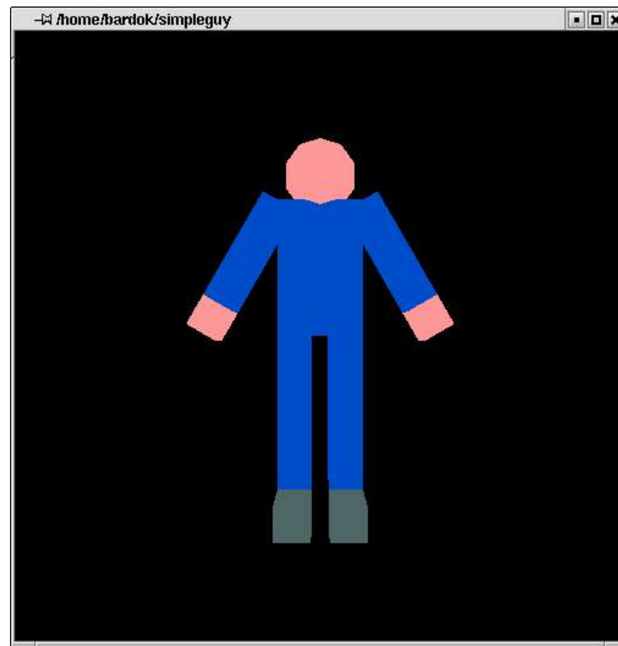


Figura 3.15. Salida del programa "simpleguy"

Vamos a estudiar más detalladamente el giro del brazo, ya que puede resultar difícil de ver:

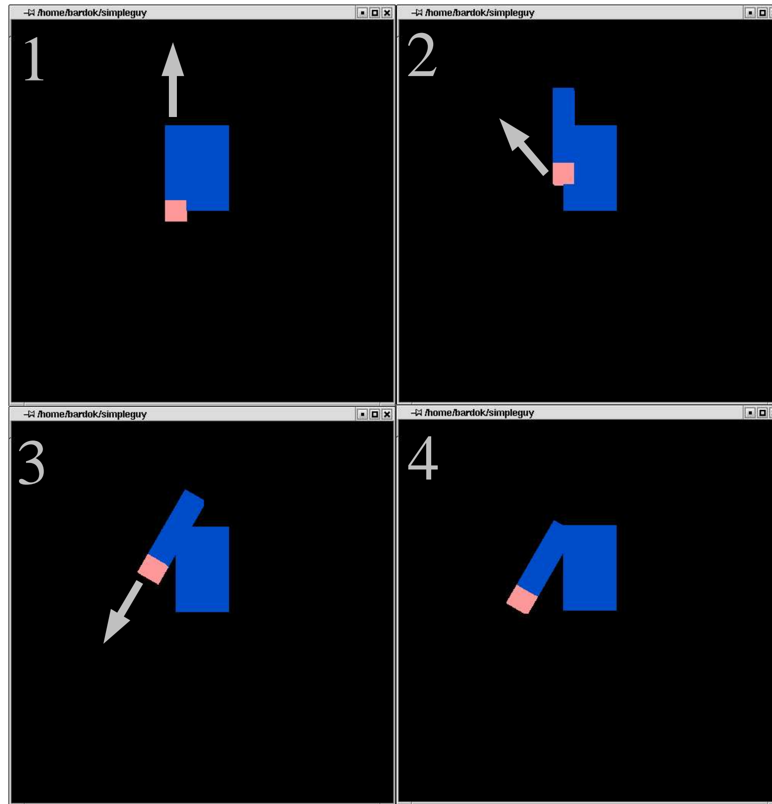


Figura 3.16. Rotación del brazo (simpleguy)

En el primer recuadro, tenemos el cuerpo junto con el brazo, sin rotar. Desplazamos el brazo de manera que su centro de giro se encuentre en la posción sobre la que originalmente queríamos pivotar el brazo. Giramos el brazo el ángulo deseado, y ahora, con ese giro desplazamos el brazo hacia abajo, según su eje de coordenadas, la misma distancia que lo habíamos desplazado antes.

# Capítulo 4: Animaciones

*"Cabe canem" (proverbio latino)*

En este capítulo vamos a describir de manera muy breve cómo animar las escenas que creamos con OpenGL.

En un principio, animar una figura es algo muy sencillo. Basta con:

1. actualizar los datos de la figura
2. borrar la pantalla
3. dibujar la figura
4. volver al punto 1

## 4.1 Los vectores para animar figuras

En muchos casos, dependiendo del tipo de animación, es muy recomendable el uso de vectores para representar hacia dónde se mueve un objeto y con qué velocidad.

Imaginemos que tenemos un objeto que se mueve a lo largo del eje X, a la velocidad de 1 unidad por frame, y que actualmente está en la posición (-3,3,0).

Podemos representar que el objeto se encuentra situado por un vector de traslación (-3,3,0), de esta manera:

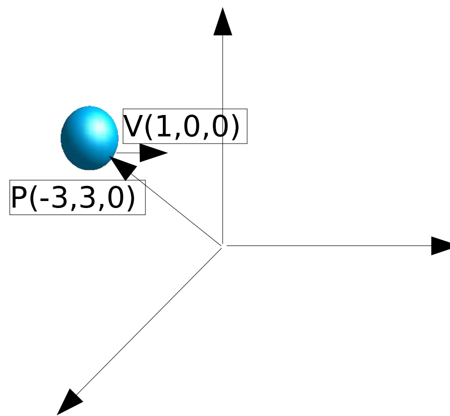


Figura 4.1. Representación de la posición y velocidad con vectores

P es el vector de posción, y V el de dirección. Si después de dibujar un frame, sumamos al vector de posción el de dirección, tenemos que en el siguiente frame el vector de posción será P(-2,3,0), con lo que la figura habrá avanzado por el eje X a la velocidad antes especificada.

Si queremos cambiar el movimiento, podemos variar la dirección del vector V, con lo que se modificará la dirección del movimiento, o su módulo, con lo que se modificará la velocidad del movimiento.

Vamos con un ejemplo. El programa que vamos a elaborar va a consistir en una esfera que se va a mover dentro de un cubo, de manera que cuando llegue a una de las paredes va a rebotar:

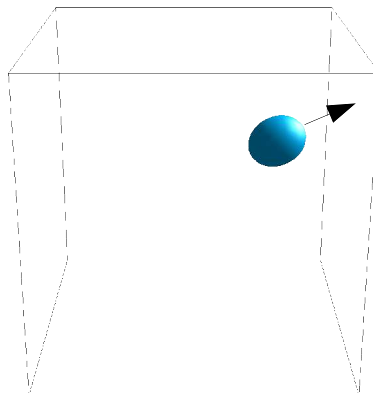

Figura 4.2. Comportamiento del programa "rebotes"

El esquema del dibujado será el siguiente:

- Comprobar que la esfera no esté en el límite del cubo
  - Sí está: invertir la componente del vector de dirección correspondiente al lado con el que ha rebotado (p. ej. si hemos llegado al límite por la derecha, el valor de la X se volverá negativo...)
- Sumar el vector de dirección al vector de posición de la esfera

- Dibujar la esfera
- Volver al comienzo del bucle

Para este programa vamos a definir una clase que se encargue de representar la esfera, y que guardará su posición y velocidad. Los métodos que proporciona esta clase se definen en la siguiente tabla:

| Métodos | Función |
|---|---|
| `Tsphere (float maxpos, float speed);` | Crea una instancia de la clase. La esfera rebotará si llega a la posción +-"maxpos", y tendrá una velocidad "speed". Su situación y dirección iniciales son aleatorias. |
| `void test();` | Comprueba que la figura esté dentro de los márgenes, y si no le está, corregirá su dirección. Después, actualiza su posción |
| `void modifySpeed(float inc);` | Incrementa la velocidad en "inc" |
| `float * getPosv();` | Obtiene un vector con la posción de la esfera (en la posición 0, la coord. X, en la 1 la Y y en la 2 la Z) |

Tabla 4.1. Métodos de la clase TSphere

Para las animaciones vamos a utilizar varias funciones de GLUT. Por un lado, glutDisplayFunc(), que como ya hemos visto, dibuja un frame, y glutIdleFunc(), que especifica la operación que se realizará entre dos frames. Además, agruparemos todas las llamadas a OpenGL previas al dibujado de cualquier frame en una función de inicialización.

Con esto, el código quedará de la siguiente manera:

Vamos a utilizar una variable global para indicar la posición de la esfera, y otra para apuntar a la propia esfera:

```
float * pos;
TSphere * sphere;
```

La inicialización de OpenGL la vamos a realizar en una función aparte:

```
void initgl()
{
glEnable(GL_DEPTH_TEST);
glClearColor(0.0,0.0,0.0,0.0);

glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(60.0,1.0,1.0,100.0);

sphere = new TSphere(5,1);

glMatrixMode(GL_MODELVIEW);
gluLookAt(3,3,14,0,0,0,0,1,0);
}
```

En esta inicialización vamos a hacer todo lo relativo a OpenGL que sólo tiene que hacerse una única vez: establecer el color de fondo, habilitar el

modo de perspectiva, alejar la cámara, y de paso, crear la esfera, que se va a mover en el rango [-5,5] con una velocidad de 1. Para alejar la cámara, utilizaremos la función “gluLookAt”. Esta función posiciona la cámara, ahorrando la realización de giros y traslaciones manuales, de la siguiente manera: recibe tres ternas: la posción de la cámara en el espacio, la posición hacia la que está mirando, y un vector que señala a la parte superior de la escena... en este caso:

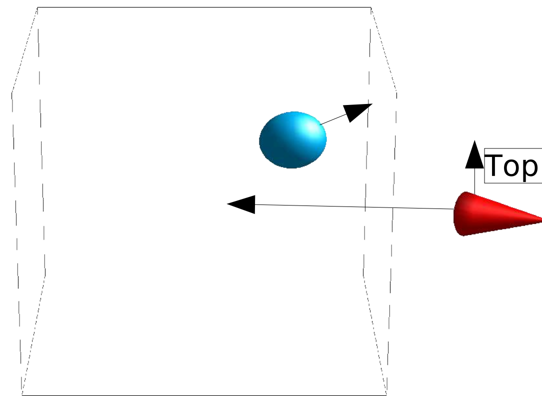


Figura 4.3. Posición de la cámara en el ejemplo "sphere-rebotes"

La cámara se situa por delante, y ligeramente por encima del eje de coordenadas, y algo desplazada en el eje X. Está mirando hacia el centro de coordenadas, y la parte superior de la escena está apuntada por el vector “top”.

La siguiente función será la función de dibujado de cada frame, su cuerpo será el siguiente:

```
void display(void)
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glColor3f(1.0,1.0,1.0);
  glutWireCube(10);
  glPushMatrix();
    glColor3f(0.0,0.0,1.0);
    pos = sphere->getPosv();
    glTranslatef(pos[0],pos[1],pos[2]);
    glutSolidSphere(1,10,10);
  glPopMatrix();
  glFlush();
}
```

Por cada frame, se borra la pantalla, se dibuja el cubo, se dibuja la esfera (deshaciendo su transformación, para no tener que rehacer la transformación de la cámara en cada pasada).

Finalmente vamos a crear una función llamada “idle” que va a ejecutar la actualización de la esfera entre dos frames, y va a esperar 33 milisegundos para dibujar el siguiente fotograma:

```
void idle(void)
{
  sphere->test();
  usleep(33);
```

```
    glutPostRedisplay();
}
```

La última llamada a glutPostRedisplay() hace que tras la actualización se dibuje el siguiente frame.

Para registrar estas funciones hacemos, en el programa principal:

```
initgl();
glutDisplayFunc(display);
glutIdleFunc(idle);
```

Pero si ejecutamos el programa, vemos un pequeño problema al visualizar: hay parpadeo (flickering). Esto se debe a que la pantalla está continuamente borrándose (se pone a negro) y redibujando la imágen. Para que esto no ocurra hemos de utilizar el llamado "doble buffering".

## *4.2 El doble buffering*

El doble buffering es una técnica muy simple, que dibuja la escena en un buffer fuera de la pantalla, mientras que la imágen de la pantalla no se toca para nada. Una vez que la imágen se ha dibujado en el buffer de fondo, se intercambia por el de la pantalla, de manera que eliminamos el parpadeo. El siguiente frame lo dibujaremos en lo que antes era el buffer de pantalla (ahora oculto), y luego se cambiará, y así sucesivamente.

Activar el doble buffering es algo muy sencillo. Hemos de indicar a GLUT que vamos a utilizar doble buffering al crear la pantalla con GLUT_DOUBLE, en lugar de GLUT_SINGLE:

```
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);
```

Y después de dibujar un frame hemos de cambiar el buffer oculto por el activo con:

```
glutSwapBuffers();
```

Una vez añadido esto, el código quedaría como el que se encuentra en el anexo.

El resultado del programa es el siguiente:

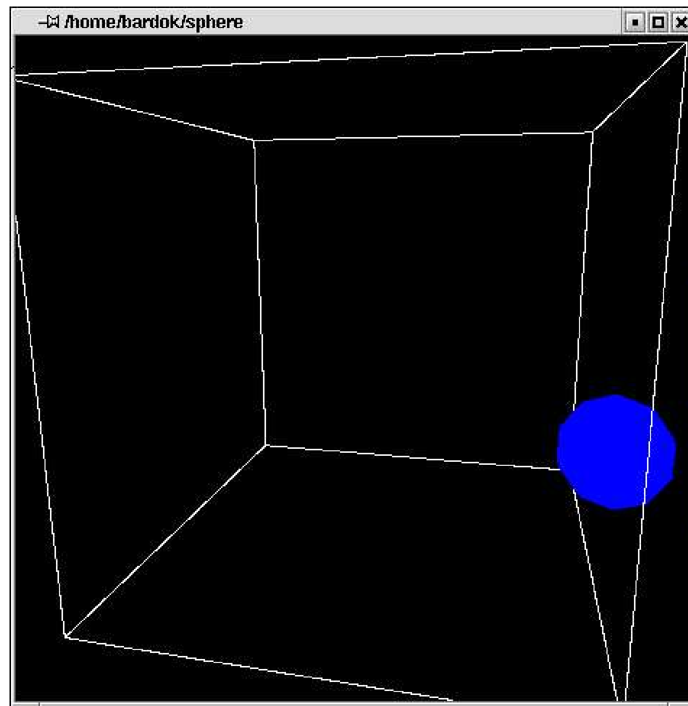

Figura 4.4. Salida del programa "sphere-rebotes"

Y con una pequeña modificación (sphere-rebotes-multi) podemos conseguir una locura como esta:

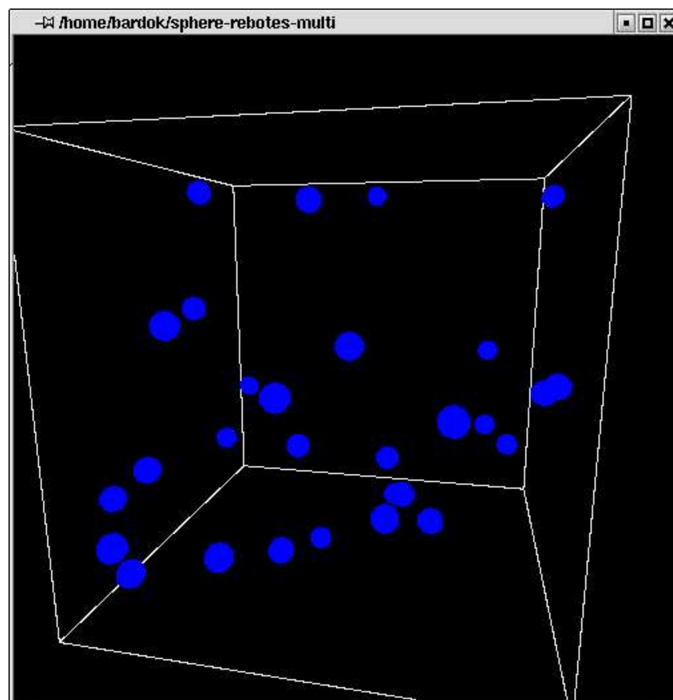


Figura 4.5. Salida del programa "sphere-rebotes-multi"

# Capítulo 5: Iluminación

*"Ratones: Los roedores durante muchos años han estado "compartiendo" el alimento del hombre, son omnívoros, aunque muestran preferencia por las semillas de los cereales y productos derivados. Cuando éstas faltan pueden comer las cosas más extrañas que nos podamos imaginar : jabón, cuero, cera, plásticos o papel etc" (www.amezdy.com/ratones.htm)*

La iluminación nos permitirá crear escenas más realistas con OpenGL. Cada implementación de OpenGL puede mantener un número de luces mayor o igual a 8 (nunca menor), aunque no se recomienda abusar de éstas, ya que el consumo de CPU por la utilización de luces es muy grande.

## *5.1 El modelo de iluminación en OpenGL*

La iluminación de OpenGL se basa en luces y materiales. Una luz es una fuente de iluminación para la escena. Emite un haz de luz de un color determinado, dividido en las tres componentes de color RGB. Un material determina la cantidad de cada color que refleja un objeto determinado.

Por ejemplo, si un objeto tiene un material de color rojo -RGB(1,0,0)-, es decir, refleja todo el color rojo, y es iluminado por una luz blanca -RGB (1,1,1)-, reflejará toda la componente de color rojo, pero nada de la verde y azul, por lo que se verá de color rojo. Si este mismo objeto fuese iluminado por una luz verde -RGB(0,1,0)-, se vería de color negro, al no tener nada de luz roja que poder reflejar. Además, dependiendo del tipo de luz, el color

final con el que se vea el objeto puede verse afectado por el ángulo de incidencia de la luz, la distancia a ésta, etc.

### 5.1.1 Luces

Como se ha dicho, una luz aporta iluminación a una escena, según unas determinadas componentes de color.

A partir de ahora distinguiremos entre fuente de luz, como entidad que proporciona luz a una escena, y luz, como aportación de esa fuente a la iluminación de la escena.

Una fuente puede emitir tipos diferentes de luz, que son complementarias, y pueden darse a la vez para un cierto tipo de luz.

Los tipos de luz son:

- "Emitted" (emitida): es la luz emitida por un objeto. No se ve afectada por ningún otro tipo de luz. Por ejemplo, un fuego emite una determinada luz, y si lo miramos, lo veremos de ese color, independientemente del color de las luces que estén apuntando al fuego.
- "Diffuse" (difusa): es la luz que indice sobre un objeto, y proviene de un determinado punto. La intensidad con la que se refleja en la superficie del objeto puede depender del ángulo de incidencia, dirección, etc. Una vez incide sobre un objeto se refleja en todas direcciones.
- "Specular" (especular): es la luz que, al incidir sobre un objeto, se ve reflejada con un ángulo similar al de incidencia. Podemos pensar que es la luz que produce los brillos.
- "Ambient" (ambiental): podemos considerarlo como los restos de luz que aparecen defido a la reflexión residual de una luz que ya se ha reflejado sobre muchos objetos, y es imposible determinar su procedencia. Es algo así como la iluminación global de una escena.

### 5.1.2 Materiales

Un material define, para un determinado objeto, qué componentes refleja de cada tipo de luz.

Por ejemplo, un plástico rojo, emitirá toda la componente roja de las luces ambiental y difusa, pero generalmente emitirá brillos de color blanco bajo una luz blanca, por lo que su componente especular será de color blanco. De este modo, la componente de emisión será negra, puesto que un plastico no emite luz (aunque si es un plástico radioactivo, puede emitir un leve destello fantasmal ;-)

En un modelo con iluminación, el color de un objeto se define por su material, no por el color de sus vértices, tal y como hemos visto hasta ahora.

### 5.1.3 Normales

La iluminación en OpenGL se calcula a nivel de vértice, es decir, por cada

vértice, se calcula su color a partir del material activo, las luces activas, y cómo estas luces inciden sobre el vértice. Para saber cómo incide una luz sobre un vértice, empleamos la normal del vértice, un vector, que, generalmente, será perpendicular a la cara que estemos dibujando, aunque podremos variarlo para conseguir distintos efectos.

En la figura, podemos ver cómo incide la misma luz en el mismo objeto, pero con las normales cambiadas.

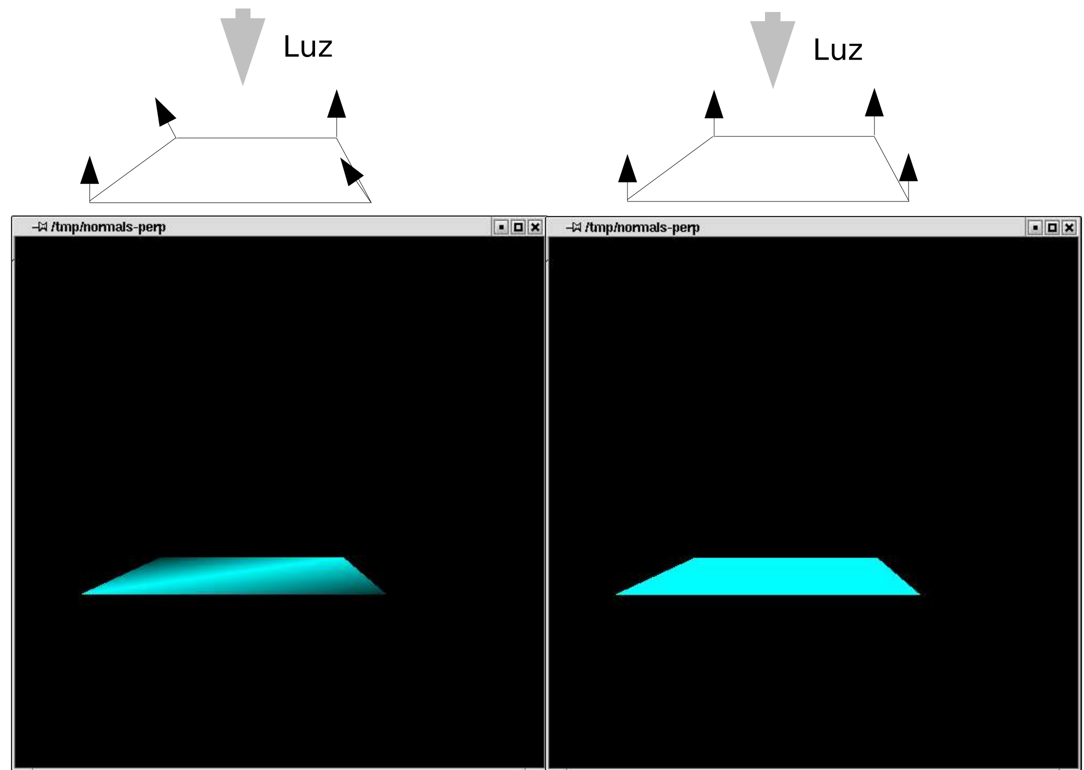


Figura 5.1. Efecto de las normales en la iluminación

## *5.2 Utilizando iluminación en OpenGL*

### 5.2.1 Luces

El primer paso para utilizar la iluminación en una aplicación OpenGL es activarla, mediante la llamada

```
glEnable(GL_LINGHTING);
```

Una vez activada, hemos de establecer las propiedades de cada luz en la escena, y activarlas. La especificación inicial de OpenGL comtempla que, al menos, cada implementación debe de poder definir 8 luces, identificadas por las constantes GL_LIGHTn, dónde 'n' es el número de la luz, comenzando por 0.

Para establecer las propiedades de una luz utilizaremos llamadas a las funciones del tipo glLight*(). Las propiedades de toda luz son las siguientes:

– Posición/Dirección:

Indica la posición/dirección de la luz, y especifica si ésta es una luz posicional o direccional. Una luz posicional tiene una posición concreta en el espacio, mientras que una luz direccional consiste en un conjunto de haces de luz paralelos. Un ejemplo de luz posicional es una lampara, mientras que, debido a su distancia, podríamos considerar al sol como una fuente de luz direccional. Las luces posicionales pueden ser de dos tipos: luces puntuales, que emiten luz a su alrededor de manera radial, y en todas direcciones, y luces focales, que emiten luz en una dirección concreta, en un radio de acción con forma de cono (como un foco).

Podemos ver los diferentes tipos de luces en la siguiente figura:

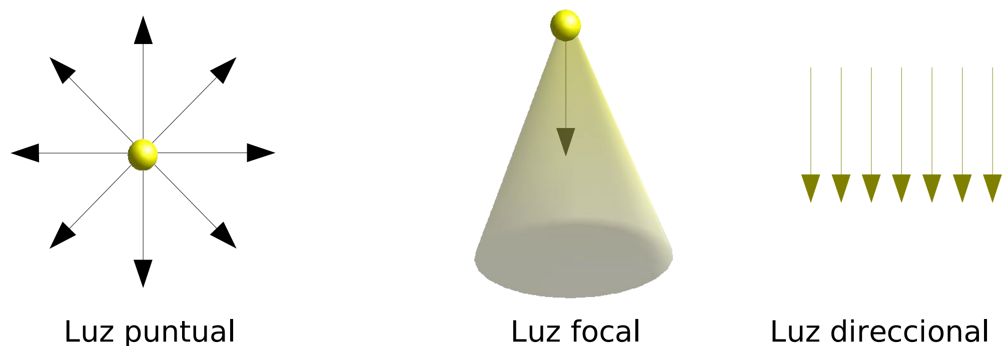


Figura 5.2. Tipos de luces

Para establecer esta propiedad utilizaremos la llamada:

```
glLightfv(GL_LIGHTn,GL_POSITION,val_ptr);
```

"val_prt" es un puntero a un vector de cuatro componentes de tipo float, de la forma (x,y,z,w). En el caso de que w sea 1, estaremos ante una luz posicional, y su posción está determinada por (x,y,z). Si w es 0, la luz es direccional, y su dirección es el vector (x,y,z).

- Dirección del foco:

  En el caso de una luz focal, debemos establecer su dirección. Esto lo haremos con la llamada:

  ```
  glLightfv(GL_LIGHTn,GL_SPOT_DIRECTION,val_prt);
  ```

  "val_ptr" es un puntero a un vector con la dirección, en formato (x,y,z).

- Apertura del foco:

  El ángulo de apertura del foco se define mediante:

  ```
  glLightf(GL_LIGHTn,GL_SPOT_CUTOFF,val);
  ```

  "val" expresa en grados la mitad del ángulo de apertura del foco.

- Atenuación del foco:

  La atenuación del foco (degradación de la intensidad a medida que nos acercamos al borde) se define mediante:

  ```
  glLightf(GL_LIGHTn,GL_SPOT_EXPONENT,val);
  ```

- Intensidad de la luz:

Define el color ambiental, difuso y especular de la luz. Se define mediante la llamada:

```
glLightfv(GL_LIGHTn,GL_[AMBIENT|DIFFUSE|SPECULAR],val_ptr);
```

“val_ptr” es un puntero a un vector de cuatro componentes de color RGBA.

– Atenuación de la luz

Define la pérdida de intensidad de la luz a medida que nos alejamos del foco (no afecta a las luces direccionales). Se establece mediante:

glLightf(GL_LIGHTn,GL_[CONSTANT|LINEAR|QUADRATIC]_ATTENUATION,val);

El factor de atenuación de la luz se calcula según la fórmula:

$$\frac{1}{k_c + k_l d + k_q^2}$$

Una vez establecidas las propiedades de una determinada luz, las activaremos con la llamada:

```
glEnable(GL_LIGHTn);
```

### 5.2.2 Materiales

Tras definir las luces, antes de dibujar un objeto, hemos de definir su material.

Para ello, estableceremos los valores de las diversas características del mismo, mediante las funciones del tipo glMaterial*(). Estas funciones reciben tres parámetros:

1. Caras del polígono a las que se aplica el material. Podemos aplicar el material a la cara frontal (GL_FRONT) o a ambas caras del polígono (GL_FRONT_AND_BACK).
2. Característica que definimos, mediante su correspondiente constante.
3. Valor de la característica.

Las diferentes características son:

– Color del material:

Define el comportamiento del material para los distintos tipos de luz descritos anteriormente. El color ambiental, difuso y especular definen los colores del espectro que este material refleja de las luces que recibe. El color emitido es el tipo de luminosidad que posee el material, intedependientemente de la luz que le afecte.

Se define mediante la llamada:

```
glMaterialfv(GL_FRONT[_AND_BACK],[GL_AMBIENT|GL_DIFUSSE|
GL_AMBIENT_AND_DIFUSSE|GL_SPECULAR],val_ptr);
```

“val_ptr” es un puntero a un conjunto de valores RGBA que especifica el

color de la componente específica.

Como antes se ha comentado, es habitual que el comportamiento ante la luz difusa y la ambiental sea el mismo, por lo que es posible establecer ambos al mismo tiempo.

- Brillo de los reflejos:

  La intensidad del brillo puede definirse a través de la llamada:

```
glMaterialf(GL_FRONT[_AND_BACK],GL_SHININESS,val);
```

Además, podemos cambiar alguno de los valores establecidos para los colores de un material mediante la función glColorMaterial(). Mediante esta función, que debemos activar, podemos cambiar alguno de los colores del material.

Esta opción se activa mediante:

```
glEnable(GL_COLOR_MATERIAL);
```

y una vez activada, establecemos la propiedad que queremos cambiar mediante:

```
glColorMaterial(GL_FRONT[_AND_BACK],[GL_AMBIENT|GL_DIFUSSE|
GL_AMBIENT_AND_DIFUSSE|GL_SPECULAR]);
```

y cambiamos el color reflejado de ese tipo de luz mediante, por ejemplo:

```
glColor3f(r_val,g_val,b_val);
```

Para evitar complicaciones, debemos desactivar esta opción al terminar de usarla. Veremos la utilidad de esta opción algo más adelante, con un ejemplo.

### 5.2.3 Ejemplo: iluminación direccional sobre una superficie

Vamos a ver cómo se ha realizado el ejemplo con el que comprobamos el efecto de las normales sobre la iluminación.

Inicialmente vamos a definir una serie de arrays para los colores que vamos a emplear:

El color ambiental y difuso del material:

```
GLfloat mat_color [] = {0.0,1.0,1.0,1.0};
```

El color difuso y especular de la luz:

```
GLfloat light_color [] = {1.0,1.0,1.0,1.0};
```

El color ambiental de la luz (si algo está fuera de la luz, no es iluminado):

```
GLfloat light_ambient [] = {0.0,0.0,0.0,1.0};
```

El valor de las normales (como el cuadrado está en el plano (x,z), las normales van hacia arriba).

```
GLfloat normal [] = {0.0,1.0,0.0};
```

El valor de la dirección de la luz (es una luz situada en el eje Y, en el infinito, luego sus rayos irán hacia abajo en el eje Y):

```
GLfloat light_dir [] = {0.0,1.0,0.0,0.0};
```

Seleccionamos el color de fondo, y borramos la pantalla:

```
glClearColor(0.0,0.0,0.0,0.0);
glClear(GL_COLOR_BUFFER_BIT);
```

Activamos la luz, y sus características, excepto la dirección:

```
glEnable(GL_LIGHTING);

glLightfv(GL_LIGHT0,GL_AMBIENT,light_ambient);
glLightfv(GL_LIGHT0,GL_DIFFUSE,light_color);
glLightfv(GL_LIGHT0,GL_SPECULAR,light_color);
glEnable(GL_LIGHT0);
```

Seleccionamos el color del material:

```
glMaterialfv(GL_FRONT,GL_AMBIENT_AND_DIFFUSE,mat_color);
```

Establecemos la perspectiva y posicionamos al observador:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(60.0,1.0,1.0,100.0);

glMatrixMode(GL_MODELVIEW);
glTranslatef(-0.3,-0.6,-4.0);
```

Establecemos la dirección de la luz:

```
glLightfv(GL_LIGHT0,GL_POSITION,light_dir);
```

Esta dirección la establecemos después de la transformación de la cámara, porque queremos que, en este caso, esté también sometida a la transformación de la cámara (por ejemplo, si dibujamos una lampara, su posición en la escena será relativa a dónde esté el espectador de la escena).

Dibujamos el cuadrado, y antes de cada vertice, establecemos su normal. En el caso de que el símbolo “SINGLE_NORMAL” esté definido, la normal será siempre la misma. En caso contrario, los vértices 1 y 3 tendrán una normal distinta, para lograr el efecto que se muestra en la figura 5.1:

```
glBegin(GL_QUADS);

#ifdef SINGLE_NORMAL
  glNormal3fv(normal);
#else
  glNormal3f(1.0,1.0,-1.0);
#endif
  glVertex3f(-1.0,0.0,-1.0);

  glNormal3fv(normal);
  glVertex3f(-1.0,0.0,1.0);

#ifdef SINGLE_NORMAL
  glNormal3fv(normal);
#else
  glNormal3f(-1.0,1.0,-1.0);
#endif
  glVertex3f(1.0,0.0,1.0);
  glNormal3fv(normal);
  glVertex3f(1.0,0.0,-1.0);
glEnd();
```

Finalmente, forzamos el dibujado, esperamos, y salimos del programa:

```
glFlush();

sleep(20);
exit(0);
```

### 5.2.4 Ejemplo: moviendo un foco alrededor de una esfera

En este ejemplo vamos a mover un foco alrededor de una esfera, y vamos a ver cómo afectan los distintos tipos de luz. Para ello utilizaremos dibujado con doble buffer, y zBuffer.

El dibujado se hará de este modo:

– Inicialmente definimos las características de la luz:

```
float light_ambient [] = {0.0,0.2,0.0,1.0};
float light_diffuse_specular [] = {0.8,0.8,0.8,1.0};
float light_pos [] = {0.0,0.0,2.0,1.0};
float spot_dir [] = {0.0,0.0,-1.0};
float spot_cutoff = 30.0;
float spot_exponent = 1.0;
```

– Las características del material de la esfera:

```
float mat_ambient_diffuse [] = {0.0,0.8,1.0,1.0};
float mat_specular [] = {0.7,0.0,0.0,1.0};
float mat_emission [] = {0.0,0.0,0.0,1.0};
float mat_shininess = 0.4;
```

– Vamos a poner un pequeño cono cuyo material tenga una “emisión” similar al color de la luz, que nos indicará dónde está el foco. Este es su color de “emisión”:

```
float focus_emission [] = {0.8,0.8,0.8,1.0};
```

– Definimos un par de variables que indican la rotación del foco:

```
float rot_angle_y = 0.0;
float rot_angle_x = 0.0;
```

– Habilitamos el zBuffer y establecemos el color de fondo:

```
glEnable(GL_DEPTH_TEST);
glClearColor(0.0,0.0,0.0,0.0);
```

– Activamos las luces y sus características. La primera llamada establece que las caras traseras de los polígonos no se iluminarán:

```
glLightModeli(GL_LIGHT_MODEL_TWO_SIDE,GL_FALSE);
glEnable(GL_LIGHTING);
glLightfv(GL_LIGHT0,GL_AMBIENT,light_ambient);
glLightfv(GL_LIGHT0,GL_DIFFUSE,light_diffuse_specular);
glLightfv(GL_LIGHT0,GL_SPECULAR,light_diffuse_specular);
glLightf(GL_LIGHT0,GL_SPOT_CUTOFF,spot_cutoff);
glLightf(GL_LIGHT0,GL_SPOT_EXPONENT,spot_exponent);
glEnable(GL_LIGHT0);
```

– Establecemos el material  de la esfera:

```
glMaterialfv(GL_FRONT,GL_AMBIENT_AND_DIFFUSE,
mat_ambient_diffuse);
glMaterialfv(GL_FRONT,GL_SPECULAR,mat_specular);
glMaterialf(GL_FRONT,GL_SHININESS,mat_shininess);
```

– Establecemos la perspectiva y trasladamos la cámara:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(60.0,1.0,1.0,100.0);

glMatrixMode(GL_MODELVIEW);
glTranslatef(0.0,0.0,-5.0);
```

Todos los pasos anteriores se engloban en lo que sería la función de inicialización.

– A continuación tenemos el bucle de dibujado (glutDisplayFunc). En él, una vez borrada la pantalla, rotamos la luz, la posicionamos, y establecemos su dirección, que serán relativas a su rotación. Después, dibujamos un cono en la posición de la luz. Este cono tendrá como color de emisión el definido como tal. El cambio lo hacemos con glColorMaterial(). Una vez dibujado, deshacemos estas transformaciones, dibujamos la esfera, dibujamos y cambiamos los buffers:

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glPushMatrix();
   glRotatef(30.0,0.0,0.0,1.0);
   glRotatef(rot_angle_y,0.0,1.0,0.0);
   glRotatef(rot_angle_x,1.0,0.0,0.0);
   glLightfv(GL_LIGHT0,GL_POSITION,light_pos);
   glLightfv(GL_LIGHT0,GL_SPOT_DIRECTION,spot_dir);
      glTranslatef(light_pos[0],light_pos[1],light_pos[2]);
      glColorMaterial(GL_FRONT,GL_EMISSION);
      glEnable(GL_COLOR_MATERIAL);
         glColor4fv(focus_emission);
         glutSolidCone(0.2,0.5,7,7);
         glColor4fv(mat_emission);
      glDisable(GL_COLOR_MATERIAL);
glPopMatrix();
glutSolidSphere(1.0,20,20);
glFlush();
glutSwapBuffers();
```

Finalmente, definimos que, entre frame y frame (glutIdleFunc), la acción a realizar será actualizar la rotación del foco:

```
rot_angle_y = (rot_angle_y > 360.0)?0:rot_angle_y + ROT_INC;
rot_angle_x = (rot_angle_x > 360.0)?0:rot_angle_x + ROT_INC/
(2*3.1416);
glutPostRedisplay();
```

Y aquí tenemos un frame de la animación que hemos creado:

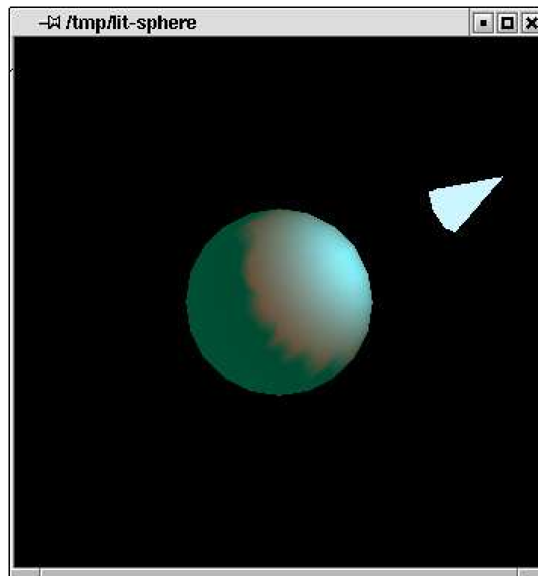


Figura 5.3. Salida del programa "lit-sphere"

Un efecto curioso es el hecho de observar los reflejos de color rojo que se emiten desde los bordes de la esfera, gracias a que hemos definido este color como el color “especular” del material.

Como puede observarse, en todos estos ejemplos hay gran cantidad de trozos de código que se repiten. Por ello, es conveniente la agrupación de estas llamadas en funciones de motores gráficos.

## 5.2.5 Ejercicio propuesto

Como final de este capítulo se propone un ejercicio, consistente en una modificación del programa “sphere-rebotes-multi”, de manera que la escena esté iluminada por una luz direccional, blanca, situada en el infinito, en una posición apuntada por el vector (1,1,1), y cada esfera tenga su propio color (aleatorio) definido por las características ambiental y difusa del material. El cubo no debe de verse afectado por la luz al dibujarse.

El resultado debería de ser parecido al siguiente:

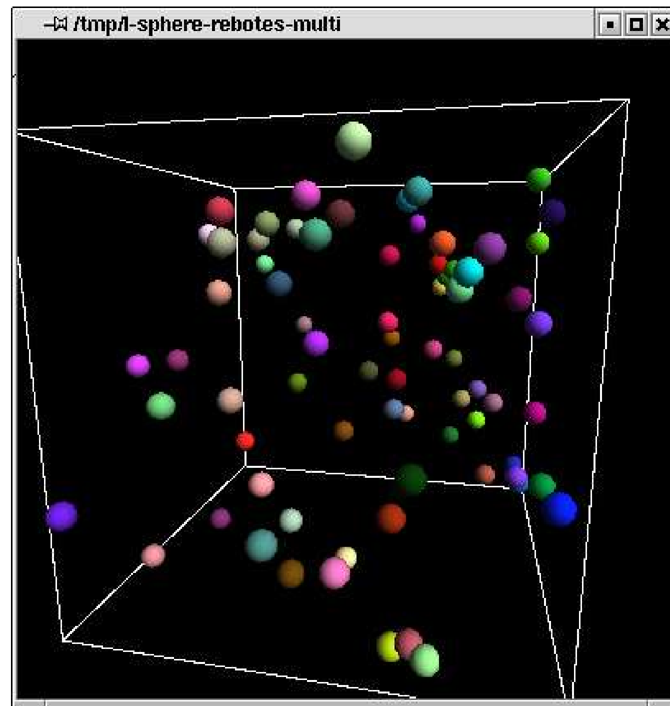


Figura 5.4. Salida del programa "l-sphere-rebotes-multi"

Una solución a este ejercicio aparece en el anexo A, bajo el epígrafe “l-sphere-rebotes-multi.cpp”.

# Capítulo 6: Texturas

*“Tú me das cremita, yo te doy cremita...” (La canción del verano...)*

Las texturas permiten personalizar aún más el material de un objeto, y nos permiten obtener imágenes mucho más realistas en nuestras escenas. Por ejemplo, si queremos dibujar una pared de ladrillo, tenemos dos opciones: dibujar cada ladrillo, definiendo su material con colores, y dibujar el cemento entre los ladrillos, a base de polígonos, definiendo también su material, o dibujar un único cuadrado con la extensión de la pared, y hacer que su material, sea, por ejemplo, la foto de una pared de ladrillo.

## 6.1 Las coordenadas de textura

Para saber qué partes de una imágen se dibujan en un polígono (por ejemplo un triángulo), utilizamos lo que se denominan coordenadas de textura, o coordenadas UV. Estas coordenadas están representadas por números reales de 0 a 1, de la siguiente manera:

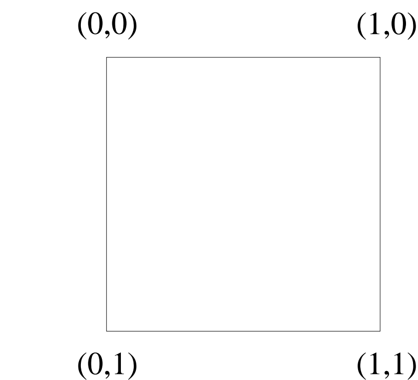

Figura 6.1. Coordenadas de textura

De esta manera, si queremos dibujar un triángulo con una textura, y aplicacamos a cada vértice las coordenadas de textura indicadas en la figura, tenemos el siguiente resultado:

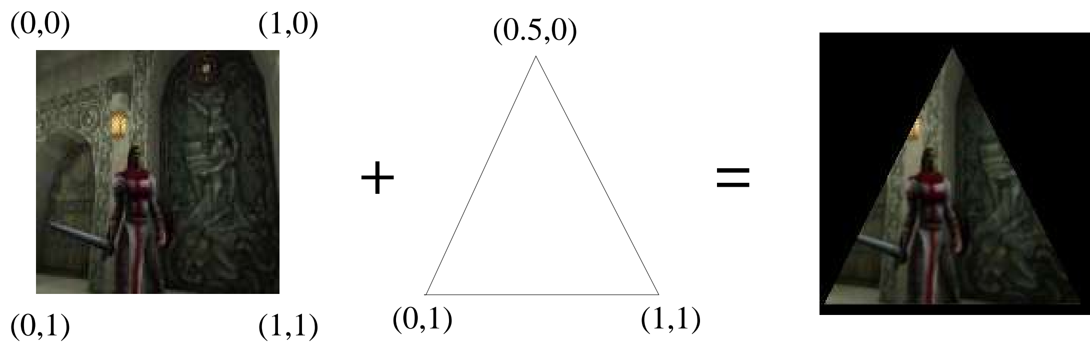

Figura 6.2. Mapeado de textura

## *6.2 Aplicar las texturas*

Para aplicar una textura tenemos que seguir una serie de pasos muy definidos:

1. Creamos la textura
2. Definimos las condiciones en que se va a aplicar la textura
3. Habilitar la aplicación de texturas
4. Dibujar las escenas, proporcionando las coordenadas de textura

Como ejemplo sencillo, vamos a ir viendo cómo se ha creado el ejemplo del mapeado de texturas:

– Creación de la textura [TEXT]:

  Primero hemos de obtener un identificador para la textura. Para ello pedimos a OpenGL que nos devuelva un identificador de textura libre:

```
int texture;
glGenTextures(1,&texture);
```

Activamos la textura como textura activa:

```
glBindTexture(GL_TEXTURE_2D,texture);
```

Creamos la textura. El modo más sencillo de hacerlo es a través de una función de GLU que crea la textura y sus variaciones a aplicar según la distancia a la que se encuentre:

```
gluBuild2DMipmaps( GL_TEXTURE_2D, gimp_image.bytes_per_pixel,
gimp_image.width, gimp_image.height,GL_RGB, GL_UNSIGNED_BYTE,
gimp_image.pixel_data );
```

Con el primer parámetro indicamos el tipo de textura. En este caso (GL_TEXTURE_2D) es una textura 2D. Los siguentes parámetros indican el número de bytes por cada pixel, que dependerán de la imágen (p. ej. 3 para RGB y 4 para RGBA; en este caso, la imágen es de tipo RGB), su anchura y altura (que han de ser pares, y muy recomendable, múltiplos de 2), el formato de los datos (GL_RGB, GL_RGBA,...), el tipo de los datos, que en nuestro caso vendrá dado por bytes sin signo, por lo que usaremos la constante GL_UNSIGNED_BYTE, y finalmente, un puntero a los datos.

En este ejemplo, hemos utilizado una estructura para generar la imágen con el siguiente formato:

```
static const struct {
  unsigned int     width;
  unsigned int     height;
  unsigned int     bytes_per_pixel; /* 3:RGB, 4:RGBA */
  unsigned char    pixel_data[128 * 128 * 3 + 1];
} gimp_image = {
128, 128, 3, [...] }
```

Este formato de datos es producido de manera automática por el programa “Gimp” al grabar una imágen como un archivo en “c”.

- Definir las condiciones en que se va a aplicar la textura:

Estas condiciones se establecen a través de la función glTexEnvf().

```
glTexEnvf( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE );
```

En nuestro caso, hemos especificado que la textura se va a fundir con el color de fondo del polígono (GL_MODULATE), aunque bien podría sustituirlo (usando GL_REPLACE). La fusión con el color de fondo se utiliza para aplicar la iluminación a un objeto con textura. En el ejemplo vemos cómo se visualiza un objeto iluminado, con la textura aplicada mediante GL_MODULATE, y mediante GL_REPLACE:

Figura 6.3. Diferencias entre GL_REPLACE y GL_MODULATE

- Habilitar la aplicación de texturas:

  Mediante:

```
glEnable(GL_TEXTURE_2D);
```

- Dibujar la geometría proporcionando las coordenadas de textura de cada vértice:

```
glBegin(GL_TRIANGLES);
   glTexCoord2d(0.0,1.0);
   glVertex3f(-0.5,-0.5,0.5);
   glTexCoord2d(1.0,1.0);
   glVertex3f(0.5,-0.5,0.5);
   glTexCoord2d(0.5,0.0);
   glVertex3f(0.0,0.5,0.5);
glEnd();
```

Si en algún momento queremos cambiar la textura activa, tan sólo hemos de indicar qué textura queremos activar mediante la llamada:

```
glBindTexture(GL_TEXTURE_2D,texture);
```

“texture” es el identificador de la textura que queremos activar.

El código completo de los ejemplos utilizados puede verse en el anexo A.

### 6.2.1 Repetición de texturas

Imaginemos que queremos dibujar una pared de ladrillos junto a un cesped, y tenemos estas dos texturas, para la pared y el suelo:

Figura 6.4. Texturas para suelo y pared

El suelo va a ser u cuadrado, y la pared un rectángulo, y vamos a mapear las esquinas del suelo y de la pared con las esquinas de de las texturas (código completo en el anexo A):

```
glBindTexture(GL_TEXTURE_2D,texture_floor);
glBegin(GL_QUADS);
   glTexCoord2d(0.0,0.0);
   glVertex3f(-6.0,0.0,-6.0);
   glTexCoord2d(0.0,1.0);
   glVertex3f(-6.0,0.0,6.0);
   glTexCoord2d(1.0,1.0);
   glVertex3f(6.0,0.0,6.0);
   glTexCoord2d(1.0,0.0);
   glVertex3f(6.0,0.0,-6.0);
glEnd();

glBindTexture(GL_TEXTURE_2D,texture_wall);
glBegin(GL_QUADS);
   glTexCoord2d(0.0,0.0);
   glVertex3f(-6.0,4.0,-6.0);
   glTexCoord2d(0.0,1.0);
   glVertex3f(-6.0,0.0,-6.0);
   glTexCoord2d(1.0,1.0);
   glVertex3f(6.0,0.0,-6.0);
   glTexCoord2d(1.0,0.0);
   glVertex3f(6.0,4.0,-6.0);
glEnd();
```

Un aspecto a resaltar es que la operación glBindTexture, para seleccionar cuál es la textura activa en este momento, tiene que realizarse fuera del contexto de glBegin/glEnd, o de lo contrario no funcionará.

El resultado de dibujar esto es el siguiente:

Figura 6.5. Salida del programa "texture-no-clamp"

Como podemos observar, los ladrillos de la pared salen demasiado alargados, y el suelo demasiado distorsionado, por el hecho de que estamos agrandando las texturas para acomodarlas al tamaño de los objetos.

Pero, ¿y si repetimos las texturas? Por ejemplo, podríamos repetir la textura de la en 3 divisiones verticales, y la del suelo 36 veces (6 horizontales x 6 verticales).

Para ello, debemos especificar que queremos repetir las texturas con las siguientes llamadas a función:

```
glTexParameterfv(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_REPEAT);
glTexParameterfv(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_REPEAT);
```

La primera permite que las texturas se repitan en horizontal, y la segunda, en vertical.

Con este cambio, podríamos mapear las texturas con coordenadas mayores a 1, de manera que las texturas se repetirían, de esta manera:

```
glBindTexture(GL_TEXTURE_2D,texture_floor);
glBegin(GL_QUADS);
   glTexCoord2d(0.0,0.0);
   glVertex3f(-6.0,0.0,-6.0);
   glTexCoord2d(0.0,6.0);
   glVertex3f(-6.0,0.0,6.0);
   glTexCoord2d(6.0,6.0);
   glVertex3f(6.0,0.0,6.0);
   glTexCoord2d(6.0,0.0);
   glVertex3f(6.0,0.0,-6.0);
glEnd();

glBindTexture(GL_TEXTURE_2D,texture_wall);
glBegin(GL_QUADS);
   glTexCoord2d(0.0,0.0);
   glVertex3f(-6.0,4.0,-6.0);
   glTexCoord2d(0.0,1.0);
```

```
    glVertex3f(-6.0,0.0,-6.0);
    glTexCoord2d(3.0,1.0);
    glVertex3f(6.0,0.0,-6.0);
    glTexCoord2d(3.0,0.0);
    glVertex3f(6.0,4.0,-6.0);
  glEnd();
```

La textura de la pared ha sido mapeada entre (0,0) y (3,1), para repetirla 3 veces horizontalmente, y la textura del suelo entre (0,0) y (6,6). El resultado es sensiblemente distinto:

Figura 6.6. Salida del programa "texture-yes-clamp"

# Capítulo 7: Interacción básica con GLUT

*"¡Esto es to..., esto es to..., esto es todo amigos!" (Porky)*

Como último apartado, vamos a comentar de manera breve cómo interactuar con las ventanas utilizando GLUT.

Ya se ha comentado anteriormente que GLUT trabaja generalmente a través de funciones CALLBACK.

En la mayor parte de los casos, tendremos que registrar en GLUT a qué función queremos que se llame cuando se produzca cierto evento sobre la ventana (pulsación de una tecla, movimiento del ratón...).

## 7.1 Interacción con el teclado

Para mostrar esto vamos a realizar un ejemplo, consistente en modificar el programa "lit-sphere" de manera que seamos nosotros quienes controlemos el movimiento de la luz. Esta estará quiera, y cuando toquemos las teclas derecha/izquierda se moverá, rotando alrededor del centro de la esfera, alrededor del eje Y, y cuando pulsemos arriba/abajo, lo hará alrededor del eje X.

La función que nos permite registrar este evento es glutSpecialFunc, que se encarga de manejar la pulsación de teclas "especiales", como los cursores.

La declaración de esta función es:

```
glutSpecialFunc(void (*func)(int key, int x, int y));
```

Es decir, recibe un puntero a función, que no devuelve nada, y recibe tres enteros: la tecla pulsada, y la posción X e Y del ratón en la pantalla.

La rotación alrededor de los ejes X e Y la vamos a controlar mediante dos variables globales (rot_angle_x, rot_angle_y).

El código de la función a la que hay que llamar al pulsar alguna tecla “especial” será el siguiente:

```
void specialKeys(int key, int x, int y)
{
  switch (key)
  {
    case GLUT_KEY_UP: rot_angle_x--;
         break;
    case GLUT_KEY_DOWN: rot_angle_x++;
         break;
    case GLUT_KEY_RIGHT: rot_angle_y++;
         break;
    case GLUT_KEY_LEFT: rot_angle_y--;
         break;
  }
  glutPostRedisplay();
}
```

Con esto, modificaremos la rotación según la tecla pulsada.

Además, para facilitar la salida del programa, vamos a añadir que, al pulsar la tecla “Esc.”, el programa termine. Para ello utilizaremos la función glutKeyboardFunc(), que recibirá como parámetro un puntero a la siguiente función:

```
static void keys(unsigned char key, int x, int y)
{
  switch (key)
  {
    case 27: exit(0);
         break;
  }
  glutPostRedisplay();
}
```

El resto del programa es prácticamente idéndico al comentado en el capítulo de iluminación (la función idle se hace innecesaria, y hay que dar de alta los dos nuevos manejadores de eventos añadiendo al programa principal:

```
glutSpecialFunc(specialKeys);
glutKeyboardFunc(keys);
```

## 7.2 Redimensionado de la ventana

Finalmente, comentaremos que es posible redimensionar la ventana y su contenido mediante la función de CALLBACK glutReshapeFunc(), a la que se llama cada vez que la ventana cambia de tamaño, y que recibe el nuevo tamaño de la ventana.

En caso de que esto ocurra, tendremos que volver a ajustar la perspectiva (la relación ancho/alto) y establecer el tamaño de la nueva ventana de

visualización:

```
void reshape(int width, int height)
{
  GLfloat h = (GLfloat) height / (GLfloat) width;

  glViewport(0, 0, (GLint) width, (GLint) height);

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(60.0,1.0,1.0,100.0);

  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  glTranslatef(0.0,0.0,-5.0);

  glutPostRedisplay();
}
```

glViewport() nos permitirá establecer qué porción de la ventana será el área visualizable. En este caso, establecemos que el área visualizable será toda la ventana con su nuevo tamaño.

Además, vamos a hacer que, al pulsar la tecla "f" pasemos a pantalla completa/modo ventana de manera alternativa.

Para ello, añadiremos una variable global que indique el modo actual de la pantalla, y el tratamiento para la pulsación de la tecla "f", que será:

```
case 'f': if (windowed == 1){glutFullScreen(); windowed = 0;}
          else{
                  glutPositionWindow ( 20 , 20 );
                  glutReshapeWindow ( 350,350);
                  windowed = 0;
          }
          break;
```

Si estamos en modo ventana, se pasa a pantalla completa. En caso contrario, se reposiciona y reforma la ventana, para pasar de nuevo a modo ventana.

# Capítulo 8: Recursos de interés

*"Por fin... se acabó..." (Bardok)*

En este último epígrafe, me gustaría comentar algunos recursos (libros y páginas web) que considero de gran interés:

- El libro rojo de OpenGL (The Red Book): es un libro básico para cualquier persona que quiera empezar a programar con OpenGL. También se conoce como "la biblia de OpenGL".
- "The OpenGL Utility Toolkit (GLUT) Programming Interface": contiene todo lo necesario para la utilización de la librería GLUT.
- Los tutoriales de "nehe" (nehe.gamedev.net/): estos tutoriales tienen una gran cantidad de información destinada, en gran parte, a la programación de juegos.

Muchas gracias por tu interés en este curso. Para cualquier duda o sugerencia, no dudes en contactar conmigo en "shadow@bardok.net".

# Anexo A: Código de los programas de ejemplo

*"¡Ja ja ja! ¡Con este libro, tengo el poder de controlar hasta los árboles, y la propia ordenación de todos los elementos...!" (Bardok, en 1º de carrera, con el libro de EDA en la mano...)*

## A.a myfirstopenglprogram.c

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

void display(void)
{
   glClearColor(0.0,0.0,0.0,0.0);
   glClear(GL_COLOR_BUFFER_BIT);

   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();

   glOrtho(-1.0,1.0,-1.0,1.0,-1.0,1.0);

   glMatrixMode(GL_MODELVIEW);

   glBegin(GL_TRIANGLES);
      glColor3f(1.0,0.0,0.0);
      glVertex3f(0.0,0.8,0.0);
      glColor3f(0.0,1.0,0.0);
      glVertex3f(-0.6,-0.2,0.0);
      glColor3f(0.0,0.0,1.0);
      glVertex3f(0.6,-0.2,0.0);
   glEnd();

   glFlush();
```

```
   sleep(10);
   exit(0);
}

int main(int argc, char ** argv)
{
   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGBA);
   glutInitWindowPosition(20,20);
   glutInitWindowSize(500,500);
   glutCreateWindow(argv[0]);

   glutDisplayFunc(display);
   glutMainLoop();

   return 0;
}
```

## A.b quadortho.c

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

void display(void)
{
   glClearColor(0.0,0.0,0.0,0.0);
   glClear(GL_COLOR_BUFFER_BIT);

   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();

   glOrtho(-1.0,1.0,-1.0,1.0,-1.0,1.0);

   glMatrixMode(GL_MODELVIEW);

      glBegin(GL_QUADS);
      // Dibujamos un cuadrado
         glColor3f(0.0,1.0,1.0);
         // Color para el cuadrado
         glVertex3f(-0.5,0.5,-0.5);
         // Coordenadas del primer vértice (superior-izquierda)
         glVertex3f(-0.5,-0.5,0.5);
         // Coordenadas del segundo vértice (inferior-izquierda)
         glVertex3f(0.5,-0.5,0.5);
         // Coordenadas del primer vértice (inferior-derecha)
         glVertex3f(0.5,0.5,-0.5);
         // Coordenadas del primer vértice (superior-derecha)
      glEnd();
      // Terminamos de dibujar

   glFlush();

   sleep(10);
   exit(0);
}

int main(int argc, char ** argv)
{
   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGBA);
   glutInitWindowPosition(20,20);
   glutInitWindowSize(500,500);
   glutCreateWindow(argv[0]);

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

```
  return 0;
}
```

## A.c quadpersp.c

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

void display(void)
{
  glClearColor(0.0,0.0,0.0,0.0);
  glClear(GL_COLOR_BUFFER_BIT);

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();

  gluPerspective(60.0,1.0,1.0,100.0);
  // Proyección perspectiva. El ángulo de visualización es de 60
grados, la razón ancho/alto es 1 (son inguales), la distancia
mínima es z=1.0, y la distancia máxima es z=100.0

  glMatrixMode(GL_MODELVIEW);

  glTranslatef(0.0,0.0,-2.0);

    glBegin(GL_QUADS);
    // Dibujamos un cuadrado
      glColor3f(0.0,1.0,1.0);
      // Color para el cuadrado
      glVertex3f(-0.5,0.5,-0.5);
      // Coordenadas del primer vértice (superior-izquierda)
      glVertex3f(-0.5,-0.5,0.5);
      // Coordenadas del segundo vértice (inferior-izquierda)
      glVertex3f(0.5,-0.5,0.5);
      // Coordenadas del primer vértice (inferior-derecha)
      glVertex3f(0.5,0.5,-0.5);
      // Coordenadas del primer vértice (superior-derecha)
    glEnd();
    // Terminamos de dibujar

  glFlush();

  sleep(20);
  exit(0);
}

int main(int argc, char ** argv)
{
  glutInitDisplayMode(GLUT_SINGLE | GLUT_RGBA);
  glutInitWindowPosition(20,20);
  glutInitWindowSize(500,500);
  glutCreateWindow(argv[0]);

  glutDisplayFunc(display);
  glutMainLoop();

  return 0;
}
```

## *A.d zbuffer-yes.c*

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

void display(void)
{
  glDepthFunc(GL_LEQUAL);
  glEnable(GL_DEPTH_TEST);
  // Activamos el  el Z-Buffer
  glClearColor(0.0,0.0,0.0,0.0);
  glClearDepth(1.0);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  // Borramos el buffer de color y el Z-Buffer

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();

  gluPerspective(60.0,1.0,1.0,100.0);
  // Proyección perspectiva. El ángulo de visualización es de 60
grados, la razón ancho/alto es 1 (son inguales), la distancia
mínima es z=1.0, y la distancia máxima es z=100.0

  glMatrixMode(GL_MODELVIEW);

  glTranslatef(0.0,0.0,-2.0);

  glBegin(GL_QUADS);
  // Dibujamos un cuadrado
    glColor3f(0.0,1.0,1.0);
    // Color para el cuadrado
    glVertex3f(-0.5,0.5,-0.5);
    // Coordenadas del primer vértice (superior-izquierda)
    glVertex3f(-0.5,-0.5,0.5);
    // Coordenadas del segundo vértice (inferior-izquierda)
    glVertex3f(0.5,-0.5,0.5);
    // Coordenadas del primer vértice (inferior-derecha)
    glVertex3f(0.5,0.5,-0.5);
    // Coordenadas del primer vértice (superior-derecha)
   glEnd();
  // Terminamos de dibujar

  glBegin(GL_TRIANGLES);
    glColor3f(1.0,0.0,0.0);
    glVertex3f(0.0,0.5,0.0);
    glVertex3f(-0.7,-0.5,0.0);
    glVertex3f(0.7,-0.5,0.0);
  glEnd();

  glFlush();

  sleep(20);
  exit(0);
}

int main(int argc, char ** argv)
{
  glutInitDisplayMode(GLUT_SINGLE | GLUT_RGBA | GLUT_DEPTH);
  glutInitWindowPosition(20,20);
  glutInitWindowSize(500,500);
  glutCreateWindow(argv[0]);

  glutDisplayFunc(display);
  glutMainLoop();

  return 0;
```

```
}
```

## A.e simpleguy.c

```
#include <GL/glut.h>
#include <unistd.h>

// Medidas del cuerpo

#define  BODY_HEIGHT                     4.0
#define  BODY_WIDTH             2.5
#define  BODY_LENGTH                     1.0

#define  ARM_HEIGHT             3.5
#define  ARM_WIDTH                       1.0
#define  ARM_LENGTH             1.0

#define  LEG_HEIGHT             4.5
#define  LEG_WIDTH                       1.0
#define  LEG_LENGTH             1.0

#define  HEAD_RADIUS                     1.1

void display(void)
{
  glDepthFunc(GL_LEQUAL);
  glEnable(GL_DEPTH_TEST);
  // Activamos el  el Z-Buffer
  glClearColor(0.0,0.0,0.0,0.0);
  glClearDepth(1.0);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  // Borramos el buffer de color y el Z-Buffer

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();

  gluPerspective(60.0,1.0,1.0,100.0);
  // Proyección perspectiva. El ángulo de visualización es de 60
grados, la razón ancho/alto es 1 (son inguales), la distancia
mínima es z=1.0, y la distancia máxima es z=100.0

  glMatrixMode(GL_MODELVIEW);

  glTranslatef(0.0,0.0,-16.0);

  // Dibujamos el cuerpo
  glTranslatef(0,BODY_HEIGHT/2,0);
  glPushMatrix();
    glScalef(BODY_WIDTH,BODY_HEIGHT,BODY_LENGTH);
    glColor3f(0.0,0.3,0.8);
    glutSolidCube(1);
  glPopMatrix();

  // Dibujamos el brazo derecho
  glPushMatrix();
    glTranslatef(-(BODY_WIDTH)/2,(BODY_HEIGHT-ARM_HEIGHT)/2,0);
    glTranslatef(0,ARM_HEIGHT/2,0);
    glRotatef(-30,0,0,1);
    glTranslatef(0,-ARM_HEIGHT/2,0);
    glPushMatrix();
      glScalef(ARM_WIDTH,ARM_HEIGHT,ARM_LENGTH);
      glutSolidCube(1);
    glPopMatrix();
```

```
        // ya tenemos el brazo... la mano
        glTranslatef(0,-(ARM_HEIGHT+ARM_WIDTH)/2,0);
        glColor3f(1,0.6,0.6);
        glScalef(ARM_WIDTH,ARM_WIDTH,ARM_LENGTH);
        glutSolidCube(1);
    glPopMatrix();

    // Dibujamos el brazo izquierdo
    glColor3f(0.0,0.3,0.8);
    glPushMatrix();
        glTranslatef((BODY_WIDTH)/2,(BODY_HEIGHT-ARM_HEIGHT)/2,0);
        glTranslatef(0,ARM_HEIGHT/2,0);
        glRotatef(30,0,0,1);
        glTranslatef(0,-ARM_HEIGHT/2,0);
        glPushMatrix();
            glScalef(ARM_WIDTH,ARM_HEIGHT,ARM_LENGTH);
            glutSolidCube(1);
        glPopMatrix();
        // ya tenemos el brazo... la mano
        glTranslatef(0,-(ARM_HEIGHT+ARM_WIDTH)/2,0);
        glColor3f(1,0.6,0.6);
        glScalef(ARM_WIDTH,ARM_WIDTH,ARM_LENGTH);
        glutSolidCube(1);
    glPopMatrix();

    //Dibujamos la pierna derecha
    glColor3f(0.0,0.3,0.8);
    glPushMatrix();
        glTranslatef(-(BODY_WIDTH-LEG_WIDTH)/2,-
(BODY_HEIGHT+LEG_HEIGHT)/2,0);
        glPushMatrix();
            glScalef(LEG_WIDTH,LEG_HEIGHT,LEG_LENGTH);
            glutSolidCube(1);
        glPopMatrix();
        // ahora el píe
        glTranslatef(0,-(LEG_HEIGHT+LEG_WIDTH)/2,LEG_LENGTH);
        glColor3f(0.3,0.4,0.4);
        glScalef(LEG_WIDTH,LEG_WIDTH,LEG_LENGTH*2);
        glutSolidCube(1);
    glPopMatrix();

    //Dibujamos la pierna izquierda
    glColor3f(0.0,0.3,0.8);
    glPushMatrix();
        glTranslatef((BODY_WIDTH-LEG_WIDTH)/2,-
(BODY_HEIGHT+LEG_HEIGHT)/2,0);
        glPushMatrix();
            glScalef(LEG_WIDTH,LEG_HEIGHT,LEG_LENGTH);
            glutSolidCube(1);
        glPopMatrix();
        // ahora el píe
        glTranslatef(0,-(LEG_HEIGHT+LEG_WIDTH)/2,LEG_LENGTH);
        glColor3f(0.3,0.4,0.4);
        glScalef(LEG_WIDTH,LEG_WIDTH,LEG_LENGTH*2);
        glutSolidCube(1);
    glPopMatrix();

    // Dibujamos la cabeza
    glColor3f(1,0.6,0.6);
    glPushMatrix();
        glTranslatef(0,BODY_HEIGHT/2 + HEAD_RADIUS*3/4,0);
        glutSolidSphere(HEAD_RADIUS,10,10);
    glPopMatrix();

    glFlush();
```

```
    sleep(20);
    exit(0);
}

int main(int argc, char ** argv)
{
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGBA | GLUT_DEPTH);
    glutInitWindowPosition(20,20);
    glutInitWindowSize(500,500);
    glutCreateWindow(argv[0]);

    glutDisplayFunc(display);
    glutMainLoop();

    return 0;
}
```

## A.f jerarquia-cuadro.c

```
#include "GL/glut.h"

void dibujarCuadro(float tam)
{
    glBegin(GL_QUADS);
        glVertex3f(-tam/2.0,tam/2.0,0.0);
        glVertex3f(-tam/2.0,-tam/2.0,0.0);
        glVertex3f(tam/2.0,-tam/2.0,0.0);
        glVertex3f(tam/2.0,tam/2.0,0.0);
    glEnd();
}

void display()
{
    glClearColor(0,0,0,0);
    glClear(GL_COLOR_BUFFER_BIT);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.0,1.0,1.0,100.0);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0,0.0,-6.0);

    glColor4f(1.0,0.0,0.0,1.0);
    dibujarCuadro(1.0);

    glPushMatrix();
        glTranslatef(0.0, 2.0, 0.0);
        dibujarCuadro(0.5);
    glPopMatrix();

    glPushMatrix();
        glTranslatef(0.0, -2.0, 0.0);
        dibujarCuadro(0.5);
    glPopMatrix();

    glPushMatrix();
        glTranslatef(2.0, 0.0, 0.0);
        dibujarCuadro(0.5);
    glPopMatrix();
```

```
  glPushMatrix();
    glTranslatef(-2.0, 0.0, 0.0);
    dibujarCuadro(0.5);
  glPopMatrix();

  glFlush();
}

int main(int argc, char **argv)
{
  glutInitDisplayMode(GLUT_SINGLE | GLUT_RGBA);
  glutInitWindowPosition(20,20);
  glutInitWindowSize(500,500);
  glutCreateWindow(argv[0]);

  glutDisplayFunc(display);
  glutMainLoop();

  return 0;
}
```

## *A.g jerarquia-cuadro-recursivo.c*

```
#include "GL/glut.h"

void dibujarCuadro(float tam)
{
  glBegin(GL_QUADS);
    glVertex3f(-tam/2.0,tam/2.0,0.0);
    glVertex3f(-tam/2.0,-tam/2.0,0.0);
    glVertex3f(tam/2.0,-tam/2.0,0.0);
    glVertex3f(tam/2.0,tam/2.0,0.0);
  glEnd();
}

void dibujarCuadroRecursivo(float tam)
{
  if (tam > 0.005)
  {
    dibujarCuadro(tam);

    glPushMatrix();
      glTranslatef(0.0, tam * 2.0, 0.0);
      dibujarCuadroRecursivo(tam / 2.0);
    glPopMatrix();

    glPushMatrix();
      glTranslatef(0.0, -tam * 2.0, 0.0);
      dibujarCuadroRecursivo(tam / 2.0);
    glPopMatrix();

    glPushMatrix();
      glTranslatef(tam * 2.0, 0.0, 0.0);
      dibujarCuadroRecursivo(tam / 2.0);
    glPopMatrix();

    glPushMatrix();
      glTranslatef(-tam * 2.0, 0.0, 0.0);
      dibujarCuadroRecursivo(tam / 2.0);
    glPopMatrix();
  }
}
```

```
void display()
{
  glClearColor(0,0,0,0);
  glClear(GL_COLOR_BUFFER_BIT);

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(60.0,1.0,1.0,100.0);

  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  glTranslatef(0.0,0.0,-6.0);

  glColor4f(1.0,0.0,0.0,1.0);
  dibujarCuadroRecursivo(1.0);

  glFlush();
}

int main(int argc, char **argv)
{
  glutInitDisplayMode(GLUT_SINGLE | GLUT_RGBA);
  glutInitWindowPosition(20,20);
  glutInitWindowSize(500,500);
  glutCreateWindow(argv[0]);

  glutDisplayFunc(display);
  glutMainLoop();

  return 0;
}
```

## A.h sphere-rebotes.cpp

```
/////////////////////////////////////////////////////////

// Fichero sphere-rebotes.cpp

/////////////////////////////////////////////////////////

#include <GL/glut.h>
#include <unistd.h>
#include "sphere.h"

float * pos;

TSphere * sphere = NULL;

void initgl()
{
  glEnable(GL_DEPTH_TEST);

  glClearColor(0.0,0.0,0.0,0.0);

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();

  gluPerspective(60.0,1.0,1.0,100.0);
  // Proyección perspectiva. El ángulo de visualización es de 60
grados, la razón ancho/alto es 1 (son inguales), la distancia
mínima es z=1.0, y la distancia máxima es z=100.0

  sphere = new TSphere(5,0.1);
```

```
  glMatrixMode(GL_MODELVIEW);
  gluLookAt(3,3,14,0,0,0,0,1,0);
}

void display(void)
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glColor3f(1.0,1.0,1.0);
  glutWireCube(10);
  glPushMatrix();
    glColor3f(0.0,0.0,1.0);
    pos = sphere->getPosv();
    glTranslatef(pos[0],pos[1],pos[2]);
    glutSolidSphere(1,10,10);
  glPopMatrix();
  glFlush();
}

void idle(void)
{
  sphere->test();
  usleep(33);
  glutPostRedisplay();
}

int main(int argc, char ** argv)
{
  glutInitDisplayMode(GLUT_SINGLE | GLUT_RGBA | GLUT_DEPTH);
  glutInitWindowPosition(20,20);
  glutInitWindowSize(500,500);
  glutCreateWindow(argv[0]);

  initgl();
  glutDisplayFunc(display);
  glutIdleFunc(idle);
  glutMainLoop();

  return 0;
}


////////////////////////////////////////////////////////

// Fichero sphere.h

////////////////////////////////////////////////////////

#ifndef sphere_h
#define sphere_h

#include <stdlib.h>
#include <time.h>
#include <math.h>

typedef class TSphere
{
  private:
    float maxpos;
    float pos[3];
    float dir[3];
    float speed;
  public:
    TSphere(float maxpos, float speed);
    void test();
    void modifySpeed(float inc);
```

```
    float * getPosv();
} TSphere;

#endif

/////////////////////////////////////////////////////////

// Fichero sphere.cpp

/////////////////////////////////////////////////////////

#include "sphere.h"

int pass=0;

TSphere::TSphere(float maxpos, float speed)
{
  this->maxpos = maxpos;

  if (!pass)
  {
    srandom(time(NULL));
    pass = 1;
  }

  pos[0] = (random() % (int)maxpos) - maxpos/2;
  pos[1] = (random() % (int)maxpos) - maxpos/2;
  pos[2] = (random() % (int)maxpos) - maxpos/2;

  dir[0] = random();
  dir[1] = random();
  dir[2] = random();

  float dirmod = sqrt(dir[0]*dir[0] + dir[1]*dir[1] + dir[2]*dir
[2]);

  dir[0] /= dirmod;
  dir[1] /= dirmod;
  dir[2] /= dirmod;

  dir[0] *= speed;
  dir[1] *= speed;
  dir[2] *= speed;
}

void TSphere::test()
{
  ((pos[0] < -maxpos) || (pos[0] > maxpos))?dir[0]*=-1:0;
  ((pos[1] < -maxpos) || (pos[1] > maxpos))?dir[1]*=-1:0;
  ((pos[2] < -maxpos) || (pos[2] > maxpos))?dir[2]*=-1:0;

  pos[0] += dir[0];
  pos[1] += dir[1];
  pos[2] += dir[2];
}

void TSphere::modifySpeed(float inc)
{
  float factor = (speed+inc)/speed;
  speed += inc;
  dir[0] *= factor;
  dir[1] *= factor;
  dir[2] *= factor;
}
```

```
float * TSphere::getPosv()
{
  return pos;
}
```

## A.i sphere-rebotes-multi.cpp

```
#include <GL/glut.h>
#include <unistd.h>
#include "sphere.h"
#include <math.h>

#define  NUM_SPHERES                 50
#define  SPHERE_RADIUS               0.3

float * pos;
TSphere * sphere[NUM_SPHERES];

void initgl()
{
  glEnable(GL_DEPTH_TEST);
  glClearColor(0.0,0.0,0.0,0.0);

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(60.0,1.0,1.0,100.0);

  for (int i=0;i<NUM_SPHERES;i++)
    sphere[i] = new TSphere(5,((random() % 10)/(float)20)+0.1);

  glMatrixMode(GL_MODELVIEW);
  gluLookAt(3,3,14,0,0,0,0,1,0);
}

void display(void)
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glColor3f(1.0,1.0,1.0);
  glutWireCube(10);
  for (int i=0;i<NUM_SPHERES;i++)
  {
    glPushMatrix();
      glColor3f(0.0,0.0,1.0);
      pos = sphere[i]->getPosv();
      glTranslatef(pos[0],pos[1],pos[2]);
      glutSolidSphere(SPHERE_RADIUS,10,10);
    glPopMatrix();
    sphere[i]->test();
  }
  glFlush();
  glutSwapBuffers();
}

void idle(void)
{
  usleep(33);
  glutPostRedisplay();
}

int main(int argc, char ** argv)
{
  glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);
  glutInitWindowPosition(20,20);
```

```
   glutInitWindowSize(500,500);
   glutCreateWindow(argv[0]);

   initgl();
   glutDisplayFunc(display);
   glutIdleFunc(idle);
   glutMainLoop();

   return 0;
}
```

## A.j normals-perp.c

```
#include <GL/glut.h>
#include <stdlib.h>

//#define   SINGLE_NORMAL

void display(void)
{
   GLfloat mat_color [] = {0.0,1.0,1.0,1.0};
   GLfloat light_color [] = {1.0,1.0,1.0,1.0};
   GLfloat light_ambient [] = {0.0,0.0,0.0,1.0};
   GLfloat normal [] = {0.0,1.0,0.0};
   GLfloat light_dir [] = {0.0,1.0,0.0,0.0};

   glClearColor(0.0,0.0,0.0,0.0);
   glClear(GL_COLOR_BUFFER_BIT);

   glEnable(GL_LIGHTING);

   glLightfv(GL_LIGHT0,GL_AMBIENT,light_ambient);
   glLightfv(GL_LIGHT0,GL_DIFFUSE,light_color);
   glLightfv(GL_LIGHT0,GL_SPECULAR,light_color);
   glEnable(GL_LIGHT0);

   glMaterialfv(GL_FRONT,GL_AMBIENT_AND_DIFFUSE,mat_color);

   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluPerspective(60.0,1.0,1.0,100.0);

   glMatrixMode(GL_MODELVIEW);
   glTranslatef(-0.3,-0.6,-4.0);

   glLightfv(GL_LIGHT0,GL_POSITION,light_dir);

   glBegin(GL_QUADS);

#ifdef SINGLE_NORMAL
      glNormal3fv(normal);
#else
      glNormal3f(1.0,1.0,-1.0);
#endif
      glVertex3f(-1.0,0.0,-1.0);

      glNormal3fv(normal);
      glVertex3f(-1.0,0.0,1.0);

#ifdef SINGLE_NORMAL
      glNormal3fv(normal);
#else
      glNormal3f(-1.0,1.0,-1.0);
```

```
#endif
    glVertex3f(1.0,0.0,1.0);

    glNormal3fv(normal);
    glVertex3f(1.0,0.0,-1.0);
  glEnd();

  glFlush();

  sleep(20);
  exit(0);
}

int main(int argc, char ** argv)
{
  glutInitDisplayMode(GLUT_SINGLE | GLUT_RGBA);
  glutInitWindowPosition(20,20);
  glutInitWindowSize(500,500);
  glutCreateWindow(argv[0]);

  glutDisplayFunc(display);
  glutMainLoop();

  return 0;
}
```

## *A.k lit-sphere.c*

```
#include <GL/glut.h>
#include <unistd.h>

#define   ROT_INC              1.0

float light_ambient [] = {0.0,0.2,0.0,1.0};
float light_diffuse_specular [] = {0.8,0.8,0.8,1.0};
float light_pos [] = {0.0,0.0,2.0,1.0};
float spot_dir [] = {0.0,0.0,-1.0};
float spot_cutoff = 30.0;
float spot_exponent = 1.0;

float mat_ambient_diffuse [] = {0.0,0.8,1.0,1.0};
float mat_specular [] = {0.7,0.0,0.0,1.0};
float mat_emission [] = {0.0,0.0,0.0,1.0};
float mat_shininess = 0.4;

float focus_emission [] = {0.8,0.8,0.8,1.0};

float rot_angle_y = 0.0;
float rot_angle_x = 0.0;


void initgl()
{
  glEnable(GL_DEPTH_TEST);
  glClearColor(0.0,0.0,0.0,0.0);

  glLightModeli(GL_LIGHT_MODEL_TWO_SIDE,GL_FALSE);
  glEnable(GL_LIGHTING);
  glLightfv(GL_LIGHT0,GL_AMBIENT,light_ambient);
  glLightfv(GL_LIGHT0,GL_DIFFUSE,light_diffuse_specular);
  glLightfv(GL_LIGHT0,GL_SPECULAR,light_diffuse_specular);
  glLightf(GL_LIGHT0,GL_SPOT_CUTOFF,spot_cutoff);
  glLightf(GL_LIGHT0,GL_SPOT_EXPONENT,spot_exponent);
```

```
  glEnable(GL_LIGHT0);

  glMaterialfv
(GL_FRONT,GL_AMBIENT_AND_DIFFUSE,mat_ambient_diffuse);
  glMaterialfv(GL_FRONT,GL_SPECULAR,mat_specular);
  glMaterialf(GL_FRONT,GL_SHININESS,mat_shininess);

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(60.0,1.0,1.0,100.0);

  glMatrixMode(GL_MODELVIEW);
  glTranslatef(0.0,0.0,-5.0);
}

void idle(void)
{
  rot_angle_y = (rot_angle_y > 360.0)?0:rot_angle_y + ROT_INC;
  rot_angle_x = (rot_angle_x > 360.0)?0:rot_angle_x + ROT_INC/
(2*3.1416);
  glutPostRedisplay();
}

void display(void)
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  glPushMatrix();
    glRotatef(30.0,0.0,0.0,1.0);
    glRotatef(rot_angle_y,0.0,1.0,0.0);
    glRotatef(rot_angle_x,1.0,0.0,0.0);
    glLightfv(GL_LIGHT0,GL_POSITION,light_pos);
    glLightfv(GL_LIGHT0,GL_SPOT_DIRECTION,spot_dir);
      glTranslatef(light_pos[0],light_pos[1],light_pos[2]);
      glColorMaterial(GL_FRONT,GL_EMISSION);
      glEnable(GL_COLOR_MATERIAL);
        glColor4fv(focus_emission);
        glutSolidCone(0.2,0.5,7,7);
        glColor4fv(mat_emission);
      glDisable(GL_COLOR_MATERIAL);
  glPopMatrix();
  glutSolidSphere(1.0,20,20);
  glFlush();
  glutSwapBuffers();
}

int main(int argc, char ** argv)
{
  glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);
  glutInitWindowPosition(20,20);
  glutInitWindowSize(350,350);
  glutCreateWindow(argv[0]);

  initgl();
  glutDisplayFunc(display);
  glutIdleFunc(idle);
  glutMainLoop();

  return 0;
}
```

## *A.l l-sphere-rebotes-multi.cpp*

```
/////////////////////////////////////////////////////////

// Fichero l-sphere.h

/////////////////////////////////////////////////////////
#ifndef l_sphere_h
#define l_sphere_h

#include <stdlib.h>
#include <time.h>
#include <math.h>

typedef class TSphere
{
  private:
    float maxpos;
    float pos[3];
    float dir[3];
    float color[3];
    float speed;
  public:
    TSphere(float maxpos, float speed);
    void test();
    void modifySpeed(float inc);
    float * getPosv();
    float * getColor();
} TSphere;

#endif

/////////////////////////////////////////////////////////

// Fichero l-sphere.cpp

/////////////////////////////////////////////////////////
#include "l-sphere.h"

int pass=0;

TSphere::TSphere(float maxpos, float speed)
{
  this->maxpos = maxpos;

  if (!pass)
  {
    srandom(time(NULL));
    pass = 1;
  }

  pos[0] = (random() % (int)maxpos) - maxpos/2;
  pos[1] = (random() % (int)maxpos) - maxpos/2;
  pos[2] = (random() % (int)maxpos) - maxpos/2;

  dir[0] = random();
  dir[1] = random();
  dir[2] = random();

  color[0] = (random() % 1001)/1000.0;
  color[1] = (random() % 1001)/1000.0;
  color[2] = (random() % 1001)/1000.0;

  float dirmod = sqrt(dir[0]*dir[0] + dir[1]*dir[1] + dir[2]*dir
[2]);
```

```
    dir[0] /= dirmod;
    dir[1] /= dirmod;
    dir[2] /= dirmod;

    dir[0] *= speed;
    dir[1] *= speed;
    dir[2] *= speed;
}

void TSphere::test()
{
    ((pos[0] < -maxpos) || (pos[0] > maxpos))?dir[0]*=-1:0;
    ((pos[1] < -maxpos) || (pos[1] > maxpos))?dir[1]*=-1:0;
    ((pos[2] < -maxpos) || (pos[2] > maxpos))?dir[2]*=-1:0;

    pos[0] += dir[0];
    pos[1] += dir[1];
    pos[2] += dir[2];
}

void TSphere::modifySpeed(float inc)
{
    float factor = (speed+inc)/speed;
    speed += inc;
    dir[0] *= factor;
    dir[1] *= factor;
    dir[2] *= factor;
}

float * TSphere::getPosv()
{
    return pos;
}

float * TSphere::getColor()
{
    return color;
}

/////////////////////////////////////////////////////////

// Fichero l-sphere-rebotes-multi.cpp

/////////////////////////////////////////////////////////
#include <GL/glut.h>
#include <unistd.h>
#include "l-sphere.h"
#include <math.h>

#define   NUM_SPHERES                    75
#define   SPHERE_RADIUS                  0.3

float light_color [] = {1.0,1.0,1.0,1.0};
float light_pos [] = {1.0,1.0,1.0,0.0};

float * pos;
TSphere * sphere[NUM_SPHERES];

void initgl()
{
    glEnable(GL_DEPTH_TEST);
    glClearColor(0.0,0.0,0.0,0.0);

    glLightfv(GL_LIGHT0,GL_DIFFUSE,light_color);
```

```
  glLightfv(GL_LIGHT0,GL_SPECULAR,light_color);
  glEnable(GL_LIGHT0);

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(60.0,1.0,1.0,100.0);

  for (int i=0;i<NUM_SPHERES;i++)
    sphere[i] = new TSphere(5,((random() % 10)/(float)20)+0.1);

  glMatrixMode(GL_MODELVIEW);
  gluLookAt(3,3,14,0,0,0,0,1,0);
  glLightfv(GL_LIGHT0,GL_POSITION,light_pos);
}

void idle(void)
{
  usleep(33);
  glutPostRedisplay();
}

void display(void)
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glDisable(GL_LIGHTING);
    glColor3f(1.0,1.0,1.0);
    glutWireCube(10);
  glEnable(GL_LIGHTING);
  glColorMaterial(GL_FRONT,GL_AMBIENT_AND_DIFFUSE);
  glEnable(GL_COLOR_MATERIAL);
    for (int i=0;i<NUM_SPHERES;i++)
    {
      glPushMatrix();
      glColor4fv(sphere[i]->getColor());
      pos = sphere[i]->getPosv();
      glTranslatef(pos[0],pos[1],pos[2]);
      glutSolidSphere(SPHERE_RADIUS,10,10);
      glPopMatrix();
      sphere[i]->test();
    }
  glDisable(GL_COLOR_MATERIAL);
  glFlush();
  glutSwapBuffers();
}

int main(int argc, char ** argv)
{
  glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);
  glutInitWindowPosition(20,20);
  glutInitWindowSize(400,400);
  glutCreateWindow(argv[0]);

  initgl();
  glutDisplayFunc(display);
  glutIdleFunc(idle);
  glutMainLoop();

  return 0;
}
```

## *A.m triang-texture.c*

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

```
static const struct {
  unsigned int       width;
  unsigned int       height;
  unsigned int       bytes_per_pixel; /* 3:RGB, 4:RGBA */
  unsigned char      pixel_data[128 * 128 * 3 + 1];
} gimp_image = {
  128, 128, 3,
  ">@2>@2@B4@B4?A3>@2<>0;=/=?1@B4<>0CE7CE7DF8FH:HJ<LJ=OM@OM@NL>LJ
>IG:GE8FE8"
[... image data ...]
  "\6\22\23\16\11\12\5",
};

void display(void)
{
   int texture;

   glGenTextures(1,&texture);
   glBindTexture(GL_TEXTURE_2D,texture);
   glTexEnvf( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE );
   gluBuild2DMipmaps( GL_TEXTURE_2D, gimp_image.bytes_per_pixel,
gimp_image.width, gimp_image.height,GL_RGB, GL_UNSIGNED_BYTE,
gimp_image.pixel_data );
   glEnable(GL_TEXTURE_2D);

   glClearColor(0.0,0.0,0.0,0.0);
   glClear(GL_COLOR_BUFFER_BIT);

   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();

   gluPerspective(60.0,1.0,1.0,100.0);

   glMatrixMode(GL_MODELVIEW);

   glTranslatef(0.0,0.0,-2.0);

      glBegin(GL_TRIANGLES);
         glTexCoord2d(0.0,1.0);
         glVertex3f(-0.5,-0.5,0.5);
         glTexCoord2d(1.0,1.0);
         glVertex3f(0.5,-0.5,0.5);
         glTexCoord2d(0.5,0.0);
         glVertex3f(0.0,0.5,0.5);
      glEnd();
   glFlush();

   sleep(20);
   exit(0);
}

int main(int argc, char ** argv)
{
   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGBA);
   glutInitWindowPosition(20,20);
   glutInitWindowSize(300,300);
   glutCreateWindow(argv[0]);

  glutDisplayFunc(display);
   glutMainLoop();

   return 0;
}
```

## A.n normals-perp-texture.c

```
#include <GL/glut.h>
#include <stdlib.h>

static const struct {
  unsigned int       width;
  unsigned int       height;
  unsigned int       bytes_per_pixel; /* 3:RGB, 4:RGBA */
  unsigned char      pixel_data[128 * 128 * 3 + 1];
} gimp_image = {
  128, 128, 3,
  ">@2>@2@B4@B4?A3>@2<>0;=/=?1@B4<>0CE7CE7DF8FH:HJ<LJ=OM@OM@NL>LJ
>IG:GE8FE8"
[... image data ...]
  "\6\22\23\16\11\12\5",
};

void display(void)
{
   GLfloat mat_color [] = {0.5,1.0,1.0,1.0};
   GLfloat light_color [] = {1.0,1.0,1.0,1.0};
   GLfloat light_ambient [] = {0.2,0.2,0.2,1.0};
   GLfloat normal [] = {0.0,1.0,0.0};
   GLfloat light_dir [] = {0.0,1.0,0.0,0.0};

   int texture;

   glGenTextures(1,&texture);
   glBindTexture(GL_TEXTURE_2D,texture);
   glTexEnvf( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE );
   // Descomentar esta línea y comentar la anterior en el caso
   // de querer utilizar GL_REPLACE
   //glTexEnvf( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE );
   gluBuild2DMipmaps( GL_TEXTURE_2D, gimp_image.bytes_per_pixel,
gimp_image.width, gimp_image.height,GL_RGB, GL_UNSIGNED_BYTE,
gimp_image.pixel_data );
   glEnable(GL_TEXTURE_2D);

glClearColor(0.0,0.0,0.0,0.0);
   glClear(GL_COLOR_BUFFER_BIT);

   glEnable(GL_LIGHTING);

   glLightfv(GL_LIGHT0,GL_AMBIENT,light_ambient);
   glLightfv(GL_LIGHT0,GL_DIFFUSE,light_color);
   glLightfv(GL_LIGHT0,GL_SPECULAR,light_color);
   glEnable(GL_LIGHT0);

   glMaterialfv(GL_FRONT,GL_AMBIENT_AND_DIFFUSE,mat_color);

   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluPerspective(60.0,1.0,1.0,100.0);

   glMatrixMode(GL_MODELVIEW);
   glTranslatef(-0.3,-0.6,-4.0);

   glRotatef(90.0,1.0,0.0,0.0);

   glLightfv(GL_LIGHT0,GL_POSITION,light_dir);

   glBegin(GL_QUADS);

      glNormal3f(1.0,-1.0,1.0);
```

```
    glTexCoord2d(0.0,0.0);
    glVertex3f(-1.5,0.0,-1.5);

    glNormal3fv(normal);
    glTexCoord2d(0.0,1.0);
    glVertex3f(-1.5,0.0,1.5);

    glNormal3f(1.0,-1.0,1.0);
    glTexCoord2d(1.0,1.0);
    glVertex3f(1.5,0.0,1.5);

    glNormal3fv(normal);
    glTexCoord2d(1.0,0.0);
    glVertex3f(1.5,0.0,-1.5);
  glEnd();

  glFlush();

  sleep(20);
  exit(0);
}

int main(int argc, char ** argv)
{
  glutInitDisplayMode(GLUT_SINGLE | GLUT_RGBA);
  glutInitWindowPosition(20,20);
  glutInitWindowSize(350,350);
  glutCreateWindow(argv[0]);

  glutDisplayFunc(display);
  glutMainLoop();

  return 0;
}
```

## A.o texture-no-clamp.c + texture-yes-clamp.c

```
#include <GL/glut.h>
// Los dos siguientes includes tienen las definiciones de las
// imágenes en sendas estructuras de c.
#include "grass_texture.c"
#include "brick_texture.c"

void display(void)
{
  int texture_floor;
  int texture_wall;

  glGenTextures(1,&texture_floor);
  glBindTexture(GL_TEXTURE_2D,texture_floor);
  gluBuild2DMipmaps( GL_TEXTURE_2D,
grass_texture.bytes_per_pixel, grass_texture.width,
grass_texture.height,GL_RGB, GL_UNSIGNED_BYTE,
grass_texture.pixel_data );

  glGenTextures(1,&texture_wall);
  glBindTexture(GL_TEXTURE_2D,texture_wall);
  gluBuild2DMipmaps( GL_TEXTURE_2D,
brick_texture.bytes_per_pixel, brick_texture.width,
brick_texture.height,GL_RGB, GL_UNSIGNED_BYTE,
brick_texture.pixel_data );

  glTexEnvf( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE );
```

```
  // En caso de no querer repetir las texturas, comentar estas
  // dos líneas
  glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_REPEAT);
  glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_REPEAT);

  glEnable(GL_TEXTURE_2D);

  glClearColor(0.1,0.6,1.0,0.0);
  glClear(GL_COLOR_BUFFER_BIT);

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();

  gluPerspective(60.0,1.0,1.0,100.0);

  glMatrixMode(GL_MODELVIEW);

  gluLookAt(-0.0,3.0,6.0,0.0,2.0,0.0,0.0,1.0,0.0);

    // En caso de no querer repetir las texturas, mapearlas
    // entre 0 y 1 (sustituir 6 por 1 en glTexCoord2d).
    glBindTexture(GL_TEXTURE_2D,texture_floor);
    glBegin(GL_QUADS);
      glTexCoord2d(0.0,0.0);
      glVertex3f(-6.0,0.0,-6.0);
      glTexCoord2d(0.0,6.0);
      glVertex3f(-6.0,0.0,6.0);
      glTexCoord2d(6.0,6.0);
      glVertex3f(6.0,0.0,6.0);
      glTexCoord2d(6.0,0.0);
      glVertex3f(6.0,0.0,-6.0);
    glEnd();

    // En caso de no querer repetir las texturas, mapearlas
    // entre 0 y 1 (sustituir 3 por 1 en glTexCoord2d).
    glBindTexture(GL_TEXTURE_2D,texture_wall);
    glBegin(GL_QUADS);
      glTexCoord2d(0.0,0.0);
      glVertex3f(-6.0,4.0,-6.0);
      glTexCoord2d(0.0,1.0);
      glVertex3f(-6.0,0.0,-6.0);
      glTexCoord2d(3.0,1.0);
      glVertex3f(6.0,0.0,-6.0);
      glTexCoord2d(3.0,0.0);
      glVertex3f(6.0,4.0,-6.0);
    glEnd();
  glFlush();

  sleep(20);
  exit(0);
}

int main(int argc, char ** argv)
{
  glutInitDisplayMode(GLUT_SINGLE | GLUT_RGBA);
  glutInitWindowPosition(20,20);
  glutInitWindowSize(300,300);
  glutCreateWindow(argv[0]);

  glutDisplayFunc(display);
  glutMainLoop();

  return 0;
}
```

## A.p lit-sphere-keyb.c

```
#include <GL/glut.h>
#include <unistd.h>
#include <stdio.h>

#define   ROT_INC              1.0

float rot_angle_y = 0.0;
float rot_angle_x = 0.0;

int windowed = 1;

void specialKeys(int key, int x, int y)
{
  switch (key)
  {
    case GLUT_KEY_UP: rot_angle_x--;
         break;
    case GLUT_KEY_DOWN: rot_angle_x++;
         break;
    case GLUT_KEY_RIGHT: rot_angle_y++;
         break;
    case GLUT_KEY_LEFT: rot_angle_y--;
         break;
  }
  glutPostRedisplay();
}

static void keys(unsigned char key, int x, int y)
{
  switch (key)
  {
    case 27: exit(0);
         break;
    case 'f': if (windowed == 1)
                  {glutFullScreen(); windowed = 0;}
           else{
                  glutPositionWindow(20,20);
                  glutReshapeWindow(350,350);
                  windowed = 0;
           }
         break;
  }
  glutPostRedisplay();
}

static void display(void)
{
  float light_pos [] = {0.0,0.0,2.0,1.0};
  float spot_dir [] = {0.0,0.0,-1.0};

  float mat_emission [] = {0.0,0.0,0.0,1.0};
  float focus_emission [] = {0.8,0.8,0.8,1.0};

  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  glPushMatrix();
    glRotatef(rot_angle_y,0.0,1.0,0.0);
    glRotatef(rot_angle_x,1.0,0.0,0.0);
    glLightfv(GL_LIGHT0,GL_POSITION,light_pos);
    glLightfv(GL_LIGHT0,GL_SPOT_DIRECTION,spot_dir);
      glTranslatef(light_pos[0],light_pos[1],light_pos[2]);
      glColorMaterial(GL_FRONT,GL_EMISSION);
      glEnable(GL_COLOR_MATERIAL);
```

```
            glColor4fv(focus_emission);
            glutSolidCone(0.2,0.5,7,7);
            glColor4fv(mat_emission);
          glDisable(GL_COLOR_MATERIAL);
    glPopMatrix();
    glutSolidSphere(1.0,20,20);
    glFlush();
    glutSwapBuffers();
  }

  void glInit()
  {
    float light_ambient [] = {0.0,0.2,0.0,1.0};
    float light_diffuse_specular [] = {0.8,0.8,0.8,1.0};
    float spot_cutoff = 30.0;
    float spot_exponent = 1.0;

    float mat_ambient_diffuse [] = {0.0,0.8,1.0,1.0};
    float mat_specular [] = {0.7,0.0,0.0,1.0};
    float mat_shininess = 0.4;

    glEnable(GL_DEPTH_TEST);
    glClearColor(0.0,0.0,0.0,0.0);

    glLightModeli(GL_LIGHT_MODEL_TWO_SIDE,GL_FALSE);
    glEnable(GL_LIGHTING);
    glLightfv(GL_LIGHT0,GL_AMBIENT,light_ambient);
    glLightfv(GL_LIGHT0,GL_DIFFUSE,light_diffuse_specular);
    glLightfv(GL_LIGHT0,GL_SPECULAR,light_diffuse_specular);
    glLightf(GL_LIGHT0,GL_SPOT_CUTOFF,spot_cutoff);
    glLightf(GL_LIGHT0,GL_SPOT_EXPONENT,spot_exponent);
    glEnable(GL_LIGHT0);

    glMaterialfv
  (GL_FRONT,GL_AMBIENT_AND_DIFFUSE,mat_ambient_diffuse);
    glMaterialfv(GL_FRONT,GL_SPECULAR,mat_specular);
    glMaterialf(GL_FRONT,GL_SHININESS,mat_shininess);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.0,1.0,1.0,100.0);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0,0.0,-5.0);
  }

  void reshape(int width, int height)
  {
    GLfloat h = (GLfloat) height / (GLfloat) width;

    glViewport(0, 0, (GLint) width, (GLint) height);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.0,1.0,1.0,100.0);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0,0.0,-5.0);

    glutPostRedisplay();
  }

  int main(int argc, char ** argv)
```

```
{
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);

  glutInitWindowPosition(20,20);
  glutInitWindowSize(350,350);
  glutCreateWindow(argv[0]);

  glInit();
  glutDisplayFunc(display);
  glutReshapeFunc(reshape);
  glutSpecialFunc(specialKeys);
  glutKeyboardFunc(keys);
  glutMainLoop();

  return 0;
}
```

# Bibliografía

OPENGL: OpenGL - High Performance 2D/3D Graphics, 2003, http://www.opengl.org

REDBOOK: Jackie Naider, Tom Davis and Mason Woo, OpenGL Programming Guide or 'The Red Book'. Addison-Wesley, 1994

GLUT: Mark J. Kilgard, The OpenGL Utility Toolkit (GLUT) Programming Interface, 1996

MESA: Mesa Home Page, 2003, http://www.mesa3d.org

DRI: DRI - Direct Rendering Infraestructure, 2003, http://dri.sourceforge.net/

TEXT: OpenGL Texture Tutorial, 2003, http://www.nullterminator.net/gltexture.html