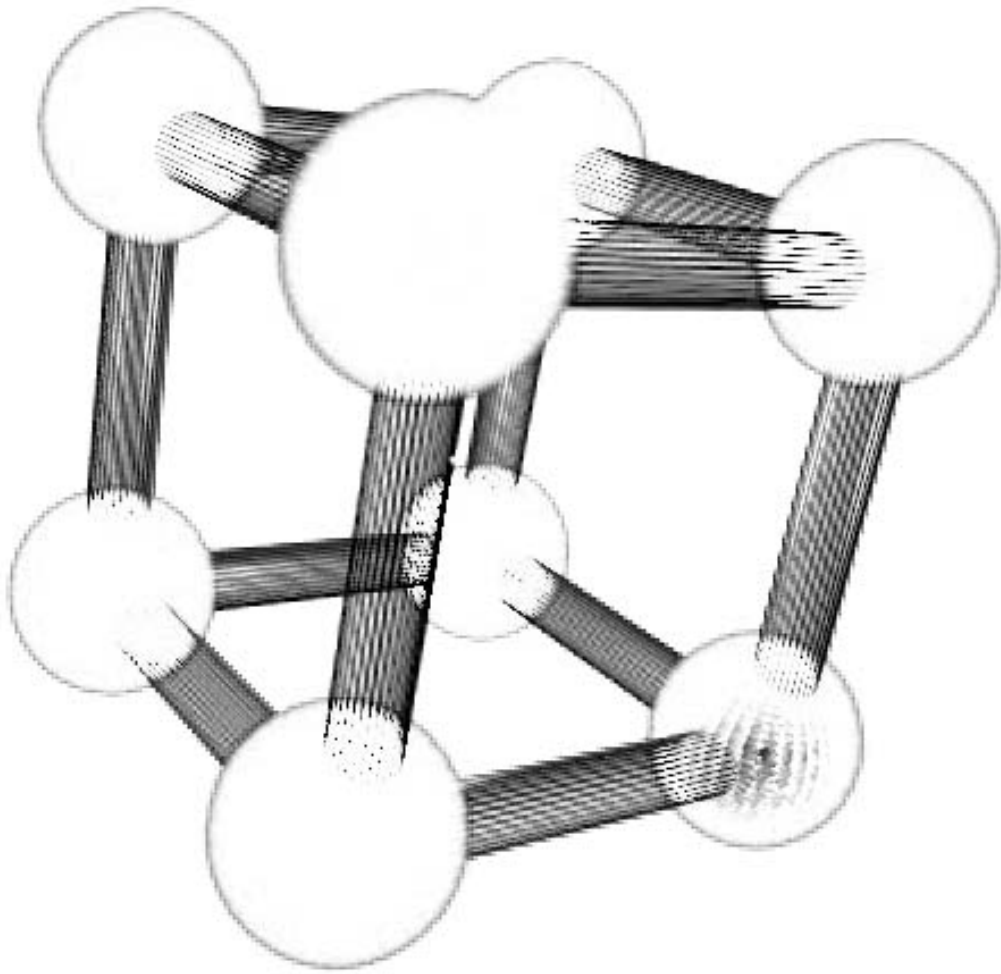


Una aproximacion a OpenGL



Julian Dorado de la Calle
Alberto Jaspe Villanueva

1	<i>Introduccion.....</i>	4
1.1	¿Que es OpenGL?.....	4
1.2	OpenGL como una maquina de estados.....	4
1.3	El Pipeline de renderizado de OpenGL.....	5
1.4	Escribir codigo basado en OpenGL.....	6
1.4.1	Sintaxis	6
1.4.2	Animacion	7
1.4.3	Librerias relacionadas con OpenGL.....	8
1.5	Pretensiones de estos apuntes	9
2	<i>El “Hello World” de OpenGL.....</i>	10
2.1	Requisitos del sistema	10
2.1.1	Hardware	10
2.1.2	Software.....	10
2.2	La OpenGL Utility Toolkit (GLUT).....	10
2.3	“Hello World”	11
2.3.1	Codigo	11
2.3.2	Analisis del codigo	12
3	<i>Dibujando en 3D.....</i>	18
3.1	Definicion de un lienzo en 3D	18
3.2	El punto en 3D: el vertice.....	19
3.3	Las primitivas	20
3.3.1	Dibujo de puntos (GL_POINTS).....	20
3.3.1.1	Ajuste del tamaño del punto	21
3.3.2	Dibujo de líneas (GL_LINES).....	21
3.3.3	Dibujo de polígonos	23
3.3.3.1	Triángulos (GL_TRIANGLES).....	24
3.3.3.2	Cuadrados (GL_QUADS)	26
3.4	Construccion de objetos solidos mediante polígonos	26
3.4.1	Color de relleno	26
3.4.2	Modelo de sombreado	27
3.4.3	Eliminacion de las caras ocultas.....	28
4	<i>Moviendonos por nuestro espacio 3D: transformaciones de coordenadas.....</i>	32
4.1	Coordenadas oculares	32

4.2	Transformaciones	33
4.2.1	El modelador	33
4.2.1.1	Transformaciones del observador.....	33
4.2.1.2	Transformaciones del modelo	34
4.2.1.3	Dualidad del modelador.....	34
4.2.2	Transformaciones de la proyeccion.....	35
4.2.3	Transformaciones de la vista	35
4.3	Matrices	35
4.3.1	El canal de transformaciones.....	35
4.3.2	La matriz del modelador	36
4.3.2.1	Translacion	36
4.3.2.2	Rotacion.....	37
4.3.2.3	Escalado.....	37
4.3.2.4	La matriz identidad.....	38
4.3.2.5	Las pilas de matrices.....	39
4.3.3	La matriz de proyeccion	40
4.3.3.1	Proyecciones ortograficas.....	40
4.3.3.2	Proyecciones perspectivas	41
4.4	Ejemplo: una escena simple.....	44
4.4.1	Codigo	44
4.4.2	Analisis del codigo	48

1 Introduccion

1.1 ¿Que es OpenGL?

OpenGL (ogl en adelante) es la interfaz software de hardware grafico. Es un motor 3D cuyas rutinas estan integradas en tarjetas graficas 3D. Ogl posee todas las caracter sticas necesarias para la representacion mediante computadoras de escenas 3D modeladas con pol gonos, desde el pintado mas basico de triangulos, hasta el mapeado de texturas, iluminacion o NURBS.

La compan a que desarrolla esta librer a es Sillicon Graphics Inc (SGI), en pro de hacer un estandar en la representacion 3D gratuito y con codigo abierto (open source). Esta basado en sus propios OS y lenguajes IRIS, de forma que es perfectamente portable a otros lenguajes. Entre ellos C, C++, etc y las librer as dinamicas permiten usarlo sin problema en Visual Basic, Visual Fortran, Java, etc.

Ogl soporta hardware 3D, y es altamente recomendable poseer este tipo de hardware grafico. Si no se tiene disposicion de el, las rutinas de representacion correran por soft, en vez de hard, decrementando en gran medida su velocidad.

1.2 OpenGL como una maquina de estados

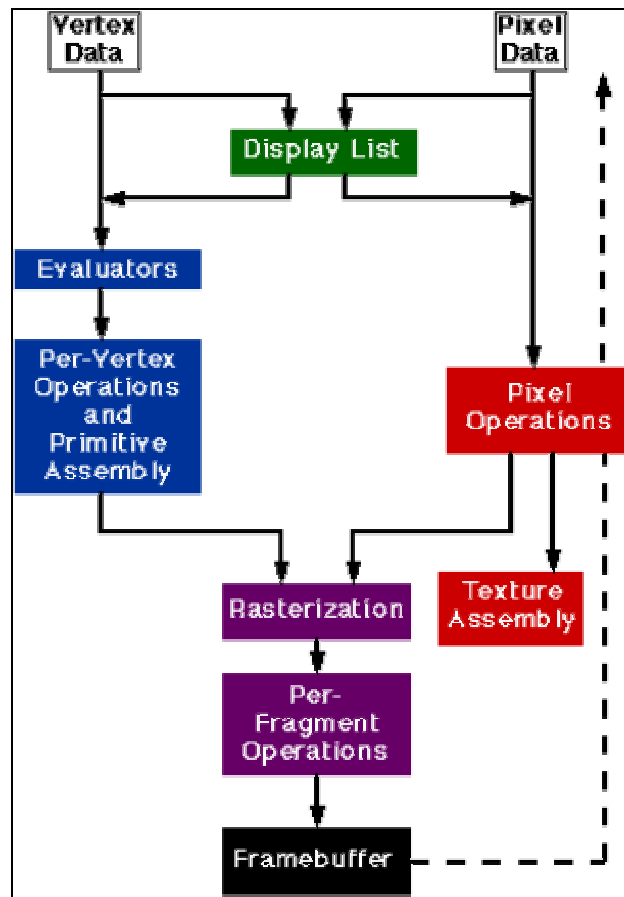
Ogl es una maquina de estados. Cuando se activan o configuran varios estados de la maquina, sus efectos perduraran hasta que sean desactivados. Por ejemplo, si el color para pintar pol gonos se pone a blanco, todos los pol gonos se pintaran de este color hasta cambiar el estado de esa variable. Existen otros estados que funcionan como booleanos (on o off, 0 o 1). Estos se activa mediante las funciones glEnable y glDisable. Veremos ejemplos practicos en los proximos cap tulos.

Todos los estados tienen un valor por defecto, y tambien alguna funcion con la que conseguir su valor actual. Estas pueden ser mas generales, del tipo glGetDoublev() o glIsEnabled(), o mas especificas, como glGetLight() o glGetError().

1.3 El Pipeline de renderizado de OpenGL

La mayor parte de las implementaciones de ogl siguen un mismo orden en sus operaciones, una serie de plataformas de proceso, que en su conjunto crean lo que se suele llamar el “OpenGL Rendering Pipeline” .

El siguiente diagrama (Ilustracion 1.1) describe el funcionamiento del pipeline:



Ilustracion 1.1

En este diagrama se puede apreciar el orden de operaciones que sigue el pipeline para renderizar. Por un lado tenemos el “vertex data”, que describe los objetos de nuestra escena, y por el otro, el “pixel data”, que describe las propiedades de la escena que se aplican sobre la imagen tal y como se representa en el buffer. Ambas se pueden guardar

en una “display list”, que es un conjunto de operaciones que se guardan para ser ejecutadas en cualquier momento.

Sobre el “vertex data” se pueden aplicar “evaluators”, para describir curvas o superficies parametrizadas mediante puntos de control. Luego se aplicaran las “per-vertex operations”, que convierten los vertices en primitivas. Aqu es donde se aplican las transformaciones geometricas como rotaciones, translaciones, etc., por cada vertice. En la seccion de “primitive assembly”, se hace clipping de lo que queda fuera del plano de proyeccion, entre otros.

Por la parte de “pixel data”, tenemos las “pixel operations”. Aqu los pixels son desempaquetados desde algun array del sistema (como el framebuffer) y tratados (escalados, etc.). Luego, si estamos tratando con texturas, se preparan en la seccion “texture assembly”.

Ambos caminos convergen en la “Rasterization”, donde son convertidos en fragmentos. Cada fragmento sera un pixel del framebuffer. Aqu es donde se tiene en cuenta el modelo de sombreado, la anchura de las lineas, o el antialiasing.

En la ultima etapa, las “per-fragment operations”, es donde se preparan los texels (elementos de texturas) para ser aplicados a cada pixel, la fog (niebla), el z-buffering, el blending, etc. Todas estas operaciones desembocan en el framebuffer, donde obtenemos el render final.

1.4 Escribir codigo basado en OpenGL

1.4.1 Sintaxis

Todas las funciones de ogl comienzan con el prefijo “gl” y las constantes con “GL_”. Como ejemplos, la funcion `glClearColor()` y la constante `GL_COLOR_BUFFER_BIT`.

En muchas de las funciones, aparece un sufijo compuesto por dos digitos, una cifra y una letra, como por ejemplo `glColor3f()` o `glVertex3i()`. La cifra simboliza el numero de parametros que se le deben pasar a la funcion, y la letra el tipo de estos parametros.

En ogl existen 8 tipos distintos de datos, de una forma muy parecida a los tipos de datos de C o C++. Ademas, ogl viene con sus propias definiciones de estos datos (typedef en C). Los tipos de datos:

Sufijo	Tipo de dato	Corresponde en C al tipo...	Definicion en ogl del tipo
b	Entero 8-bits	signed char	GLbyte
s	Entero 16-bits	short	GLshort
i	Entero 32-bits	int o long	GLint, GLsizei
f	Punto flotante 32-bits	float	GLfloat, GLclampf
d	Punto flotante 64-bits	double	GLdouble, GLclampd
ub	Entero sin signo 8-bits	unsigned char	GLubyte, GLboolean
us	Entero sin signo 16-bits	unsigned short	GLushort
ui	Entero sin signo 32-bits	unsigned int	GLuint, GLenum, GLbitfield

1.4.2 Animacion

Es muy importante dentro de los graficos en computacion la capacidad de conseguir movimiento a partir de una secuencia de fotografias o “frames”. La animacion es fundamental para un simulador de vuelo, una aplicacion de mecanica industrial o un juego.

En el momento en que el ojo humano percibe mas de 24 frames en un segundo, el cerebro lo interpreta como movimiento real. En este momento, cualquier proyector convencional es capaz de alcanzar frecuencias de 60 frames/s o mas. Evidentemente, 60 fps (frames por segundo) sera mas suave (y, por tanto, mas real) que 30 fps.

La clave para que la animacion funcione es que cada frame este completo (haya terminado de renderizar) cuando sea mostrado por pantalla. Supongamos que nuestro algoritmo (en pseudocodigo) para la animacion es el siguiente:

```

abre_ventana();
for (i=0; i< ultimo_frame; i++) {
  borra_ventana();

```

```
dibuja_frame(i);  
espera_hasta_la_1/24_parte_de_segundo();  
}
```

Si el borrado de pantalla y el dibujado del frame tardan mas de 1/24 de segundo, antes de que se pueda dibujar el siguiente, el borrado ya se habra producido. Esto causar a un parpadeo en la pantalla puesto que no hay una sincronizacion en los tiempos.

Para solucionar este problema, ogl nos permite usar la clasica tecnica del doble-buffering: las imagenes renderizadas se van colocando en el primer buffer, y cuando termina el renderizado, se vuelca al segundo buffer, que es el que se dibuja en pantalla. As nunca veremos una imagen cortada, solventando el problema del parpadeo. El algoritmo quedar a:

```
abre_ventana();  
for (i=0; i< ultimo_frame; I++) {  
  borra_ventana();  
  dibuja_frame(i);  
  swap_buffers();  
}
```

La cantidad de fps siempre quedara limitada por el refresco de nuestro monitor. Aunque OpenGL sea capaz de renderizar 100 frames en un segundo, si el periferico utilizado (monitor, canon) solo nos alcanza los 60 hz., es decir, las 60 imagenes por segundo, aproximadamente 40 frames renderizados se perderan.

1.4.3 Librer as relacionadas con OpenGL

OpenGL contiene un conjunto de poderosos pero primitivos comandos, a muy bajo nivel. Ademas la apertura de una ventana en el sistema grafico que utilicemos (win32, X11, etc.) donde pintar no entra en la [] de OpenGL. Por eso las siguientes librer as son muy utilizadas en la programacion de aplicaciones de ogl:

OpenGL Utility Library (GLU): contiene bastantes rutinas que usan ogl a bajo nivel para realizar tareas como transformaciones de matrices para tener una orientacion especifica, subdivision de pol gonos, etc.

GLX y WGL: GLX da soporte para maquinas que utilicen X Windows System, para inicializar una ventana, etc. WGL seria el equivalente para sistemas Microsoft.

OpenGL Utility Toolkit (GLUT): es un sistema de ventanas, escrito por Mark Kilgard, que seria independiente del sistema usado, dandonos funciones tipo abrir_ventana().

1.5 Pretensiones de estos apuntes

Este “tutorial” de ogl pretende enseñar al lector a “aprender OpenGL”. Ogl es un API muy extenso, con gran cantidad de funciones, estados, etc. Aquí se intentarán asentar las bases de la programación con el API de ogl, la dinámica con la que se debe empezar a escribir código para una aplicación gráfica que utilice ogl.

Los ejemplos que se muestran durante esta guía están escritos utilizando C, por ser el lenguaje más utilizado actualmente en este tipo de aplicaciones, además de ser el código “nativo” de OpenGL. También se utilizan las librerías GLU y GLUT, que serán explicadas más adelante.

Se presuponen una serie de conocimientos matemáticos, como son el tratamiento con matrices, o la manera de hallar los vectores normales a una superficie.

Se pretende que esta guía junto con una buena referencia (como el RedBook, citado en la bibliografía) sea suficiente para comenzar con el mundo 3D.

Estos apuntes contendrán los siguientes capítulos:

El **primero**, esta introducción.

El **segundo**, un ejemplo práctico para empezar a analizar la dinámica con la que se va a trabajar.

El **tercero**, aprenderemos a dibujar en tres dimensiones.

El **cuarto**, nos empezaremos a mover por nuestro entorno 3D gracias a las transformaciones matriciales.

El **quinto**, donde aprenderemos a usar los colores, materiales, iluminación y sombreado.

El **sexto**, veremos una aproximación a las técnicas de “blending” y mapeado de texturas, y

el **septimo**, sobre las “display list” y otras características útiles de OpenGL.

2 El “Hello World” de OpenGL

2.1 Requisitos del sistema

2.1.1 Hardware

En realidad, el API de ogl esta pensado para trabajar bajo el respaldo de un hardware capaz de realizar las operaciones necesarias para el renderizado, pero si no se dispone de ese hardware, estas operaciones se calcularan por medio de un software contra la CPU del sistema. As que los requerimientos hardware son escasos, aunque cuanto mayor sea las capacidades de la maquina, mayor sera el rendimiento de las aplicaciones ogl.

2.1.2 Software

Para estos apuntes, supondremos la utilizacion de un sistema Unix, con X Windows System. Concretamente, sistema linux con X11. La libreria Mesa3D, es un clon de OpenGL gratuito, y que nos vale perfectamente al ser compatible al 100% con OpenGL. Se puede bajar de <http://mesa3d.sourceforge.net>

Lo segundo que necesitaremos sera la libreria GLUT, que podemos bajar directamente de SGI. La direccion es <http://reality.sgi.com/opengl/glut3/glut3.html>

Con esto debera ser suficiente para empezar a desarrollar aplicaciones ogl sin problema.

2.2 La OpenGL Utility Toolkit (GLUT)

Como ya se ha mencionado, la libreria glut esta disenada para no tener preocupaciones con respecto al sistema de ventanas, incluyendo funciones del tipo `abre_ventana()`, que nos ocultan la complejidad de librerias a mas bajo nivel como la GLX. Pero ademas GLUT nos ofrece toda una dinamica de programacion de aplicaciones ogl, gracias a la definicion de funciones callback. Una funcion callback sera llamada cada vez que se produzca un evento, como la pulsacion de una tecla, el reescalado de la ventana o el mismo idle. Cuando utilizamos esta libreria, le damos el control del flujo del programa

a glut, de forma que ejecutara código de diferentes funciones dependiendo del estado actual del programa (idle, el ratón cambia de posición, etc.).

2.3 “Hello World”

2.3.1 Código

Lo que podrá considerarse la aplicación más sencilla utilizando OpenGL se presenta en esta sección. El siguiente código abre una ventana y dibuja dentro un polígono (triángulo en este caso) blanco.

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

void reshape(int width, int height)
{
    glViewport(0, 0, width, height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-1, 1, -1, 1, -1, 1);
    glMatrixMode(GL_MODELVIEW);
}

void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1,1,1);
    glLoadIdentity();
    glBegin(GL_TRIANGLES);
        glVertex3f(-1,-1,0);
        glVertex3f(1,-1,0);
        glVertex3f(0,1,0);
    glEnd();
    glFlush();
}
```

```

void init()
{
    glClearColor(0,0,0,0);
}

int main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowPosition(50, 50);
    glutInitWindowSize(500, 500);
    glutCreateWindow("Hello OpenGL");
    init();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutMainLoop();
    return 0;
}

```

Para compilar este código:

```
gcc -lglut -lGLU -lGL hello.c -o hello
```

2.3.2 Analisis del código

La estructura de un clásico programa sobre GLUT es la que se aprecia en el hello.c. Analicemos la función main():

```
glutInit(&argc, argv);
```

Esta función es la que inicializa la GLUT, y negocia con el sistema de ventanas para abrir una. Los parámetros deben ser los mismos argc y argv sin modificar de la main(). Glut entiende una serie de parámetros que pueden ser pasados por línea de comandos.

```
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
```

Define el modo en el que debe dibujar en la ventana. Los parametros, como gran parte de las funciones que iremos viendo, se definen con flags o mascarar de bits. En este caso en concreto, GLUT_SINGLE indica que se debe usar un solo buffer y GLUT_RGB el tipo de modelo de color con el que se dibujara.

```
glutInitWindowPosition(50, 50);
```

Posicion x e y de la esquina superior izquierda de la nueva ventana, con respecto al escritorio en el que se trabaje.

```
glutInitWindowSize(500, 500);
```

El ancho y alto de la nueva ventana.

```
glutCreateWindow("Hello OpenGL");
```

Esta funcion es la que propiamente crea la ventana, y el parametro es el nombre de la misma.

```
init();
```

En esta funcion que hemos definido nosotros, activamos y definimos una serie de estados de ogl, antes de pasar el control del programa a la GLUT.

```
glutDisplayFunc(display);
```

Aqu se define el primer callback. La funcion pasada como parametro sera llamada cada vez que GLUT determine oportuno que la ventana debe ser redibujada, como al maximizarse, poner otras ventanas por encima o sacarlas, etc.

```
glutReshapeFunc(reshape);
```

Aqu definimos otro callback, en este caso para saber que hace cuando la ventana es expl citamente reescalada. Esta accion afecta en principio directamente al render, puesto que se esta cambiando el tamano del plano de proyeccion. Por eso en esta funcion (en este caso reshape) se suele corregir esto de alguna forma. Lo veremos mejor mas adelante.

```
glutMainLoop();
```

Esta funcion cede el control del flujo del programa a la GLUT, que a partir de estos "eventos", ira llamando a las funciones que han sido pasadas como callbacks.

GLUT tiene muchas mas posibilidades, por supuesto. Nos ofrece funciones para el manejo del raton y teclado, y gran facilidad para la creacion de menus, que permite vincular con un evento como el clic del raton.

Contenido de los callbacks

Aunque no sea uno de ellos, empezaremos analizando la funcion `init()`. Como se ha comentado, ogl puede ser vista como una maquina de estados. Por lo tanto antes de empezar a hacer nada, habra que configurar alguno de estos estados. En `init()` podemos apreciar:

```
glClearColor(0,0,0,0);
```

Con esto se define el color con el que se borrara el buffer al hacer un `glClear()`. Los 3 primeros parametros son las componentes R, G y B, siguiendo un rango de `[0..1]`. La ultima es el valor alpha, del que hablaremos mas adelante.

Veamos ahora la funcion `reshape()`. Esta funcion, al ser pasada a `glutReshapeFunc`, sera llamada cada vez que se reescale a ventana. La funcion siempre debe ser definida con el siguiente esqueleto:

```
void rescala(int, int) { ... }
```

El primer parametro sera el ancho y el segundo el alto, despues del reescalado. Con estos dos valores trabajara la funcion cuando, en tiempo de ejecucion, el usuario reescale la ventana.

Analicemos ahora el contenido de nuestra funcion:

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

Esta funcion define la porcion de ventana donde puede dibujar ogl. Los parametros son `x` e `y`, esquina superior izquierda del "cuadro" donde puede dibujar (con referencia la ventana), y ancho y alto. En este caso coje el `width` y `height`, que son los parametros de `reshape()`, es decir, los datos que acaba de recibir por culpa del reescalado de la ventana.

```
glMatrixMode(GL_PROJECTION);
```

Especifica la matriz actual. En ogl las operaciones de rotacion, translacion, escalado, etc. se realizan a traves de matrices de transformacion. Dependiendo de lo que estemos tratando, hay tres tipos de matriz (que son los tres posibles flags que puede llevar de parametro la funcion): matriz de proyeccion (GL_PROJECTION), matriz de modelo (GL_MODELVIEW) y matriz de textura (GL_TEXTURE). Con esta funcion indicamos a cual de estas tres se deben afectar las operaciones. Concretamente, GL_PROJECTION afecta a las vistas o perspectivas o proyecciones. Todo esto se vera en el capitulo 4.

```
glLoadIdentity();
```

Con esto cargamos en el "tipo" de matriz actual la matriz identidad (es como resetear la matriz).

```
glOrtho(-1, 1, -1, 1, -1, 1);
```

glOrtho() define una perspectiva ortonormal. Esto quiere decir que lo que veremos sera una proyeccion en uno de los planos definidos por los ejes. Es como plasmar los objetos en un plano, y luego observar el plano. Los parametros son para delimitar la zona de trabajo, y son x_minima, x_maxima, y_minima, y_maxima, z_minima, z_maxima. Con estos seis puntos, definimos una caja que sera lo que se proyecte. Esta funcion se trata mas a fondo en el capitulo 4.

```
glMatrixMode(GL_MODELVIEW);
```

Se vuelve a este tipo de matrices, que afecta a las primitivas geometricas.

Ya solo nos queda la funcion display() por ver. Como se ha dicho, al ser pasada a glutDisplayFunc(), sera llamada cada vez que haya que redibujar la ventana. La funcion debe ser definida con el siguiente esqueleto:

```
void dibujar(void) { ... }
```

Analicemos ahora el contenido de la funcion en nuestro ejemplo:

```
glClear(GL_COLOR_BUFFER_BIT);
```

Borra un buffer, o una combinacion de varios, definidos por flags. En este caso, el buffer de los colores lo borra (en realidad, cada componente R G y B tienen un buffer distinto, pero aqui los trata como el mismo). Para borrarlos utiliza el color que ha sido

previamente definido en `init()` mediante `glClearColor()`, en este caso, el (0,0,0,0) es decir, negro. La composicion de colores se vera en el capitulo 5.

```
glColor3f(1,1,1);
```

Selecciona el color actual con el que dibujar. Parametros R G y B, rango [0..1], as que estamos ante el color blanco.

```
glLoadIdentity();
```

Carga la matriz identidad.

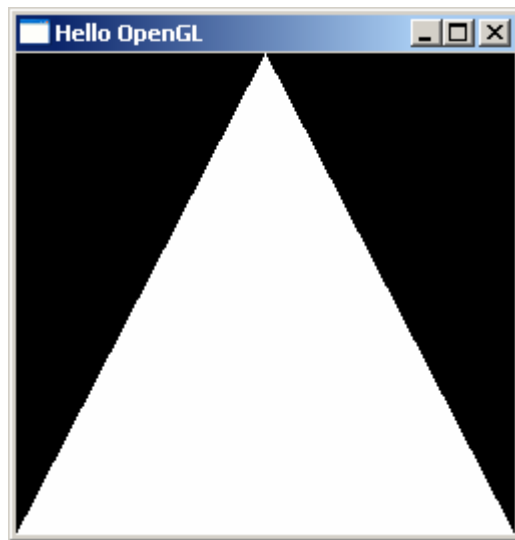
```
glBegin(GL_TRIANGLES);  
glVertex3f(-1,-1,0);  
glVertex3f(1,-1,0);  
glVertex3f(0,1,0);  
glEnd();
```

Analizamos toda esta parte entera, por formar una estructura. `glBegin()` comienza una secuencia de vertices con los que se construiran primitivas. El tipo de primitivas viene dado por el parametro de `glBegin()`, en este caso `GL_TRIANGLES`. Al haber tres vertices dentro de la estructura, esta definiendo un triangulo. `glEnd()` simplemente cierra la estructura. Los posibles parametros de `glBegin`, y la manera de construir primitivas se veran mas adelante, en proximo capitulo.

```
glFlush();
```

Dependiendo del hardware, controladores, etc., ogl guarda los comandos como peticiones en pila, para optimizar el rendimiento. El comando `glFlush` causa la ejecucion de cualquier comando en espera.

El resultado final se puede ver en la ilustracion 2.1.



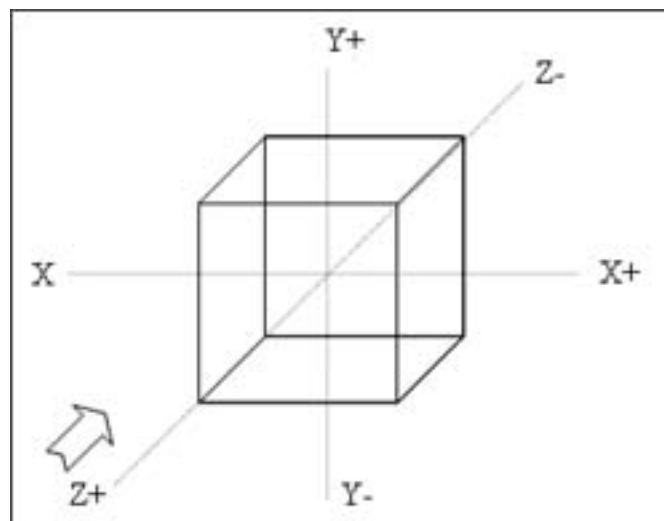
Ilustracion 2.1

3 Dibujando en 3D

El dibujo 3D en OpenGL se basa en la composicion de pequenos elementos, con los que vamos construyendo la escena deseada. Estos elementos se llaman primitivas. Todas las primitivas de ogl son objetos de una o dos dimensiones, abarcando desde puntos simples a l neas o pol gonos complejos. Las primitivas se componen de vertices, que no son mas que puntos 3D. En este cap tulo se pretende presentar las herramientas necesarias para dibujar objetos en 3D a partir de estas formas mas sencillas. Para ello necesitamos deshacernos de la mentalidad en 2D de la computacion grafica clasica y definir el nuevo espacio de trabajo, ya en 3D.

3.1 Definicion de un lienzo en 3D

La Ilustracion 3.1 muestra un eje de coordenadas inmerso en un volumen de visualizacion sencillo, que utilizaremos para definir y explicar el espacio en el que vamos a trabajar. Este volumen se corresponder a con una perspectiva ortonormal, como la que hemos definido en el capitulo anterior haciendo una llamada a `glOrtho()`. Como puede observarse en la figura, para el punto de vista el eje de las x ser a horizontal, y crecer a de izquierda a derecha; el eje y, vertical, y creciendo de abajo hacia arriba; y, por ultimo, el eje z, que ser a el de profundidad, y que crecer a hacia nuestras espaldas, es decir, cuanto mas lejos de la camara este el punto, menor sera su coordenada z. En el siguiente cap tulo se abordara este tema con mayor profundidad, cuando definamos las transformaciones sobre el espacio.



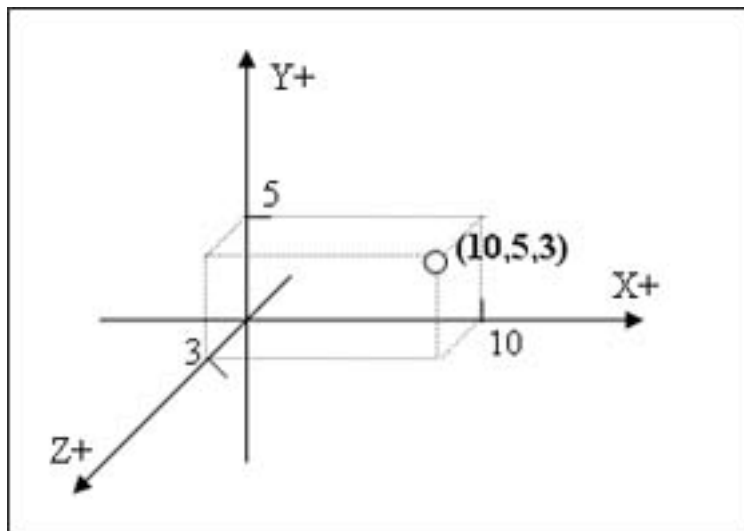
Ilustracion 3.1

3.2 El punto en 3D: el vertice

Los vertices (puntos 3D) son el denominador comun en cualquiera de las primitivas de OpenGL. Con ellos se definen puntos, lineas y poligonos. La funcion que define vertices es glVertex, y puede tomar de dos a cuatro parametros de cualquier tipo numerico. Por ejemplo, la siguiente linea de codigo define un vertice en el punto (10, 5, 3).

```
glVertex3f(10.0f, 5.0f, 3.0f);
```

Este punto se muestra en la Ilustracion 3.2. Aquì hemos decidido representar las coordenadas como valores en coma flotante, y con tres argumentos, x, y y z, respectivamente.



Ilustracion 3.2

Ahora debemos aprender a darle sentido a estos vertices, que pueden ser tanto la esquina de un cubo como el extremo de una linea.

3.3 Las primitivas

Una primitiva es simplemente la interpretacion de un conjunto de vertices, dibujados de una manera especifica en pantalla. Hay diez primitivas distintas en ogl, pero en estos apuntes explicaremos solamente las mas importantes: puntos (GL_POINTS), lineas (GL_LINES), triangulos (GL_TRIANGLES) y cuadrados (GL_QUADS). Comentaremos tambien las primitivas GL_LINES_STRIP, GL_TRIANGLE_STRIP y GL_QUAD_STRIP, utilizadas para definir “tiras” de lineas, triangulos y de cuadrados respectivamente.

Para crear primitivas en ogl se utilizan las funciones glBegin y glEnd. La sintaxis de estas funciones sigue el siguiente modelo:

```
glBegin(<tipo de primitiva>);  
    glVertex(...);  
    glVertex(...);  
    ...  
    glVertex(...);  
glEnd();
```

Puede observarse que glBegin y glEnd actúan como llaves (“{” y “}”) de las primitivas, por eso es común añadirle tabulados a las glVertex contenidos entre ellas. El parámetro de glBegin <tipo de primitiva> es del tipo GLenum (definido por OpenGL), y será el flag con el nombre de la primitiva (GL_POINTS, GL_QUADS, etc.).

3.3.1 Dibujo de puntos (GL_POINTS)

Es la más simple de las primitivas de ogl: los puntos. Para comenzar, veamos el siguiente código:

```
glBegin(GL_POINTS);  
    glVertex3f(0.0f, 0.0f, 0.0f);  
    glVertex3f(10.0f, 10.0f, 10.0f);  
glEnd();
```

El parámetro pasado a glBegin es GL_POINTS, con lo cual interpreta los vertices contenidos en el bloque glBegin-glEnd como puntos. Aquí se dibujarán dos puntos, en (0, 0, 0) y en (10, 10, 10). Como podemos ver, podemos listar múltiples primitivas entre

llamadas mientras que sean para el mismo tipo de primitiva. El siguiente código dibujara exactamente lo mismo, pero invertira mucho mas tiempo en hacerlo, decelerando considerablemente la velocidad de nuestra aplicacion:

```
glBegin(GL_POINTS);  
    glVertex3f(0.0f, 0.0f, 0.0f);  
glEnd();  
  
glBegin(GL_POINTS);  
    glVertex3f(10.0f, 10.0f, 10.0f);  
glEnd();
```

3.3.1.1 Ajuste del tamaño del punto

Cuando dibujamos un punto, el tamaño por defecto es de un pixel. Podemos cambiar este ancho con la función `glPointSize`, que lleva como parametro un flotante con el tamaño aproximado en pixels del punto dibujado. Sin embargo, no esta permitido cualquier tamaño. Para conocer el rango de tamaños soportados y el paso (incremento) de estos, podemos usar la función `glGetFloatv`, que devuelve el valor de alguna medida o variable interna de OpenGL, llamadas “variables de estado”, que pueda depender directamente de la maquina o de la implementacion de ogl. Así, el siguiente código conseguirá a estos valores:

```
Gfloat rango[2];  
Gfloat incremento;  
  
glGetFloatv(GL_POINT_SIZE_RANGE, rango);  
glGetFloatv(GL_POINT_SIZE_GRANULARITY, &incremento);
```

Por ejemplo la implementación OpenGL de MicroSoft permite tamaños de punto de 0.5 a 10.0, con un incremento o paso de 0.125. Especificar un tamaño de rango incorrecto no causará un error, si no que la maquina OpenGL usará el tamaño correcto mas aproximado al definido por el usuario.

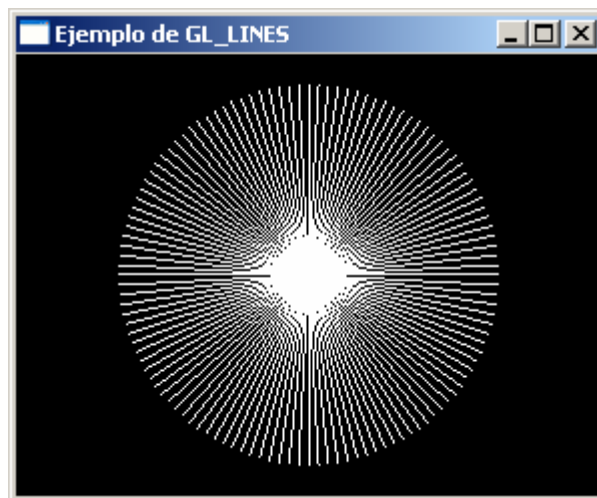
3.3.2 Dibujo de líneas (*GL_LINES*)

En el dibujado de puntos, la sintaxis era muy comoda: cada vertice es un punto. En las lineas los vertices se cuentan por parejas, denotando punto inicial y punto final de la linea. Si especificamos un numero impar de vertices, el ultimo de ellos se ignora.

El siguiente codigo dibuja una serie de lineas radiales:

```
GLfloat angulo;  
int i;  
glBegin(GL_LINES);  
for (i=0; i<360; i+=3)  
{  
    angulo = (GLfloat)i*3.14159f/180.0f; // grados a radianes  
    glVertex3f(0.0f, 0.0f, 0.0f);  
    glVertex3f(cos(angulo), sin(angulo), 0.0f);  
}  
glEnd();
```

Este ejemplo dibuja 120 lineas en el mismo plano (ya que en los puntos que las definen $z = 0.0f$), con el mismo punto inicial (0,0,0) y puntos finales describiendo una circunferencia. El resultado sera el de la Ilustracion 3.3.



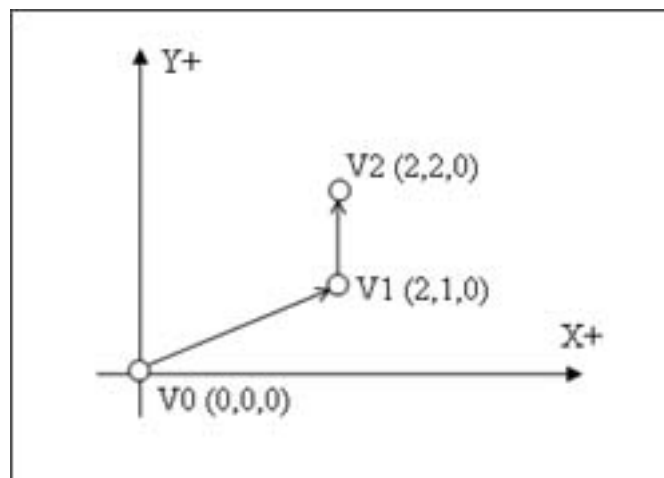
Ilustracion 3.3

Si en vez de GL_LINES utilizasemos GL_LINE_STRIP, ogl ya no tratar a los vertices en parejas, si no que el primer vertice y el segundo definir an una linea, y el final de esta

definir a otra linea con el siguiente vertice, y as sucesivamente, definiendo una segmento cont nuo. Veamos el siguiente codigo como ejemplo:

```
glBegin(GL_LINE_STRIP);  
    glVertex3f(0.0f, 0.0f, 0.0f); // V0  
    glVertex3f(2.0f, 1.0f, 0.0f); // V1  
    glVertex3f(2.0f, 2.0f, 0.0f); // V2  
glEnd();
```

Este codigo construir a las l neas como se ve en la Ilustracion 3.4.



Ilustracion 3.4

Mencionaremos por ultimo la primitiva GL_LINE_LOOP, que funciona igual que GL_LINE_STRIP, pero ademas une el primer vertice con el ultimo, creando siempre una cuerda cerrada.

3.3.3 Dibujo de poligonos

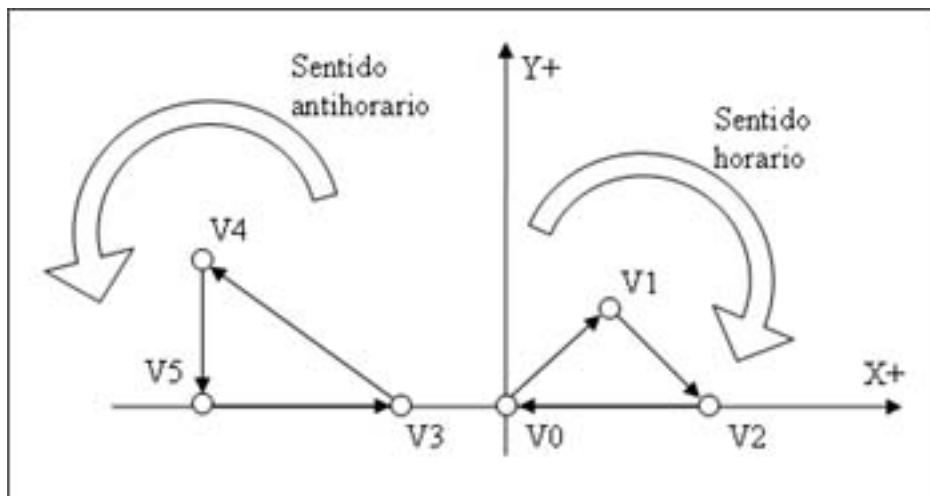
En la creacion de objetos solidos, el uso de puntos y l neas es insuficiente. Se necesitan primitivas que sean superficies cerradas, rellenas de uno o varios colores, que, en conjunto, modelen el objeto deseado. En el campo de la representacion 3D de los

graficos en computacion, se suelen utilizar poligonos (que a menudo son triangulos) para dar forma a objetos “semisolidos” (ya que en realidad son superficies, estan huecos por dentro). Ahora veremos la manera de hacer esto mediante las primitivas GL_TRIANGLES y GL_QUADS.

3.3.3.1 Triangulos (GL_TRIANGLES)

El poligono mas simple, es el triangulo, con solo tres lados. En esta primitiva, los vertices van de tres en tres. El siguiente codigo dibuja dos triangulos, como se muestra en la Ilustracion 3.5:

```
glBegin(GL_TRIANGLES);  
    glVertex3f(0.0f, 0.0f, 0.0f); // V0  
    glVertex3f(1.0f, 1.0f, 0.0f); // V1  
    glVertex3f(2.0f, 0.0f, 0.0f); // V2  
  
    glVertex3f(-1.0f, 0.0f, 0.0f); // V3  
    glVertex3f(-3.0f, 2.0f, 0.0f); // V4  
    glVertex3f(-2.0f, 0.0f, 0.0f); // V5  
glEnd();
```



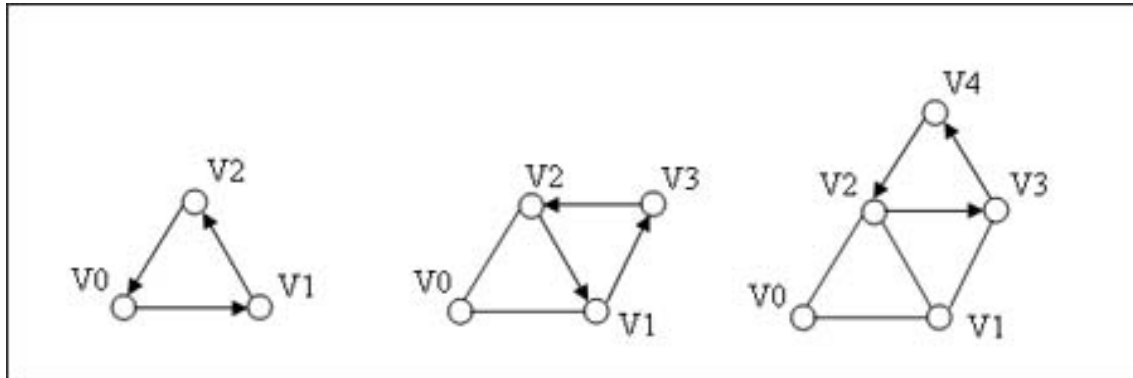
Ilustracion 3.5

Hagamos hincapie en el orden en que especificamos los vertices. En el primer triangulo (el de la derecha), se sigue la pol tica de “sentido horario , y en el segundo, “sentido antihorario . Cuando un pol gono cualquiera, tiene sentido horario (los vertices avanzan en el mismo sentido que las agujas del reloj), se dice que es positivo; en caso contrario, se dice que es negativo. OpenGL considera que, por defecto, los pol gonos que tienen sentido negativo tienen un “encare frontal . Esto significa que el triangulo de la izquierda nos muestra su cara frontal, y el de la derecha su cara trasera. Veremos mas adelante que es sumamente importante mantener una consistencia en el sentido de los triangulos al construir objetos solidos.

Si necesitamos invertir el comportamiento por defecto de ogl, basta con una llamada a la funcion `glFrontFace()`. Esta acepta como parametro `GL_CW` (considera los pol gonos positivos con encare frontal) o `GL_CCW` (considera los pol gonos negativos con encare frontal).

Como con la primitiva de l neas, con triangulos tambien existe `GL_TRIANGLE_STRIP`, y funciona como se puede observar en la Ilustracion 3.6, que sigue el siguiente pseudo-codigo:

```
glBegin(GL_TRIANGLE_STRIP);
    glVertex(v0);
    glVertex(v1);
    ...
glEnd();
```



Ilustracion 3.6

Por ultimo comentar que la manera de componer objetos y su dibujarlos mas rapida es mediante triangulos, ya que solo necesitan tres vertices.

3.3.3.2 Cuadrados (GL_QUADS)

Esta primitiva funciona exactamente igual que GL_TRIANGLES, pero dibujando cuadrados. También tiene la variación de GL_QUAD_STRIP, para dibujar “tiras” de cuadrados.

3.4 Construcción de objetos sólidos mediante polígonos

Componer un objeto sólido a partir de polígonos implica algo más que ensamblar vértices en un espacio coordenado 3D. Veremos ahora una serie de puntos a tener en cuenta para construir objetos, aunque dando solo una breve acercamiento ya que cada uno de estos puntos se verá más a fondo en los siguientes capítulos.

3.4.1 Color de relleno

Para elegir el color de los polígonos, basta con hacer una llamada a glColor entre la definición de cada polígono. Por ejemplo, modifiquemos el código que dibujaba dos triángulos:

```
glBegin(GL_TRIANGLES);
    glColor3f(1.0f, 0.0f, 0.0f);
    glVertex3f(0.0f, 0.0f, 0.0f);
    glVertex3f(2.0f, 0.0f, 0.0f);
    glVertex3f(1.0f, 1.0f, 0.0f);

    glColor3f(0.0f, 1.0f, 0.0f);
    glVertex3f(-1.0f, 0.0f, 0.0f);
    glVertex3f(-3.0f, 2.0f, 0.0f);
    glVertex3f(-2.0f, 0.0f, 0.0f);
glEnd();
```

Esta modificación provocará que el primer triángulo se pinte en rojo y el segundo en verde. La función glColor define el color de relleno actual y lleva como parámetros los valores de las componentes RGB del color deseado, y, opcionalmente, un cuarto parámetro con el valor alpha. De esto se hablará más adelante, aunque se adelanta que

estos parametros son flotantes que se mueven en el rango [0.0-1.0], y con ello se pueden componer todos los colores de el modo de video usado en ese instante.

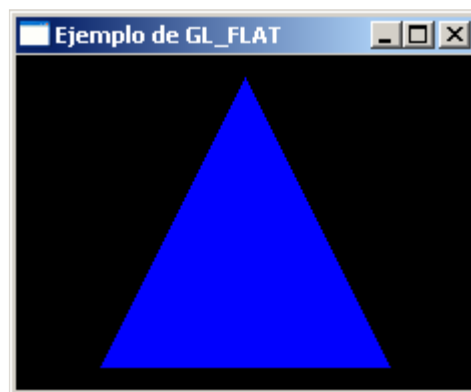
3.4.2 Modelo de sombreado

Es el metodo que utiliza OpenGL para rellenar de color los poligonos. Se especifica con la funcion `glShadeModel`. Si el parametro es `GL_FLAT`, ogl rellenara los poligonos con el color activo en el momento que se definio el ultimo parametro; si es `GL_SMOOTH`, ogl rellenara el poligono interpolando los colores activos en la definicion de cada vertice.

Este codigo es un ejemplo de `GL_FLAT`:

```
glShadeModel(GL_FLAT);
glBegin(GL_TRIANGLE);
    glColor3f(1.0f, 0.0f, 0.0f); // activamos el color rojo
    glVertex3f(-1.0f, 0.0f, 0.0f);
    glColor3f(0.0f, 1.0f, 0.0f); // activamos el color verde
    glVertex3f(1.0f, 0.0f, 0.0f);
    glColor3f(1.0f, 0.0f, 1.0f); // activamos el color azul
    glVertex3f(0.0f, 0.0f, 1.0f);
glEnd();
```

La salida ser a la Ilustracion 3.7

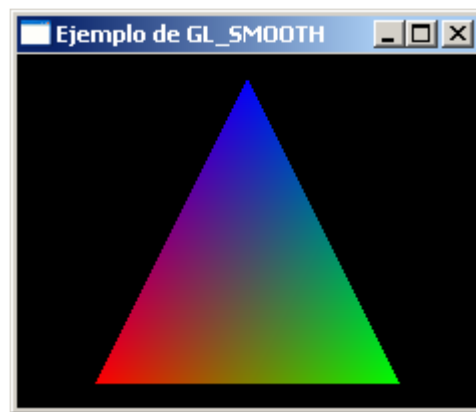


Ilustracion 3.7

El triangulo se rellena con el color azul, puesto que el modelo de sombreado es GL_FLAT y el color activo en la definicion del ultimo vertice es el azul. Sin embargo, este mismo codigo, cambiando la primera linea:

```
glShadeModel(GL_SMOOTH);
glBegin(GL_TRIANGLE);
    glColor3f(1.0f, 0.0f, 0.0f); // activamos el color rojo
    glVertex3f(-1.0f, 0.0f, 0.0f);
    glColor3f(0.0f, 1.0f, 0.0f); // activamos el color verde
    glVertex3f(1.0f, 0.0f, 0.0f);
    glColor3f(1.0f, 0.0f, 1.0f); // activamos el color azul
    glVertex3f(0.0f, 0.0f, 1.0f);
glEnd();
```

producir a una salida similar a la de la Ilustracion 3.8, donde se aprecia claramente la interpolacion de colores.



Ilustracion 3.8

3.4.3 Eliminacion de las caras ocultas

Cuando tenemos un objeto solido, o quiza varios objetos, algunos de ellos estaran mas proximos a nosotros que otros. Si un objeto esta por detras de otro, evidentemente, solo veremos el de delante, que tapa al de atras. OpenGL, por defecto, no tiene en cuenta esta posible situacion, de forma que simplemente va pintando en pantalla los puntos, lineas y poligonos siguiendo el orden en el que se especifican en el codigo. Veamos un ejemplo:

```

glColor3f(1.0f, 1.0f, 1.0f); // activamos el color blanco
glBegin(GL_TRIANGLES);
    glVertex3f(-1.0f, -1.0f, -1.0f);
    glVertex3f(1.0f, -1.0f, -1.0f);
    glVertex3f(0.0f, 1.0f, -1.0f);
glEnd();

glPointSize(6.0f); // tamaño del pixel = 6, para que se vea bien
glColor3f(1.0f, 0.0f, 0.0f); // activamos el color rojo
glBegin(GL_POINTS);
    glVertex3f(0.0f, 0.0f, -2.0f);
    glVertex3f(2.0f, 1.0f, -2.0f);
glEnd();

```

Aquí estamos pintando un triángulo de frente a nosotros, en el plano $z = -1$. Luego pintamos dos puntos, el primero en $(0,0,2)$ y el segundo en $(2,1,-2)$. Ambos están en el plano $z = -2$, es decir, más lejos de nuestro punto de vista que el triángulo. El primer punto nos debería tapar el triángulo, pero como por defecto OpenGL no comprueba que es lo que está por delante y por detrás, lo que hace es pintar primero el triángulo y después los puntos, quedando un resultado como el de la Ilustración 3.9.

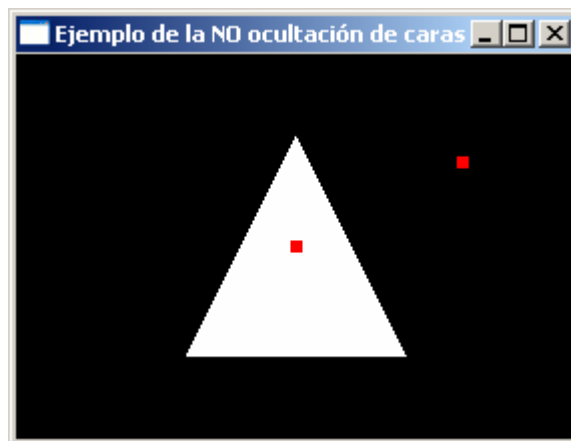


Ilustración 3.9

Para solucionar esto, introduciremos aquí uno de los buffers que OpenGL pone a nuestra disposición, el “depth buffer” (buffer de profundidad, también conocido como “z-buffer”). En él se almacenan “las zetas” o distancias desde el punto de vista a cada pixel.

de los objetos de la escena, y, a la hora de pintarlos por pantalla, hace una comprobación de que no haya ninguna primitiva que esté por delante tapando a lo que vamos a pintar en ese lugar.

Para activar esta característica, llamada “depth test”, solo tenemos que hacer una modificación de su variable en la máquina de estados de OpenGL, usando glEnable, de la siguiente forma:

```
glEnable(GL_DEPTH_BUFFER);
```

Para desactivarlo, usaremos la función glDisable con el mismo parámetro. Añadiendo esta última línea al principio del código del anterior ejemplo, obtendremos el efecto de ocultación deseado, como se puede ver en la Ilustración 3.10.

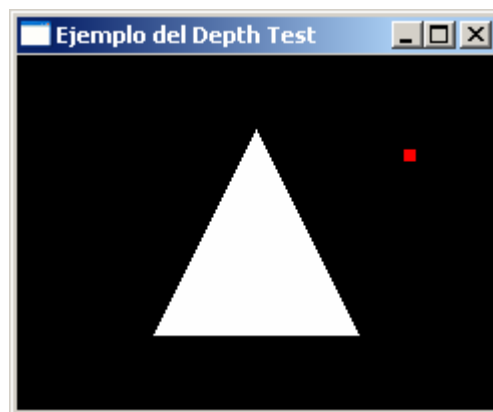


Ilustración 3.10

El uso del “test de profundidad” conlleva a que en la función que renderiza la escena, de la misma forma que hacemos al principio un glClear con la variable GL_COLOR_BUFFER_BIT, para que a cada frame nos limpiase la pantalla antes de dibujar el siguiente, debemos indicarle también que borre el depth buffer, por si algún objeto se ha movido y ha cambiado la situación de la escena, quedando la siguiente línea:

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

Podemos utilizar el operador binario “or” ya que las definiciones de ogl terminadas en “_BIT” son flags, pudiéndose combinar varias en una misma función.

Gracias a la eliminacion de las caras ocultas, ganamos gran realismo en la escena, y al tiempo ahorramos el procesado de todos aquellos pol gonos que no se ven, ganando tambien en velocidad.

4 Moviendonos por nuestro espacio 3D: transformaciones de coordenadas

En este capítulo aprenderemos a mover nuestro punto de vista sobre la escena y los objetos que la componen, sobre nuestro sistema de coordenadas. Como veremos, las herramientas que OpenGL nos aporta para hacer esto están basadas en matrices de transformación, que, aplicadas sobre los sistemas de coordenadas con un orden específico, construirán la escena deseada.

En las dos primeras secciones (coordenadas oculares y transformaciones) se intentará dar una idea de cómo trabaja OpenGL con el espacio 3D, a alto nivel. En las siguientes, se hablará más del código necesario para conseguir los resultados deseados. Se termina con un ejemplo en el que se aplica todo lo aprendido.

4.1 Coordenadas oculares

Las coordenadas oculares se sitúan en el punto de vista del observador, sin importar las transformaciones que tengan lugar. Por tanto, estas coordenadas representan un sistema virtual de coordenadas fijo usado como marco de referencia común. En la ilustración 4.1 se pueden apreciar dos perspectivas de este sistema de coordenadas.

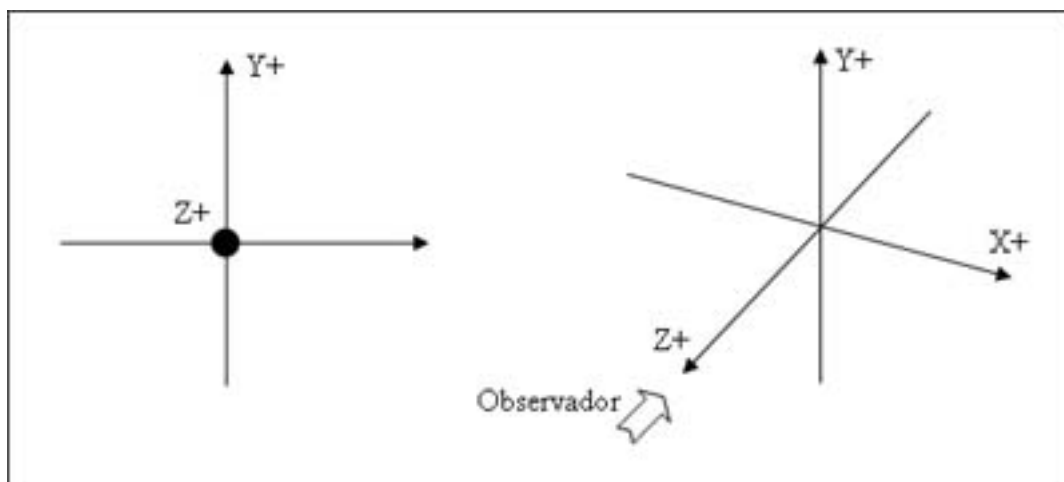


Ilustración 4.1

Cuando dibujamos en 3D con ogl, utilizamos el sistema de coordenadas cartesiano. En ausencia de cualquier transformacion, el sistema en uso sera identico al sistema de coordenadas oculares.

4.2 Transformaciones

Son las transformaciones las que hacen posible la proyeccion de coordenadas 3D sobre superficies 2D. Tambien son las encargadas de mover, rotar y escalar objetos. En realidad, estas transformaciones no se aplican a los modelos en s , si no al sistema de coordenadas, de forma que si queremos rotar un objeto, no lo rotamos a el, sino al eje sobre el que se situa. Las transformaciones 3D que OpenGL efectua se pueden apreciar en la siguiente tabla:

Del observador	Especifica la localizacion de la camara.
Del modeb	Mueve los objetos por la escena.
Del modelador	Describe la dualidad de las transformaciones del observador y del modelado.
De la proyeccion	Define y dimensiona el volumen de visualizacion.
De la vista	Escala la salida final a la ventana.

4.2.1 El modelador

En esta seccion se recogen las transformaciones del observador y del modelado puesto que, como veremos en el apartado 4.2.1.3, constituyen al fin y al cabo la misma transformacion.

4.2.1.1 Transformaciones del observador

La transformacion del observador es la primera que se aplica a nuestra escena, y se usa para determinar el punto mas ventajoso de la escena. Por defecto, el punto de vista esta en el origen (0,0,0) mirando en direccion negativa del eje z. La transformacion del observador nos permite colocar y apuntar la camara donde y hacia donde queramos. Todas las transformaciones posteriores tienen lugar basadas en el nuevo sistema de coordenadas modificado.

4.2.1.2 Transformaciones del modelo

Estas transformaciones se usan para situar, rotar y escalar los objetos de la escena. La apariencia final de nuestros objetos depende en gran medida del orden con el que se hayan aplicado las transformaciones. Por ejemplo, en la ilustración 4.2 podemos ver la diferencia entre aplicar primero un rotación y luego una translación, y hacer esto mismo invirtiendo el orden.

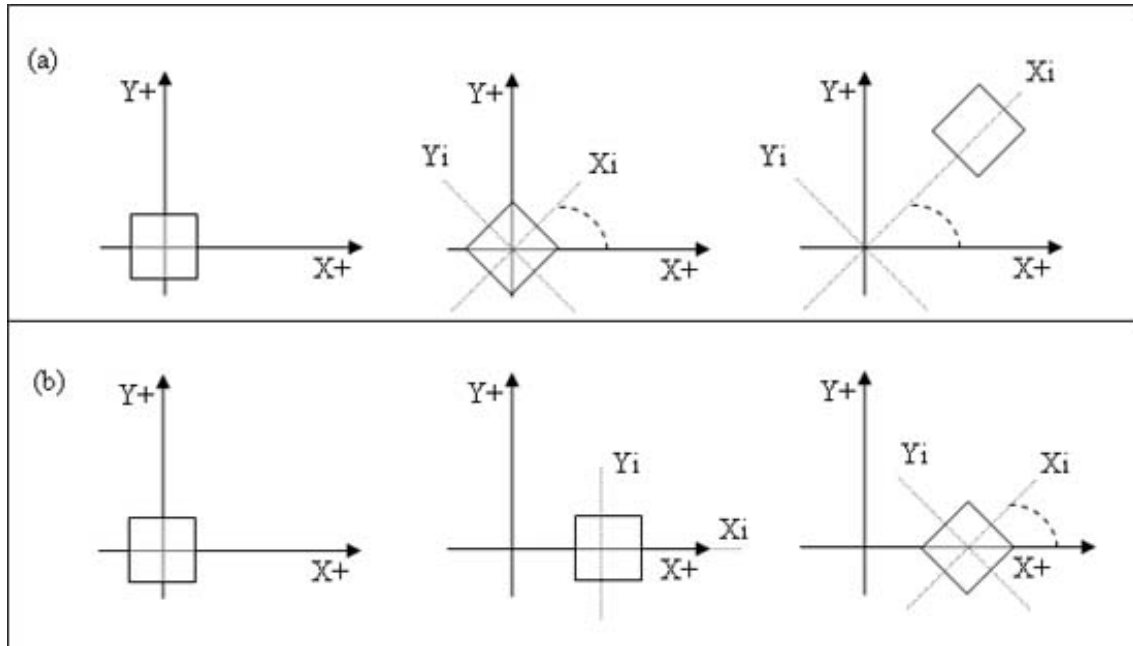


Ilustración 4.2

En el caso (a), primero se aplica una rotación, rotando así el sistema de coordenadas propio del objeto. Luego, al aplicar la translación sobre el eje x , como este está rotado, la figura ya no se desplaza sobre el eje x del sistema de coordenadas oculares, sino sobre el suyo propio. En el segundo caso, (b), primero se hace la translación sobre el eje x , así que la figura se mueve hacia nuestra derecha, también en el sistema de coordenadas oculares, ya que ambos sistemas coinciden. Luego se hace la rotación, pero el objeto gira sobre sí mismo, ya que aún está centrado en su propio sistema de coordenadas.

4.2.1.3 Dualidad del modelador

Las transformaciones del observador y del modelo son, en realidad, la misma en terminos de sus efectos internos as como en la apariencia final de la escena. Estan separadas unicamente por comodidad para el programador. En realidad, no hay ninguna diferencia en mover un objeto hacia atras o mover el sistema de coordenadas hacia delante.

4.2.2 Transformaciones de la proyeccion

La transformacion de proyeccion se aplica a la orientacion final del modelador. Esta proyeccion define el volumen de visualizacion y establece los planos de trabajo. A efectos practicos, esta translacion especifica como se traslada una escena finalizada a la imagen final de la pantalla.

Los dos tipos de proyeccion mas utilizados son la ortografica y la perspectiva, que veremos mas adelante.

4.2.3 Transformaciones de la vista

En el momento en que se ha terminado todo el proceso de transformaciones, solo queda un ultimo paso: proyectar lo que hemos dibujado en 3D al 2D de la pantalla, en la ventana en la que estamos trabajando. Esta es la denominada transformacion de la vista.

4.3 Matrices

Las matematicas que hay tras estas transformaciones se simplifican gracias a las matrices. Cada una de las transformaciones de las que acabamos de hablar puede conseguirse multiplicando una matriz que contenga los vertices por una matriz que describa la transformacion. Por tanto todas las transformaciones ejecutables con ogl pueden describirse como la multiplicacion de dos o mas matrices.

4.3.1 El canal de transformaciones

Para poder llevar a cabo todas las transformaciones de las que acabamos de hablar, deben modificarse dos matrices: la matriz del Modelador y la matriz de Proyeccion. OpenGL proporciona muchas funciones de alto nivel que hacen muy sencillo la construccion de matrices para transformaciones. Estas se aplican sobre la matriz que

este activa en ese instante. Para activar una de las dos matrices utilizamos la funcion `glMatrixMode`. Hay dos parametros posibles:

```
glMatrixMode(GL_PROJECTION);
```

activa la matriz de proyeccion, y

```
glMatrixMode(GL_MODELVIEW);
```

activa la del modelador. Es necesario especificar con que matriz se trabaja, para poder aplicar las transformaciones necesarias en funcion de lo que deseemos hacer.

El camino que va desde los datos “en bruto” de los vertices hasta la coordenadas en pantalla sigue el siguiente camino: Primero, nuestro vertice se convierte en una matriz 1x4 en la que los tres primeros valores son las coordenadas x,y,z. El cuarto numero (llamado parametro w) es un factor de escala, que no vamos a usar de momento, usando el valor 1.0. Entonces se multiplica el vertice por la matriz del modelador, para obtener las coordenadas oculares. Estas se multiplican por la matriz de proyeccion para conseguir las coordenadas de trabajo. Con esto eliminamos todos los datos que esten fuera del volumen de proyeccion. Estas coordenadas de trabajo se dividen por el parametro w del vertice, para hacer el escalado relativo del que hablamos antes. Finalmente, las coordenadas resultantes se mapean en un plano 2D mediante la transformacion de la vista.

4.3.2 La matriz del modelador

La matriz del modelador es una matriz 4x4 que representa el sistema de coordenadas transformado que estamos usando para colocar y orientar nuestros objetos. Si multiplicamos la matriz de nuestro vertice (de tamaño 1x4) por esta obtenemos otra matriz 1x4 con los vertices transformados sobre ese sistema de coordenadas.

OpenGL nos proporciona funciones de alto nivel para conseguir matrices de translacion, rotacion y escalado, y ademas la multiplican por la matriz activa en ese instante, de manera que nosotros no tenemos que preocuparnos por ello en absoluto.

4.3.2.1 Translacion

Imaginemos que queremos dibujar un cubo con la funcion de la libreria GLUT `glutSolidCube`, que lleva como parametro el lado del cubo. Si escribimos el siguiente codigo

```
glutSolidCube(5);
```

obtendremos un cubo centrado en el origen (0,0,0) y con el lado de la arista 5. Ahora pensemos que queremos moverlo 10 unidades hacia la derecha (es decir, 10 unidades en el sentido positivo del eje de las x). Para ello tendremos que construir una matriz de transformacion y multiplicarla por la matriz del modelador. Ogl nos ofrece la funcion `glTranslate`, que crea la matriz de transformacion y la multiplica por la matriz que este activa en ese instante (en este caso debera ser la del modelador, `GL_MODELVIEW`). Entonces el codigo quedara de la siguiente manera:

```
glTranslatef(5.0f, 0.0f, 0.0f);  
glutSolidCube(5);
```

La “f” anadida a la funcion indica que usaremos flotantes. Los parametros de `glTranslate` son las unidades a desplazar en el eje x, y y z, respectivamente. Pueden ser valores negativos, para trasladar en el sentido contrario.

4.3.2.2 Rotacion

Para rotar, tenemos tambien una funcion de alto nivel que construye la matriz de transformacion y la multiplica por la matriz activa, `glRotate`. Lleva como parametros el angulo a rotar (en grados, sentido horario), y despues x, y y z del vector sobre el cual queremos rotar el objeto. Una rotacion simple, sobre el eje y, de 10° sera:

```
glRotatef(10, 0.0f, 1.0f, 0.0f);
```

4.3.2.3 Escalado

Una transformacion de escala incrementa el tamaño de nuestro objeto expandiendo todos los vertices a lo largo de los tres ejes por los factores especificados. La funcion `glScale` lleva como parametros la escala en x, y y z, respectivamente. El valor 1.0f es la referencia de la escala, de tal forma que la siguiente linea:

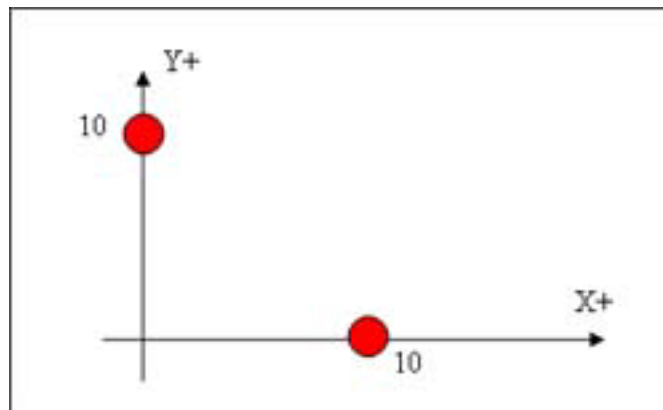
```
glScalef(1.0f, 1.0f, 1.0f);
```

no modificar a el objeto en absoluto. Un valor de 2.0f ser a el doble, y 0.5f ser a la mitad. Por ejemplo, para ensanchar un objeto a lo largo de su eje z, de tal forma que quedase cuatro veces mas “alargado” en este eje, ser a:

```
glScalef(1.0f, 1.0f, 4.0f);
```

4.3.2.4 La matriz identidad

El “problema” del uso de estas funciones surge cuando tenemos mas de un objeto en la escena. Estas funciones tienen efectos acumulativos. Es decir, si queremos preparar una escena como la de la ilustracion 4.3,



Ilustracion 4.3

con una esfera (de radio 3) centrada en (0,10,0) y otra centrada en (10,0,0), escribir amos el siguiente codigo, que es incorrecto:

```
glTranslatef(0.0f, 10.0f, 0.0f);  
glutSolidSphere(3.0f);  
glTranslate(10.0f, 0.0f, 0.0f);  
glutSolidSphere(3.0f);
```

En este codigo, dibujamos primero una esfera en (0,10,0) como quer amos. Pero despues, estamos multiplicando la matriz del modelador que ten amos (que ya estaba transformada para dibujar la primera esfera) por otra matriz de transformacion que nos desplaza 10 unidades hacia la derecha. Por ello la segunda matriz se dibujar a, como se puede ver en la ilustracion 4.4 en (10,10,0), y no en (10,0,0), como pretend amos.

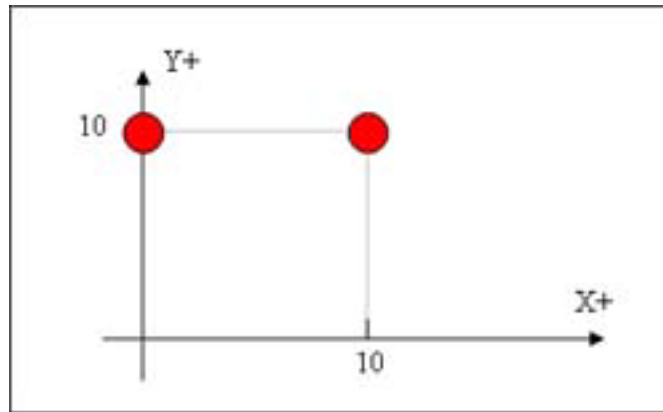


Ilustración 4.4

Para solventar este problema debemos reiniciar la matriz del modelador a un estado mas conocido, en este caso, centrada en el origen de nuestro sistema de coordenadas oculares. Para ello se carga en la matriz del modelador la matriz identidad (una matriz 4x4 llena de ceros excepto en la diagonal, que contiene unos). Esto se consigue gracias a la funcion `glLoadIdentity`, que no lleva parametros. Simplemente carga la matriz identidad en la matriz activa en ese instante. El codigo correcto para el ejemplo anterior quedar a de la siguiente manera:

```
glTranslatef(0.0f, 10.0f, 0.0f);
glutSolidSphere(3.0f);
glLoadIdentity();
glTranslate(10.0f, 0.0f, 0.0f);
glutSolidSphere(3.0f);
```

4.3.2.5 Las pilas de matrices

No siempre es deseable reiniciar la matriz del modelador con la identidad antes de colocar cada objeto. A menudo querremos almacenar el estado actual de transformacion y entonces recuperarlo despues de haber colocado varios objetos. Para ello, `ogl` mantiene una pila de matrices para el modelador (`GL_MODELVIEW`) y otra para la proyeccion (`GL_PROJECTION`). La profundidad var a dependiendo de la plataforma y puede obtenerse mediante las siguientes lineas:

```
glGet(GL_MAX_MODELVIEW_DEPTH);
glGet(GL_MAX_PROJECTION_DEPTH);
```

Por ejemplo, para la implementacion de MicroSoft de ogl, estos valores son de 32 para GL_MODELVIEW y 2 para GL_PROJECTION.

Para meter una matriz en su pila correspondiente se usa la funcion glPushMatrix (sin parametros), y para sacarla glPopMatrix (sin parametros tampoco).

4.3.3 La matriz de proyeccion

La matriz de proyeccion especifica el tamano y la forma de nuestro volumen de visualizacion. El volumen de visualizacion es aquel volumen cuyo contenido es el que representaremos en pantalla. Este esta delimitado por una serie de planos de trabajo, que lo delimitan. De estos planos, los mas importantes son los planos de corte, que son los que nos acotan el volumen de visualizacion por delante y por detras. En el plano mas cercano a la camara (znear), es donde se proyecta la escena para luego pasarla a la pantalla. Todo lo que este mas adelante del plano de corte mas alejado de la camara (zfar) no se representa.

Veremos los distintos volúmenes de visualizacion de las dos proyecciones mas usadas: ortograficas y perspectivas.

Cuando cargamos la identidad en el matriz de proyeccion, la diagonal de unos especifica que los planos de trabajo se extienden desde el origen hasta los unos positivos en todas las direcciones. Como antes, veremos ahora que existen funciones de alto nivel que nos facilitan todo el proceso.

4.3.3.1 Proyecciones ortograficas

Una proyeccion ortografica es cuadrada en todas sus caras. Esto produce una proyeccion paralela, util para aplicaciones de tipo CAD o dibujos arquitectonicos, o tambien para tomar medidas, ya que las dimensiones de lo que representan no se ven alteradas por la proyeccion.

Una aproximacion menos tecnica pero mas comprensible de esta proyeccion es imaginar que tenemos un objeto fabricado con un material deformable, y lo aplastamos literalmente como una pared. Obtendremos el mismo objeto, pero plano, liso. Pues eso es lo que veremos por pantalla.

Para definir la matriz de proyeccion ortografica y multiplicarla por la matriz activa (que deber a ser en ese momento la de proyeccion, GL_PROJECTION), utilizamos la funcion glOrtho, que se define de la siguiente forma

```
glOrtho(limiteIzquierdo, limiteDerecho, limiteAbajo, limiteArriba, znear, zfar)
```

siendo todos flotantes. Con esto simplemente acotamos lo que sera nuestro volumen de visualizacion (un cubo).

Por ejemplo, la ilustracion 4.5 es un render de un coche con proyeccion ortografica, visto desde delante.



Ilustracion 4.5

El codigo utilizado para esta proyeccion ha sido

```
glOrtho(-0.5f, 0.5f, -0.5f, 0.5f, 0.01f, 20.0f);
```

En la siguiente seccion, se podra apreciar la diferencia usando una proyeccion perspectiva.

4.3.3.2 Proyecciones perspectivas

Una proyeccion en perspectiva ejecuta una division en perspectiva para reducir y estirar los objetos mas alejados del observador. Es importante saber que las medidas de la proyeccion no tienen por que coincidir con las del objeto real, ya que han sido deformadas.

El volumen de visualización creado por una perspectiva se llama frustum. Un frustum es una sección piramidal, vista desde la parte afilada hasta la base (ilustración 4.6).

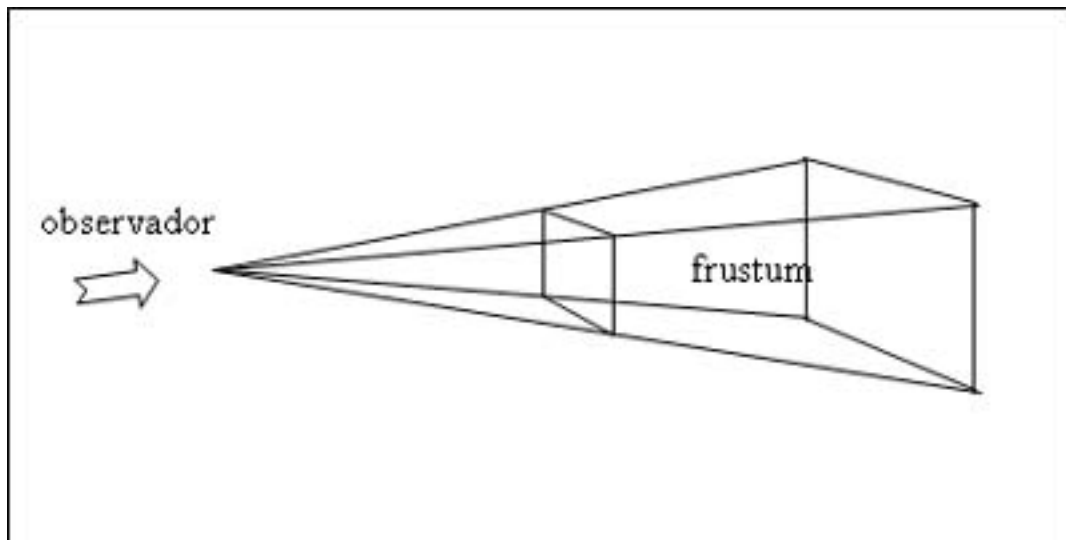
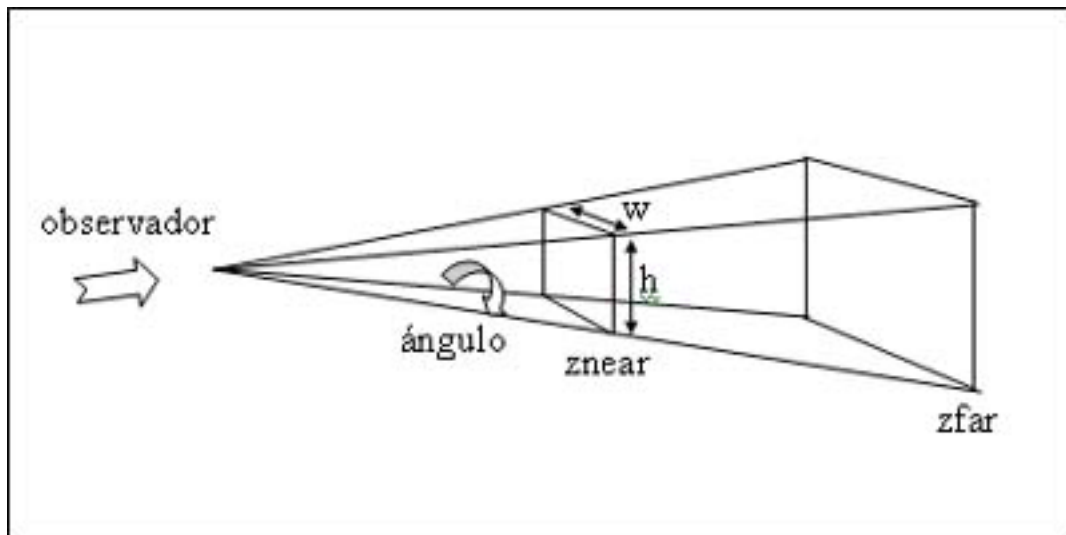


Ilustración 4.6

Podemos definir esta proyección utilizando la función `glFrustum`. Pero existe otra función de la librería GLU llamada `gluPerspective` que hace el proceso más sencillo. Se define de la siguiente forma:

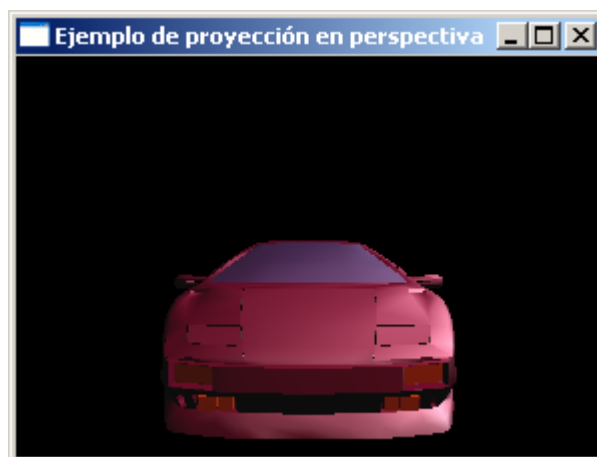
```
Void gluPerspective(angulo, aspecto, znear, zfar);
```

Los parámetros de `gluPerspective` son flotantes, y definen las características mostradas en la ilustración 4.7: el ángulo para el campo de visión en sentido vertical, el aspecto es la relación entre la altura(h) y la anchura(w) y las distancias `znear` y `zfar` de los planos que acotan el frustum.



Ilustracion 4.7

La ilustracion 4.8 muestra la escena del coche de la seccion anterior, esta vez con una proyeccion en perspectiva:



Ilustracion 4.8

El código utilizado para definir la proyección ha sido

```
gluPerspective(45.0f,(GLfloat)(width/height),0.01f,100.0f);
```

Usamos 45° de angulo, la relacion entre el ancho y alto de la pantalla (width y height son el ancho y alto actual de la ventana), y las distancias a los planos de corte znear y zfar son 0.01 y 100 respectivamente.

4.4 Ejemplo: una escena simple

4.4.1Codigo

El siguiente codigo es un programa que usa OpenGL mediante la libreria GLUT. Primero se lista el codigo completo, y luego se comenta linea por linea. Con el aplicaremos lo aprendido en el anterior capitulo y el presente, ademas de nuevas funciones de la libreria GLUT.

La escena consiste en un cubo de colores girando sobre si misma, y una esfera blanca de alambre (mas conocido como “wired”, consiste en no dibujar las caras del objeto, si no solamente las lineas que unen sus vertices, dando la sensacion de ser una figura de alambre) girando alrededor del cubo. Podemos utilizar las teclas ‘o’ y ‘p’ para cambiar el tipo de proyeccion, ortografica y perspectiva, respectivamente. Con la tecla ‘esc’ se abandona el programa.

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

GLfloat anguloCuboX = 0.0f;
GLfloat anguloCuboY = 0.0f;
GLfloat anguloEsfera = 0.0f;

Glint ancho, alto;

int hazPerspectiva = 0;

void reshape(int width, int height)
{
    glViewport(0, 0, width, height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    if(hazPerspectiva)
        gluPerspective(60.0f, (GLfloat)width/(GLfloat)height, 1.0f, 20.0f);
```

```

else
    glOrtho(-4, 4, -4, 4, 1, 10);

glMatrixMode(GL_MODELVIEW);

ancho = width;
alto = height;
}

void drawCube(void)
{
    glColor3f(1.0f, 0.0f, 0.0f);
    glBegin(GL_QUADS); //cara frontal
    glVertex3f(-1.0f, -1.0f, 1.0f);
    glVertex3f( 1.0f, -1.0f, 1.0f);
    glVertex3f( 1.0f, 1.0f, 1.0f);
    glVertex3f(-1.0f, 1.0f, 1.0f);
    glEnd();

    glColor3f(0.0f, 1.0f, 0.0f);
    glBegin(GL_QUADS); //cara trasera
    glVertex3f( 1.0f, -1.0f, -1.0f);
    glVertex3f(-1.0f, -1.0f, -1.0f);
    glVertex3f(-1.0f, 1.0f, -1.0f);
    glVertex3f( 1.0f, 1.0f, -1.0f);
    glEnd();

    glColor3f(0.0f, 0.0f, 1.0f);
    glBegin(GL_QUADS); //cara lateral izq
    glVertex3f(-1.0f, -1.0f, -1.0f);
    glVertex3f(-1.0f, -1.0f, 1.0f);
    glVertex3f(-1.0f, 1.0f, 1.0f);
    glVertex3f(-1.0f, 1.0f, -1.0f);
    glEnd();

    glColor3f(1.0f, 1.0f, 0.0f);
    glBegin(GL_QUADS); //cara lateral dcha
    glVertex3f( 1.0f, -1.0f, 1.0f);
    glVertex3f( 1.0f, -1.0f, -1.0f);
    glVertex3f( 1.0f, 1.0f, -1.0f);

```

```

glVertex3f( 1.0f, 1.0f, 1.0f);
glEnd();

glColor3f(0.0f, 1.0f, 1.0f);
glBegin(GL_QUADS); //cara arriba
glVertex3f(-1.0f, 1.0f, 1.0f);
glVertex3f( 1.0f, 1.0f, 1.0f);
glVertex3f( 1.0f, 1.0f, -1.0f);
glVertex3f(-1.0f, 1.0f, -1.0f);
glEnd();

glColor3f(1.0f, 0.0f, 1.0f);
glBegin(GL_QUADS); //cara abajo
glVertex3f( 1.0f, -1.0f, -1.0f);
glVertex3f( 1.0f, -1.0f, 1.0f);
glVertex3f(-1.0f, -1.0f, 1.0f);
glVertex3f(-1.0f, -1.0f, -1.0f);
glEnd();
}

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

    glLoadIdentity();

    glTranslatef(0.0f, 0.0f,- 5.0f);

    glRotatef(anguloCuboX, 1.0f, 0.0f, 0.0f);
    glRotatef(anguloCuboY, 0.0f, 1.0f, 0.0f);

    drawCube();

    glLoadIdentity();

    glTranslatef(0.0f, 0.0f,- 5.0f);
    glRotatef(anguloEsfera, 0.0f, 1.0f, 0.0f);
    glTranslatef(3.0f, 0.0f, 0.0f);

```

```

    glColor3f(1.0f, 1.0f, 1.0f);
    glutWireSphere(0.5f, 8, 8);

    glFlush();
    glutSwapBuffers();

    anguloCuboX+=0.1f;
    anguloCuboY+=0.1f;
    anguloEsfera+=0.2f;
}

void init()
{
    glClearColor(0,0,0,0);
    glEnable(GL_DEPTH_TEST);
    ancho = 400;
    alto = 400;
}

void idle()
{
    display();
}

void keyboard(unsigned char key, int x, int y)
{
    switch(key)
    {
        case 'p':
        case 'P':
            hazPerspectiva=1;
            reshape(ancho,alto);
            break;

        case 'o':
        case 'O':
            hazPerspectiva=0;
            reshape(ancho,alto);
            break;
    }
}

```

```

        case 27: // escape
            exit(0);
            break;

    }
}

int main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
    glutInitWindowPosition(100, 100);
    glutInitWindowSize(ancho, alto);
    glutCreateWindow("Cubo 1");
    init();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutIdleFunc(idle);
    glutKeyboardFunc(keyboard);
    glutMainLoop();
    return 0;
}

```

4.4.2 Analisis del codigo

Pasamos ahora a comentar el codigo. Se reproduce el codigo entero, pero iremos haciendo pausas en las funciones que no se hayan explicado en el capitulo 2.

Empezaremos por la funcion main():

```

int main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);

```

En esta ocasion, utilizamos GLUT_DOUBLE en vez de GLUT_SIMPLE. Esto hace posible la utilizacion de la tecnica de “double buffer”, con la utilizamos dos buffers para

ir sacando frames por pantalla, en vez de uno. Con esto conseguimos una mayor fluidez en escenas animadas.

```
glutInitWindowPosition(100, 100);  
glutInitWindowSize(ancho, alto);  
glutCreateWindow("Cubo 1");  
init();  
glutDisplayFunc(display);  
glutReshapeFunc(reshape);  
glutIdleFunc(idle);
```

Aquí añadimos una función callback nueva, la del idle. Esta función es llamada cuando la ventana está en idle, es decir, no se está realizando ninguna acción sobre ella. Normalmente se usa la misma función que la de dibujo, como en este ejemplo (la función idle() que hemos definido simplemente llama a la función redraw()).

```
glutKeyboardFunc(keyboard);
```

Otro callback nuevo, usado para capturar y manejar el teclado cuando nuestra ventana está activa. La definición de esta función ha de ser de la forma

```
void teclado(unsigned char tecla, int x, int y)
```

donde “tecla” es el código ASCII de la tecla pulsada en cada momento y “x” y “y” las coordenadas del ratón en ese instante.

```
glutMainLoop();  
return 0;  
}
```

Ahora veremos la función drawCube, que utilizamos para crear un cubo. El cubo, como veremos ahora, está centrado en el origen y el lado es igual a 2.

```
void drawCube(void)  
{  
    glColor3f(1.0f, 0.0f, 0.0f);  
    glBegin(GL_QUADS); //cara frontal (C0)  
    glVertex3f(-1.0f, -1.0f, 1.0f);  
    glVertex3f( 1.0f, -1.0f, 1.0f);
```

```
glVertex3f( 1.0f, 1.0f, 1.0f);  
glVertex3f(-1.0f, 1.0f, 1.0f);  
glEnd();
```

Con estas líneas creamos un polígono cuadrado, ensamblando cuatro vertices. Además hemos utilizado previamente glColor para asignarle el color rojo. Es importante observar que utilizamos el sentido antihorario, para que la normal del polígono vaya hacia fuera (recordemos que, por defecto, el sentido antihorario es el que define la normal hacia fuera). Este cuadrado conformara la cara del cubo frontal, la mas cercana a nosotros.

```
glColor3f(0.0f, 1.0f, 0.0f);  
glBegin(GL_QUADS); //cara trasera (C1)  
glVertex3f( 1.0f, -1.0f, -1.0f);  
glVertex3f(-1.0f, -1.0f, -1.0f);  
glVertex3f(-1.0f, 1.0f, -1.0f);  
glVertex3f( 1.0f, 1.0f, -1.0f);  
glEnd();
```

Definimos aquí la cara trasera, la mas alejada de nosotros, de color verde.

```
glColor3f(0.0f, 0.0f, 1.0f);  
glBegin(GL_QUADS); //cara lateral izq (C2)  
glVertex3f(-1.0f, -1.0f, -1.0f);  
glVertex3f(-1.0f, -1.0f, 1.0f);  
glVertex3f(-1.0f, 1.0f, 1.0f);  
glVertex3f(-1.0f, 1.0f, -1.0f);  
glEnd();
```

Cara lateral izquierda, de color azul.

```
glColor3f(1.0f, 1.0f, 0.0f);  
glBegin(GL_QUADS); //cara lateral dcha (C3)  
glVertex3f( 1.0f, -1.0f, 1.0f);  
glVertex3f( 1.0f, -1.0f, -1.0f);  
glVertex3f( 1.0f, 1.0f, -1.0f);  
glVertex3f( 1.0f, 1.0f, 1.0f);  
glEnd();
```

Cara lateral derecha, de color amarillo, por ser mezcla de rojo y verde.

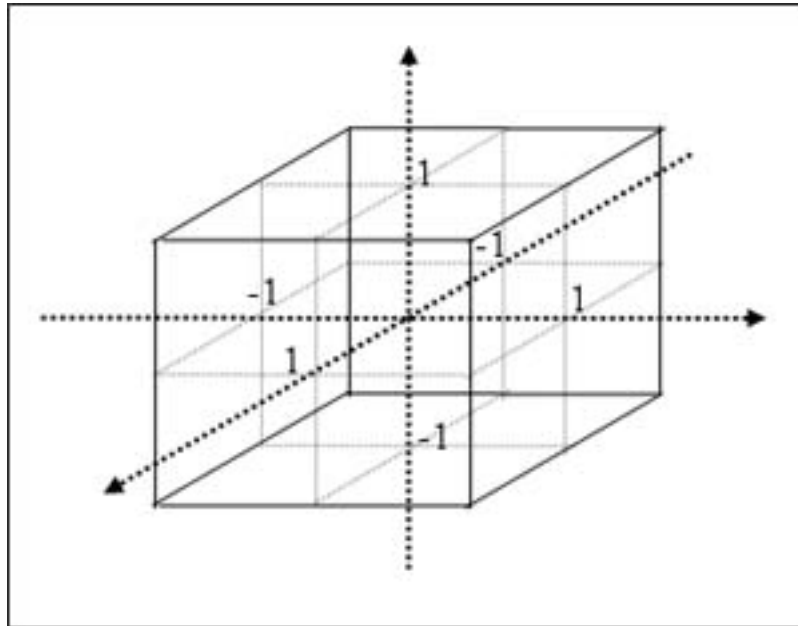
```
glColor3f(0.0f, 1.0f, 1.0f);  
glBegin(GL_QUADS); //cara arriba (C4)  
glVertex3f(-1.0f, 1.0f, 1.0f);  
glVertex3f( 1.0f, 1.0f, 1.0f);  
glVertex3f( 1.0f, 1.0f,- 1.0f);  
glVertex3f(-1.0f, 1.0f, -1.0f);  
glEnd();
```

Esta sera la cara de arriba, de color azul claro.

```
glColor3f(1.0f, 0.0f, 1.0f);  
glBegin(GL_QUADS); //cara abajo (C5)  
glVertex3f( 1.0f, -1.0f, -1.0f);  
glVertex3f( 1.0f, -1.0f, 1.0f);  
glVertex3f(-1.0f, -1.0f, 1.0f);  
glVertex3f(-1.0f, -1.0f, -1.0f);  
glEnd();  
}
```

Y, finalmente, la cara de abajo, de color violeta.

El cubo que hemos definido, se puede ver de una forma mas grafica en la ilustracion 4.9. Sobre los colores hablaremos en el proximo capitulo, ya que, a primera vista puede resultar poco coherente su conformacion.



Ilustracion 4.9

Vamos ahora con el contenido de los callbacks:

Primero vemos el `init()`, que como hemos dicho, no es un callback, simplemente activa los estados iniciales de ogl que queramos. En este caso, activamos un nuevo estado:

```
glEnable(GL_DEPTH_TEST);
```

para que haga el test de profundidad, utilizando el z-buffer. Además asignamos el ancho y alto de la ventana.

El callback `reshape`, llamado cuando la ventana se redimensiona:

```
void reshape(int width, int height)
{
    glViewport(0, 0, width, height);
    glMatrixMode(GL_PROJECTION);
```

Hacemos que la matriz activa sea la de proyección, puesto que es aquí donde definiremos el tipo de proyección a usar.

```
glLoadIdentity();
```

Cargamos en la matriz de proyeccion la identidad, para resetearla y poder trabajar sobre ella.

```
if(hazPerspectiva)
    gluPerspective(60.0f, (GLfloat)width/(GLfloat)height, 1.0f, 20.0f);
else
    glOrtho(-4, 4, -4, 4, 1, 10);
```

La variable “hazPerspectiva”, definida como un entero, hace las funciones de un booleano. Si su valor es cero, hace una proyeccion ortonormal. Para ello usamos la funcion glOrtho, definiendo los l mites de los planos de trabajo. Si la variable esta a uno, hacemos una perspectiva con gluPerspective.

```
glMatrixMode(GL_MODELVIEW);
```

Aqui reactivamos la matriz del modelador, que es con la que trabajamos habitualmente.

```
ancho = width;
alto = height;
}
```

Por ultimo, actualizamos las variables “ancho” y “alto”, con los valores actuales de la ventana.

Veamos ahora el callback del teclado:

```
void keyboard(unsigned char key, int x, int y)
{
    switch(key)
    {
        case 'p':
        case 'P':
            hazPerspectiva=1;
            reshape(ancho,alto);
            break;
    }
```

Si pulsamos la tecla p (en minuscula o mayuscula) activamos la variable “hazPerspectiva”. Para que se efectue el cambio, llamamos a la funcion reshape, de una

manera un tanto “artificial”, ya que el proposito de esta funcion es ajustar los parametros de la proyeccion ante un cambio de la dimension de la ventana. En este caso, la ventana no se ha redimensionado, y llamamos manualmente a la funcion con los valores actuales, para realizar el cambio de proyeccion.

```
case 'o':  
case 'O':  
    hazPerspectiva=0;  
    reshape(ancho,alto);  
    break;
```

Si la tecla pulsada es la ‘o’, usamos, igual que antes, la proyeccion ortografica.

```
case 27: // escape  
    exit(0);  
    break;  
}  
}
```

En caso de que la tecla sea “ESC” (su codigo ASCII es el 27), salimos del programa.

El ultimo callback a analizar es el que dibuja la escena:

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

Aqui limpiamos el frame buffer, que es el buffer donde dibujamos, y el z-buffer, utilizado para el test de profundidad.

```
glLoadIdentity();
```

Resetear la matriz del modelador.

```
glTranslatef(0.0f, 0.0f, -5.0f);  
  
glRotatef(anguloCuboX, 1.0f, 0.0f, 0.0f);  
glRotatef(anguloCuboY, 0.0f, 1.0f, 0.0f);
```

Con estas tres líneas cargamos en la matriz del modelador una transformación de la siguiente forma: primero trasladamos lo que vayamos a dibujar (que será el cubo) cinco unidades hacia atrás, para ponerlo delante de nosotros. Luego lo giramos sobre el eje x y el eje y el número de grados que marquen las variables “anguloCuboX” y “anguloCuboY” respectivamente. Estas variables se van incrementando en cada frame, como veremos pocas líneas más abajo.

```
drawCube();
```

Llamamos a la función que hemos creado para dibujar un cubo. Al hacerlo, se ejecutarán todas las funciones que crean las caras, centrando el cubo en el origen. Pero inmediatamente se ven modificadas las posiciones de los vértices por la matriz de transformación cargada en el modelador, dejando así el cubo donde nos interesa.

```
glLoadIdentity();
```

Reseteamos la matriz del modelador, ya que el cubo ya está situado en donde queremos, y necesitamos una nueva matriz de transformación para poner la esfera.

```
glTranslatef(0.0f, 0.0f, -5.0f);  
glRotatef(anguloEsfera, 0.0f, 1.0f, 0.0f);  
glTranslatef(3.0f, 0.0f, 0.0f);
```

Con estas nuevas tres líneas, creamos la matriz de transformación para la esfera. Primero la trasladamos 5 unidades hacia atrás, de forma que queda centrada en el mismo sitio que el cubo. Ahora rotamos el sistema de coordenadas tantos grados como contenga la variable “anguloEsfera”, sobre el eje y. Ahora que tenemos rotado el sistema de coordenadas de la esfera, solo hay que desplazar la esfera en el eje x, de forma que según incrementemos “anguloEsfera”, esta vaya describiendo una circunferencia, de radio el número de unidades que la desplazamos (en este caso 3).

```
glColor3f(1.0f, 1.0f, 1.0f);  
glutWireSphere(0.5f, 8, 8);
```

Activamos el color que deseemos para la esfera, blanco en este caso, y la dibujamos. Para ello, la librería GLUT proporciona una serie de funciones de alto nivel con objetos comunes, como cubos, cilindros, esferas, etc. Además GLUT permite dibujarlas como “Solid” o como “Wire”, es decir, como un objeto sólido, conformado por polígonos, o como un objeto “de alambre”. Los parámetros de glutWireSphere (y también los de glutSolidSphere) son el radio, el número de líneas de longitud y el número de líneas de latitud.

Para dibujar el cubo pod amos haber utilizado la funcion `glutSolidCube`, que lleva como parametro la longitud del lado, pero se ha preferido crear una funcion que lo haga, con fines didacticos.

```
glFlush();
```

Hacemos que todos los comandos de ogl que esten en espera se ejecuten.

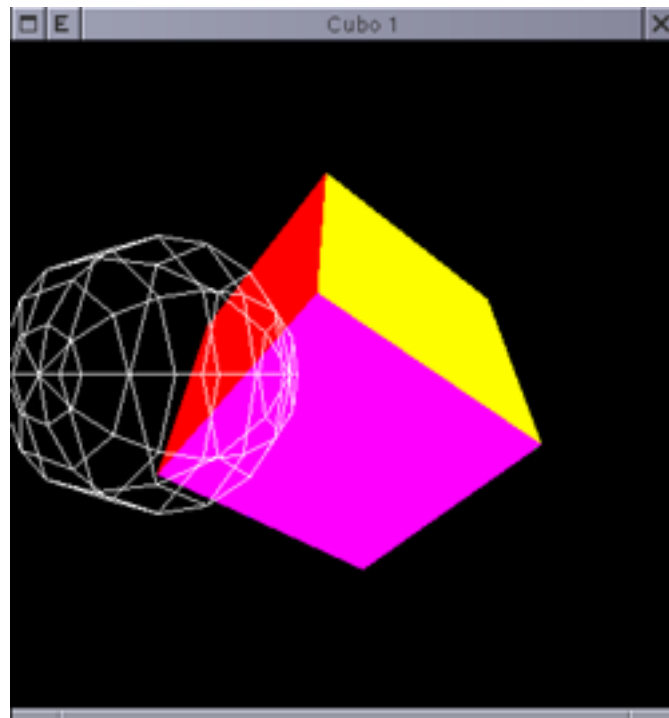
```
glutSwapBuffers();
```

Al utilizar la tecnica de doble buffer, tenemos que llamar siempre a esta funcion al final del pintado de cada frame, para que vuelque de un buffer a otro el frame correspondiente.

```
    anguloCuboX+=0.1f;  
    anguloCuboY+=0.1f;  
    anguloEsfera+=0.2f;  
}
```

Por ultimo, incrementamos todos los angulos que estamos usando.

En la ilustracion 4.10vemos un frame de la salida del programa.



Ilustracion 4.10