

GRÁFICOS POR COMPUTADOR

Práctica principal

Curso 2014-15

Objetivos

Acelerar la representación gráfica con listas de visualización (*display lists*).

Conocer las funciones OpenGL de manejo del ratón.

Manejar y controlar el tiempo en animaciones no basadas en eventos.

Dibujar elementos ortográficos sobre una vista en perspectiva (HUD).

Dibujar texto con OpenGL.

Aplicar texturas a los objetos.

Reproducir sonidos y música.

Desarrollar el motor gráfico subyacente de una aplicación gráfica de tiempo real.

Listas de visualización (*display lists*)

En todo escenario 3D hay un conjunto de objetos estáticos que no se van a mover nunca (suelo, paredes, decoración...). Generar y calcular la posición global de estos objetos en cada iteración de la función `display()` carga en exceso al *pipeline* gráfico.

Las listas de visualización nos permiten crear estos objetos una única vez en la función `init()` y utilizarla varias veces en la función `display()`, acelerando así la representación gráfica.

Visualización inmediata frente a listas de visualización

La visualización inmediata manda las primitivas gráficas al *pipeline* gráfico y se visualizan en el acto. No se usa memoria OpenGL para entidades gráficas.

Por el contrario, las listas de visualización sitúan las primitivas en memoria OpenGL (normalmente en la tarjeta gráfica), donde se mantienen hasta que sean llamadas para ser visualizadas.

Creación de una lista de visualización

En la función `init()` se crean con la función `GLuint glGenLists(GLsizei range)` donde `range` es el número de listas contiguas que se van a crear. Devuelve el índice a la primera lista.

Cada grupo de comandos OpenGL que definen objetos estáticos debe incluirse entre la función `void glNewList(GLuint list, GLenum mode)` y la función `void glEndList(void)`, donde `list` es el índice devuelto por `glGenLists()` y `mode` puede tomar dos valores:

- `GL_COMPILE`: crea la lista de comandos, pero no los ejecuta (recomendado).
- `GL_COMPILE_AND_EXECUTE`: crea la lista de comandos y los ejecuta. Este modo es más lento.

Representación de la lista de visualización

En la función `display()` se hacen las llamadas a las listas creadas previamente en `init()` con la función `void glCallList(GLuint list)`, donde `list` es el índice devuelto por `glGenLists()`.

Ejemplo:

```
GLuint teteraDL;
...
void init(void)
{
    ...
    teteraDL = glGenLists(1);
    glNewList(teteraDL, GL_COMPILE);
        glRotatef(90,0,0,1);
        glRotatef(90,1,0,0);
        glutSolidTeapot(2);
    glEndList();
}

void display(void)
{
    ...
    glCallList(teteraDL);
}
```

Uso avanzado del teclado

Para

Manejo de ratón

Dependiendo de la aplicación gráfica puede surgir la necesidad de cambiar el cursor del ratón e incluso eliminarlo. Con la función `void glutSetCursor(int cursor)` se cambia la imagen del cursor al valor del parámetro `cursor` (`GLUT_CURSOR_NONE`, `GLUT_CURSOR_CROSSHAIR`, `GLUT_CURSOR_WAIT`, `GLUT_CURSOR_TEXT`,...).

La posición del ratón está direccionada respecto a la esquina superior izquierda. La función `void glutWarpPointer(int x, int y)` coloca el cursor del ratón en la posición (x, y) especificada. Esta función es útil si queremos mover el observador con el ratón y que no esté limitado por las dimensiones de la pantalla.

Las funciones *callback*, que vimos al principio del tema de OpenGL, `glutMouseFunc(...)`, `glutMotionFunc(...)` y `glutPassiveMotionFunc(...)` permiten establecer las funciones que manejan el ratón cuando se pulsa algunos de sus botones, cuando se mueve el ratón con algún botón pulsado o cuando se mueve el ratón sin pulsar ningún botón, respectivamente.

Control del tiempo para animaciones

Los objetos móviles de la escena se tienen que seguir moviendo aun con la ausencia de eventos. Cuando la aplicación está ociosa o inactiva, es decir, sin recibir ningún evento, el control de la ejecución se puede pasar a la función asociada con la función *callback* `void glutIdleFunc(void (*)(void))`.

```
float angulo;

void main(...)
{
    ...
    glutIdleFunc(idle);
    ...
}

void idle(void)
{
    angulo += 0.5;
    glutPostRedisplay();
}
```

Velocidad de la animación

La velocidad de la animación, que se mide en fotogramas por segundo (*frames* por segundo o fps), puede ser excesivamente alta. Además su valor dependerá de la escena que se esté mostrando, de la plataforma *hardware* sobre la que se esté ejecutando la aplicación o de que haya otras aplicaciones en ejecución.

Para conocer los fotogramas por segundo a los que funciona una animación se puede utilizar la función `glutGet(GLUT_ELAPSED_TIME)`, la cual devuelve el número de milisegundos desde que se inició la librería GLUT con `glutInit()`.

El siguiente código sirve para calcular los fps de una animación:

```
int frame=0, time=0, timebase=0;
...

void idle(void)
{
    char s[30];

    frame++;
    time=glutGet(GLUT_ELAPSED_TIME);
    if (time - timebase > 1000)
    {
        sprintf(s,"FPS:%4.2f",frame*1000.0/(time-timebase));
        glutSetWindowTitle(s); // no valdría a pantalla completa
        timebase = time;
        frame = 0;
    }
}
```

Si queremos que nuestra aplicación tenga la misma velocidad independientemente del *hardware* en el que se ejecute, es necesario establecer un intervalo fijo de fotogramas por segundo y ceñirnos a él, esperando el tiempo que sobre.

```
#define TICK_INTERVAL 20 // intervalos de 20 ms suponen una tasa de 50 fps

GLuint next_time;

// Función que calcula cuántos milisegundos quedan para que empiece
// el próximo intervalo
GLuint time_left(void)
{
    GLuint now;

    now = glutGet(GLUT_ELAPSED_TIME);
    if (next_time <= now)
    {
        return 0;
    }
    else
    {
        return next_time - now;
    }
}

void init(void)
{
    ...
    next_time = glutGet(GLUT_ELAPSED_TIME) + TICK_INTERVAL;
}

void display(void)
{
    ...
    Sleep(time_left()); // esta función dependerá del S.O.
    next_time += TICK_INTERVAL;
}
```

Proyección ortográfica sobre perspectiva (HUD)

Dependiendo de dónde se sitúen las funciones primitivas con respecto a la matriz de proyección, éstas pueden verse superpuestas sobre la escena tridimensional en perspectiva. Esto resulta de especial utilidad para colocar marcadores, paneles de control, información de texto, etc. Este tipo de paneles reciben el nombre de HUD (Head Up Display), que podemos traducir como panel frontal de datos.

El panel es una superficie 2D cuyo origen se sitúa en la esquina superior izquierda, con el eje X positivo hacia la derecha y el eje Y positivo hacia abajo. Para dibujar en él, hay que cambiar previamente el modo de proyección a ortográfica y reestablecer el modo de perspectiva posteriormente. Para ello, se utilizan las dos funciones siguientes:

```
void setOrthographicProjection()
{
    glDisable(GL_LIGHTING);
    glDisable(GL_DEPTH_TEST);

    glMatrixMode(GL_PROJECTION);
    glPushMatrix();
    glLoadIdentity();
    gluOrtho2D(0, w, h, 0);
    glMatrixMode(GL_MODELVIEW);
}
```

```
void resetPerspectiveProjection()
{
    glMatrixMode(GL_PROJECTION);
    glPopMatrix();
    glMatrixMode(GL_MODELVIEW);

    glEnable(GL_LIGHTING);
    glEnable(GL_DEPTH_TEST);
}
```

Como se puede observar, es necesario desactivar la iluminación y el test de profundidad para ver los objetos 2D del panel frontal correctamente.

Las llamadas a estas funciones deben hacerse en la función `display()` y **se recomienda que se coloque al final de la misma**.

```
void display()
{
    ... // ---> aquí se dibujará toda la escena 3D
    setOrthographicProjection();
    // Aquí puede ir cualquier panel de mandos o de información
    glBegin(GL_LINES);
        glColor3f(1.0f,1.0f,1.0f);
        glVertex2f(w/2.0,h/2.0-10); // w,h son las dimensiones de la pantalla
        glVertex2f(w/2.0,h/2.0+10);
        glVertex2f(w/2.0-10,h/2.0);
        glVertex2f(w/2.0+10,h/2.0);
    glEnd();
    resetPerspectiveProjection();

    glutSwapBuffers();
}
```

Dibujar texto

OpenGL no tiene funciones para dibujar texto, sólo aporta un conjunto de directivas muy arcaicas para dibujar *bitmaps*, de modo que el programador debe crearse su propio conjunto de *bitmaps* para cada uno de los caracteres.

GLUT facilita la labor definiendo un conjunto de tipos de letra que podemos utilizar con la función `void glutBitmapCharacter(void *font, int character)`. En el parámetro `font` podemos indicar el tipo de letra eligiendo entre las siguientes opciones:

- GLUT_BITMAP_9_BY_15
- GLUT_BITMAP_8_BY_13
- GLUT_BITMAP_TIMES_ROMAN_10
- GLUT_BITMAP_TIMES_ROMAN_24
- GLUT_BITMAP_HELVETICA_10
- GLUT_BITMAP_HELVETICA_12
- GLUT_BITMAP_HELVETICA_18

En el parámetro `character` indicamos el carácter que se va representar. De esta manera, para dibujar una cadena de caracteres tenemos que hacer una llamada a `glutBitmapCharacter` por cada uno de los caracteres de la cadena. La siguiente función realiza esta operación seleccionando el tipo de letra Helvética de tamaño 10.

```
void displayString(char *s)
{
    for (int i = 0; i < strlen (s); i++)
    {
        glutBitmapCharacter (GLUT_BITMAP_HELVETICA_10, s[i]);
    }
}
```

Con la función `void glRasterPos2i(GLint x, GLint y)` podemos especificar las coordenadas 2D del píxel en el que vamos a dibujar el *bitmap* del texto. Hay que tener en cuenta que los píxeles se direccionan desde la esquina superior izquierda y que el *bitmap* se dibuja hacia arriba y hacia la izquierda. Es decir, si se especifica dibujar en la posición (0,0), justo en la esquina superior izquierda, el carácter no se verá porque se sale de la pantalla.

Con la combinación de estas funciones y las del apartado sobre *Proyección ortográfica sobre perspectiva*, podemos representar un texto en pantalla en la función `display()`.

```
void display()
{
    ...

    setOrthographicProjection();

    glPushMatrix();
    glLoadIdentity();
    glColor3f(1.0f,1.0f,1.0f);
    sprintf(label,"Prueba de texto");
    glRasterPos2i(30,30);
    displayString(label);
    glPopMatrix();

    resetPerspectiveProjection();

    ...
}
```

Como se puede comprobar, `displayString()`, `setOrthographicProjection()` y `resetPerspectiveProjection()` no son funciones de OpenGL sino definidas por el programador.

Texturas en OpenGL

Para utilizar texturas en OpenGL es necesario realizar los siguientes pasos:

- Cargar la imagen en memoria
- Generar un identificador de textura
- Configurar la textura
- Asociarla a las primitivas

Cargar la imagen

Es necesario utilizar una librería externa que permita leer el formato de imagen adecuado. Hay muchas disponibles:

- SDL_image
- FreeImage
- Corona
- SOIL
- DevIL
- ...

En la lectura de la imagen debemos obtener al menos el **ancho** y **alto** de la imagen y un **array** con los píxeles. También puede ser útil obtener el nº de bytes por píxel (bpp) y el formato de la imagen.

La librería DevIL tiene definidas varias funciones para cargar, guardar, convertir, manipular y mostrar imágenes. Permite trabajar con una gran variedad de formatos de imagen:

- BMP, JPG, PNG, TGA, TIF, PCX, RAW
- SGI, CUT, PAL, COL, PCD, PIC, PBM, PGM, PPM

Para utilizar la librería DevIL debemos incluir `ilut.h` en el código.

```
#include <IL/ilut.h>
```

Hay que añadir al linker: `-lDevIL -lILU -lILUT`

En la función `init()` se debe llamar a la función que carga las imágenes y las convierte en texturas. En la función `main()` hay que inicializar las librerías de DevIL.

```
ilInit();
iluInit();
ilutRenderer(ILUT_OPENGL);
```

Para leer la imagen se utiliza la función `ilLoadImage()`:

```
ILuint texid;
ilGenImages(1, &texid);
ilBindImage(texid);
ilLoadImage((const ILstring)"imagen.jpg");
ilConvertImage(IL_RGBA, IL_UNSIGNED_BYTE);

ancho    = ilGetInteger(IL_IMAGE_WIDTH);
alto     = ilGetInteger(IL_IMAGE_HEIGHT);
formato  = ilGetInteger(IL_IMAGE_FORMAT);
bpp      = ilGetInteger(IL_IMAGE_BPP);
data     = ilGetData();

ilDeleteImages(1, &texid);
```

Generar un identificador

En la función `init()` hay que activar el manejo de texturas con:

```
glEnable(GL_TEXTURE_2D);
```

Hay 4 tipos básicos de texturas:

- `GL_TEXTURE_1D`, para texturas de 1 dimensión.
- `GL_TEXTURE_2D`, para texturas de 2 dimensiones.
- `GL_TEXTURE_3D`, para texturas de 3 dimensiones.
- `GL_TEXTURE_CUBE_MAP`, para mapas de texturas de cubo.

La función que se utiliza para generar un identificador de textura es `glGenTextures()`.

```
void glGenTextures(GLsizei n, GLuint *texID)
```

Devuelve `n` identificadores de memoria para texturas sin usar en el *array* pasado en `texID`.

Configurar la textura

El identificador generado se utiliza en la función `glBindTexture()`:

```
glBindTexture(GL_TEXTURE_2D, texID);
```


A partir de aquí todas las funciones y parámetros que utilicemos se refieren a esa textura concreta.

Con `glTexImage2D()` le pasamos la imagen y los parámetros a la textura:

```
void glTexImage2D(GLenum target, GLint level, GLint components,
                  GLsizei width, GLsizei height, GLint border,
                  GLenum format, GLenum type, void *data);
```

- `target`: especifica el tipo básico de textura `GL_TEXTURE_2D`.
- `level`: indica el nivel de detalle. El nivel 0 especifica el estado inicial. El valor `n` especifica la `n`-ésima reducción de la imagen por medio del mipmap.
- `components`: define el número de componentes de color de la textura, `GL_RGBA`.
- `width`: es el ancho de la imagen (potencia de 2).
- `height`: es la altura de la imagen (potencia de 2).
- `border`: define el ancho del borde. Debe estar entre 0 y 1. Normalmente es 0.
- `format`: formato de los datos de los píxeles. Los valores aceptados son: `GL_COLOR_INDEX`, `GL_RED`, `GL_GREEN`, `GL_BLUE`, `GL_ALPHA`, `GL_RGB`, **`GL_RGBA`**, `GL_LUMINANCE` y `GL_LUMINANCE_ALPHA`.
- `type`: representa el tipo de datos de los píxeles. Pueden ser: **`GL_UNSIGNED_BYTE`**, `GL_BYTE`, `GL_BITMAP`, `GL_UNSIGNED_SHORT`, `GL_SHORT`, `GL_UNSIGNED_INT`, `GL_INT` y `GL_FLOAT`.
- `data`: puntero a los datos de la imagen en memoria.

La función `glTexParameterf()` permite ajustar diversos parámetros sobre cómo se va a mostrar la textura.

```
void glTexParameterf(GLenum target, GLenum pname, GLfloat param);
```

- `target`: es el tipo básico de textura, `GL_TEXTURE_2D`.
- `pname`: corresponde al nombre del parámetro a ajustar.
- `param`: es el valor de este parámetro.

Los parámetros relacionados con la forma de envolver de la textura (*texture wrap*) son `GL_TEXTURE_WRAP_S` y `GL_TEXTURE_WRAP_T`. Sirven para determinar la manera en que se verá la textura:

- `GL_CLAMP`: recorta la textura una vez se superan las coordenadas 1.0 en (s,t). Dependiendo de los ajustes en los estados (se puede hacer independiente sobre s y t).
- `GL_REPEAT`: después de 1.0, bien sea en s y/o t de acuerdo al ajuste, la textura se repite.

Los parámetros relacionados con el filtrado de textura son `GL_TEXTURE_MIN_FILTER` y `GL_TEXTURE_MAG_FILTER`. Determinan el suavizado de las texturas cuando están a cierta distancia. Los filtros son MIN (*minification*) y MAG (*magnification*) para cuando el polígono texturizado está lejos y cerca respectivamente. Los filtros son:

- `GL_NEAREST`: utiliza el téxel más cercano mapeado, así ocupa más de un pixel en pantalla. Es el filtro más sencillo.

- `GL_LINEAR`: utiliza interpolación lineal para aproximar el color de un determinado píxel de acuerdo con el téxel correspondiente y sus vecinos. El resultado es un suavizado de la textura.
- `GL_NEAREST_MIPMAP_NEAREST`,
`GL_LINEAR_MIPMAP_NEAREST`,
`GL_NEAREST_MIPMAP_LINEAR`,
`GL_LINEAR_MIPMAP_LINEAR`: utilizan el filtro sobre el nivel inferior y superior del *mipmap* respectivamente y luego promedia el resultado para mostrarlo en pantalla.



Figura 1. Ejemplo de mipmap con 7 niveles.

Asociar la textura a la primitiva

Debido a que una textura es una imagen rectangular y una primitiva no lo es necesariamente, se debe especificar la forma en que la textura se acoplará a la primitiva. Esto se hace suministrando las coordenadas *s,t* (en 3D Studio Max se denominan *u,v*) para cada vértice mediante la función `glTexCoord{1234}{sifd}(T coords)`.

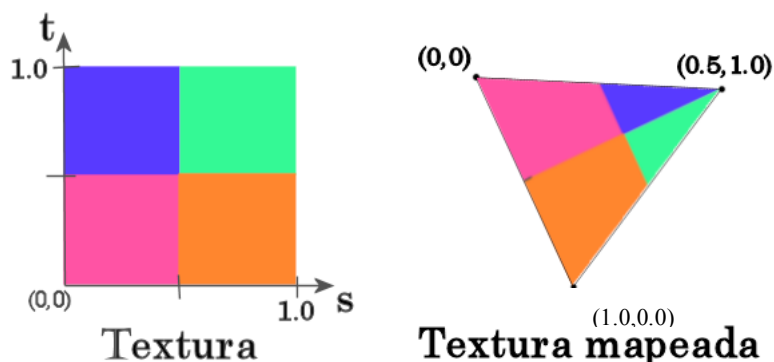


Figura 2. Ajuste de la textura a un triángulo.

Se debe colocar una función `glTexCoord` delante de cada `glVertex()`, teniendo en cuenta las coordenadas (*s,t*) de la textura y las de la primitiva.

Ejemplo:

```

GLuint texturaFondo; // Identificador de textura (variable global)

GLuint loadTexture(const char *file)
{
    GLuint textureID;

    // Cargar imagen
    loadImage(file, &w, &h, &data); // función que carga la imagen

    // Generar un identificador de textura
    glGenTextures(1, &textureID);

    // Configurar textura
    glBindTexture(GL_TEXTURE_2D, textureID);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                    GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                    GL_LINEAR);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, w, h, 0, GL_RGBA,
                 GL_UNSIGNED_BYTE, data);

    return textureID;
}

void init(void)
{
    ...
    texturaFondo = loadTexture("imagen.jpg");
}

void display(void)
{
    ...
    glBegin(GL_QUADS);
    // Asociar textura a objeto
    glTexCoord2f(0.0, 0.0);    glVertex3i(0, -5, -5);
    glTexCoord2f(1.0, 0.0);    glVertex3i(0, 5, -5);
    glTexCoord2f(1.0, 1.0);    glVertex3i(0, 5, 5);
    glTexCoord2f(0.0, 1.0);    glVertex3i(0, -5, 5);
    glEnd();
    ...
}

```

Alternativa para cargar la textura

La librería DevIL tiene una función que simplifica mucho el proceso, ya que no solo carga la imagen, sino también la asocia a una textura, ahorrándonos todos los pasos.

En la función `init()` se cargan las imágenes y se convierten a texturas con `ilutGLLoadImage()`.

```
ilInit();  
iluInit();  
ilutRenderer(ILUT_OPENGL);  
GLuint texture = ilutGLLoadImage("imagen.jpg");
```

Consideraciones a tener en cuenta

- La textura afecta a todos los objetos por igual.
- Si queremos que cada objeto tenga su propia textura, tendremos que llamar previamente a

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

cambiando el parámetro `texture`.

- También afecta a elementos como líneas. Si no queremos que se vean afectados, hay que poner:

```
glDisable(GL_TEXTURE_2D);
```

- La textura se mezcla con el color del objeto. Si el objeto es blanco, la textura se verá con sus propios colores.

Sonido y música con SDL_mixer

SDL es un conjunto de librerías escritas en C que suministran funciones para dibujo en 2D, audio y música y gestión de imágenes. La librería de audio es **SDL_mixer**. Se usa con gráficos 2D, aunque también se puede usar con 3D.

Independiza los canales de los sonidos del canal de música. De este modo se puede tener una música de fondo a la vez que suenan efectos sonoros (disparos, pisadas, golpes, etc.)

Su uso se recomienda sólo para juegos sencillos. La librería OpenAL es mucho más potente.

Instalación

Hay que instalar las librerías SDL y la SDL_mixer. En las opciones del *linker* hay que incluir:

```
-lSDLmain -lSDL -lSDL_mixer
```

En el código hay que incluir las librerías:

```
#include <SDL/SDL.h>  
#include <SDL/SDL_mixer.h>
```

Inicialización del subsistema de audio

En la inicialización de la librería SDL hay que poner el flag de este subsistema.

```
SDL_Init(SDL_INIT_AUDIO);
```

Luego hay que abrir el dispositivo de audio con `Mix_OpenAudio()`.

```
int Mix_OpenAudio(int frequency, Uint16 format, int channels,  
                  int chunksize);
```

donde:

- `frequency` es la frecuencia del sonido (22050).
- `format` es el formato de la muestra de sonido (AUDIO_U8, AUDIO_S8, AUDIO_U16, AUDIO_S16).
- `channels` es el número de canales (2 para estéreo, 1 para mono).
- `chunksize` es el número de bytes utilizados para la muestra de salida. Debe ser potencia de 2 (512, 1024, 2048, 4096...).

Carga de los archivos WAV

Se utiliza una llamada por cada archivo a reproducir.

```
Mix_Chunk *sonido;  
sonido = Mix_LoadWAV("sonido.wav");
```

Carga los formatos: WAV, AIFF, RIFF, OGG y VOC.

Reproducción del sonido

La reproducción de un sonido se hace con la función `Mix_PlayChannel()`.

```
int Mix_PlayChannel(int channel, Mix_Chunk *chunk, int loops);
```

donde:

- `channel` es el canal por el que sonará. Con el valor -1, se asigna automáticamente (recomendado).
- `chunk` es el sonido que se va a reproducir.
- `loops` indica cuántas veces se repite el sonido.

Ejemplo:

```
int canal;  
canal=Mix_PlayChannel(-1,sonido,0);
```

Si llamamos a `Mix_PlayChannel` varias veces seguidas, todas sonarán a la vez. Para que suenen una detrás de otra hay que esperar a que termine la anterior. Para ello, debemos consultar si el canal está reproduciendo sonidos hasta que deje de hacerlo.

```
while ( Mix_Playing(canal) ) ;  
canal = Mix_PlayChannel(-1,sonido,0);
```

Reproducción de música de fondo

Es similar a la reproducción de sonido. Para **cargar** se utiliza `Mix_LoadMUS()`.

```
Mix_Music *musica;  
musica = Mix_LoadMUS("musica.mp3");
```

Reproduce los formatos: MP3, WAV, MOD, MIDI, OGG y FLAC.

Para **reproducir** se utiliza la función `Mix_PlayMusic()`.

```
Mix_PlayMusic(musica, -1);
```

Con el 2º parámetro se indica cuántas veces se repite la música: 0, no se repite; -1 se repite indefinidamente.

Para **parar** la música se utiliza:

```
Mix_HaltMusic();
```

Finalización del subsistema de audio

Antes de acabar hay que liberar los *chunks* de sonido y la música utilizados:

```
Mix_FreeChunk(sonido);  
Mix_FreeMusic(musica);
```

Y si ya no necesitamos más la librería SDL, utilizamos:

```
SDL_Quit();
```

Práctica 4

En esta práctica se debe recrear un pequeño parque similar a la de la figura 3. El motor gráfico debe funcionar en tiempo real con un observador que pueda moverse libremente a través de él. Para ello, se deben utilizar las funciones de control del observador de la práctica 2.

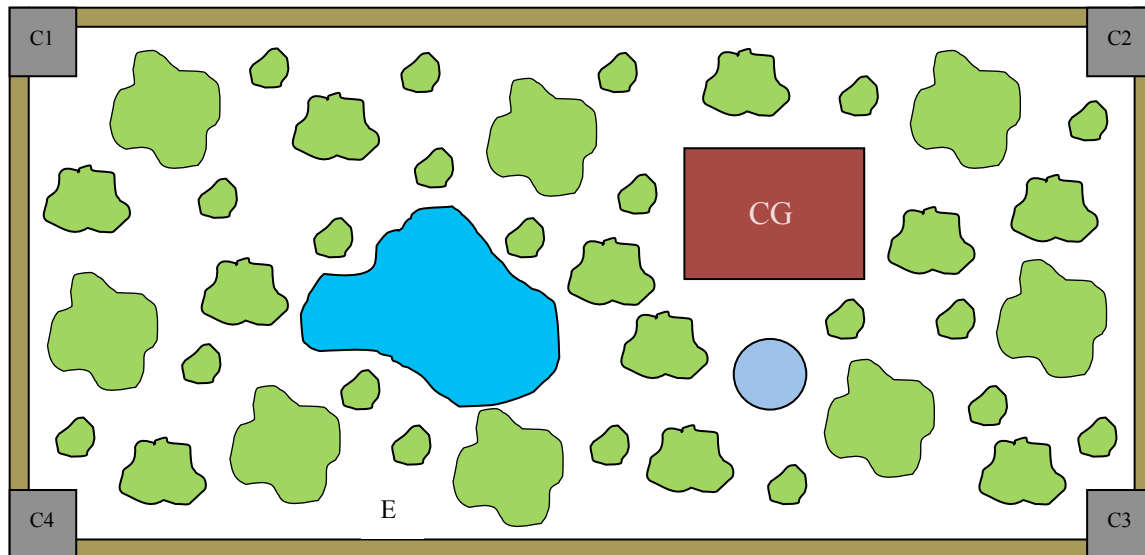








Figura 3. Ejemplo de parque.

Puede observarse en el plano del parque varias figuras que representan diferentes tipos de elementos:

-  Es una casa con tejado a dos aguas, similar a la de la figura 4a.
-  Es una fuente (figura 4b).
-  Son torres (figura 4c).
-  Representan una muralla exterior de menor altura que las torres (figura 4c).
-  Las manchas verdes representan árboles de distinta forma y tamaño, similares al de la figura 4d.
-  La mancha azul representa un lago.

Las letras tienen el siguiente significado: E es el punto de entrada donde debe empezar a moverse el observador; C1, C2, C3 y C4 son cámaras fijas situadas encima de cada torre mirando hacia el centro del lago y CG es una cámara giratoria situada encima de la casa, girando de manera continua.

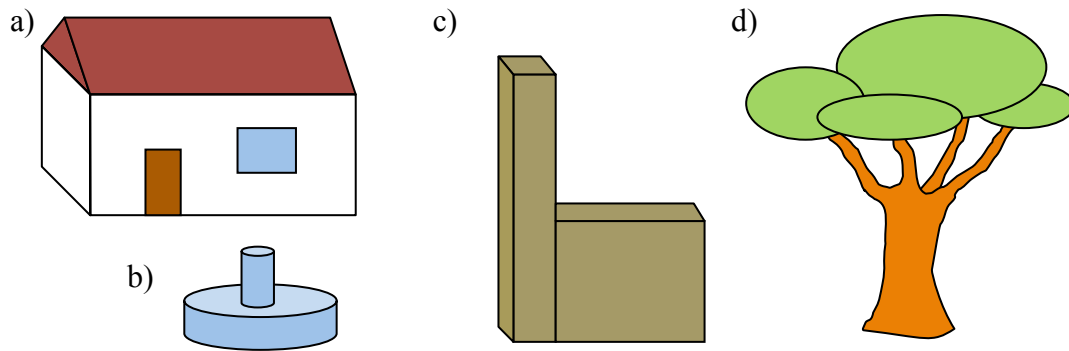


Figura 4. Elementos del parque: a) casa, b) fuente, c) torres y murallas y d) árboles.

Representación del parque

El parque, que no tiene por qué coincidir con la de la figura 3, debe tener un suelo y al menos 8 árboles grandes, 10 árboles medianos y 20 árboles pequeños, 1 casa, 1 lago, 1 fuente y 4 torres, rodeado todo el perímetro por 1 muralla. Estará **perfectamente iluminado**, para lo cual se necesitará calcular las normales a las superficies de los objetos que serán sólidos.

Los objetos se pueden representar de dos maneras diferentes:

- La primera consiste en que los objetos más complejos (casa, fuente, lago y árbol) se pueden modelar utilizando la herramienta 3D Studio MAX (3DSMAX) o similar e importando las coordenadas de sus vértices, superficies y normales a partir de un fichero ASE. Es decir, desde 3DSMAX se crea cada objeto individualmente y se exporta creando un archivo ASE por cada uno (arbol.ase, casa.ase, etc.). La fuente se importará desde los archivos `fuentes.ase` proporcionado junto con el enunciado. La aplicación `practica4` leerá cada archivo y situará el objeto correspondiente en la escena, escalándolo si es necesario.
La representación de los objetos de esta manera se valorará con **2 puntos**.
- La segunda manera consiste en utilizar las primitivas definidas en OpenGL para representar los objetos.
La representación de los objetos de esta manera se valorará con **1 punto**.

En ambos casos, se necesitarán unas **estructuras de datos adecuadas** para acceder a los elementos que forman la escena. También será necesario utilizar listas de visualización para replicar los árboles. Para los demás elementos no es necesario utilizar listas de visualización, pero sí recomendable.

Texturas de los objetos

Los objetos del parque opcionalmente pueden llevar incorporada una textura, simulando madera, ladrillos, piedra, etc.

Incluir esta opción en la práctica añadirá **1 punto** a la nota final.

Definición del observador

Para el observador se debe utilizar el mismo definido en la práctica 2, con las mismas pulsaciones de tecla para cambiar su posición, pero haciendo uso del teclado avanzado.

Movimientos del observador con el ratón

Opcionalmente, se podrán hacer movimientos con el ratón, en los que solamente se cambiará el punto al que mira el observador y nunca su posición.

Cuando se mueva el ratón **sin pulsar ningún botón**, será necesario determinar la cantidad de giro que hay que hacer, de modo que un movimiento a la izquierda o a la derecha variará el ángulo α del observador y un movimiento arriba o abajo, variará su ángulo β . Las variaciones dependerán de la sensibilidad que se le quiera dar al ratón. Una sensibilidad alta provocará un mayor ángulo de variación para un mismo desplazamiento.

Incluir esta opción en la práctica añadirá **1 punto** a la nota final.

Plano de situación

Mediante la pulsación de la tecla 'P', se deberá cambiar la vista 3D a una vista 2D con una cámara **ortográfica** que muestre un plano cenital del parque. Es decir, una vista como la mostrada en la figura 5 en la que aparezca la posición del observador con un punto **rojo**.

Para dibujar el punto puede utilizarse la primitiva `glBegin(GL_POINTS)` junto con la función `glPointSize(GLfloat size)` para especificar un tamaño en píxeles que muestre este punto aislado.

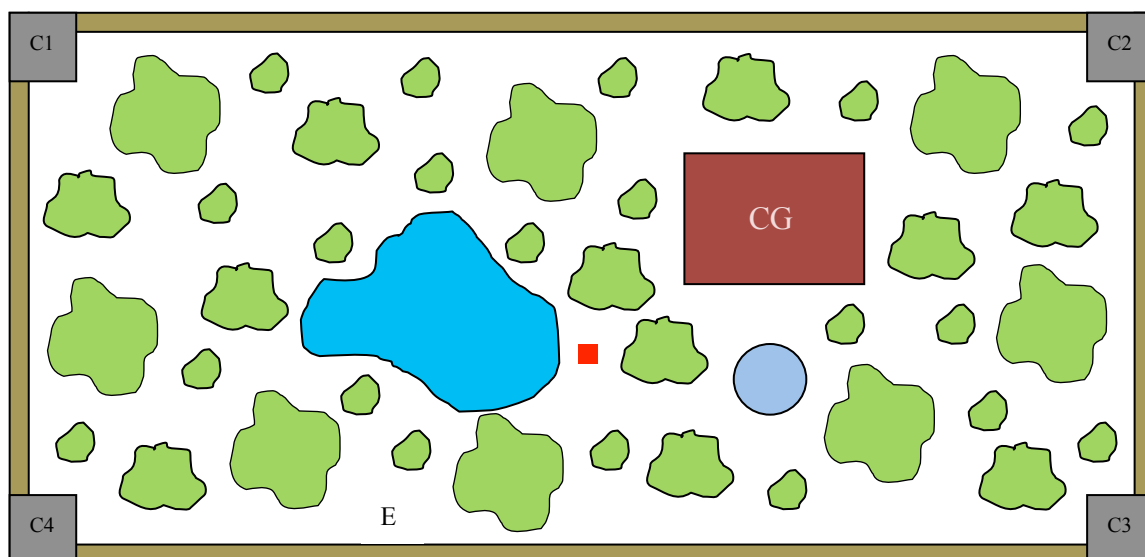


Figura 5. Vista cenital de la aldea.

Mientras esté la aplicación en esta vista 2D, se deben ignorar las pulsaciones de teclado y eventos de ratón.

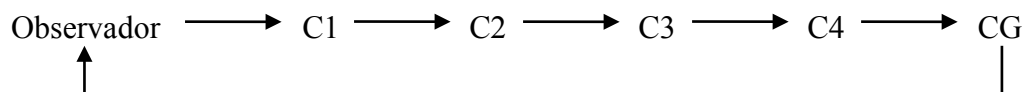
Otra pulsación de esta tecla 'P', devolverá la vista 3D con la visión y orientación que tenía antes de entrar en el plano 2D.

Incluir esta opción en la práctica añadirá **1 punto** a la nota final.

Cámaras

En las posiciones marcadas en la figura 3 como C1, C2, C3, C4 y CG deben situarse sendas cámaras de vigilancia, con cierta inclinación, de modo que se vea el interior del parque y los objetos que hay en él. Las cuatro primeras cámaras serán fijas, mientras que la última será giratoria observando los 360° de izquierda a derecha de manera continua. Se debe utilizar la función `idle()` para variar el ángulo de giro.

La pulsación de la tecla ‘C’ alternará entre cámaras y vista del observador según la siguiente secuencia:



Mientras esté la aplicación en modo cámara, se deben ignorar las pulsaciones de teclado y eventos de ratón.

Incluir esta opción en la práctica añadirá **1 punto** a la nota final.

Opcionalmente, se puede implementar un paso intermedio entre C4 y CG de modo que se visualicen las cuatro cámaras simultáneamente en la misma pantalla.

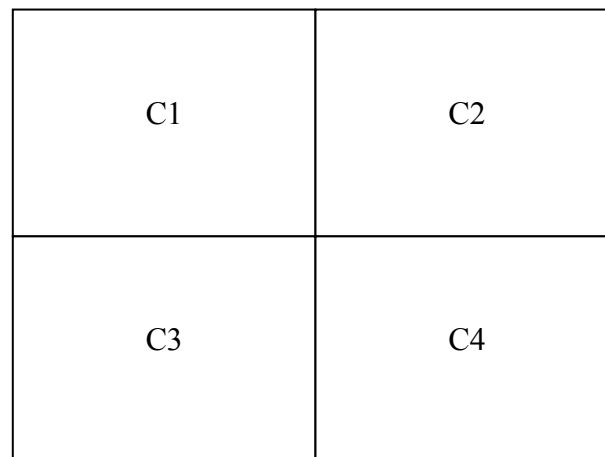


Figura 6. Vista multicámara.

Incluir la opción multicámara sumará **1 punto**.

Panel de información

Cuando se pulse la tecla ‘H’ debe aparecer en pantalla un menú mostrando las teclas y su función. Se representará como un pequeño panel con fondo oscuro con cierta transparencia y con las letras blancas (ver figura 7). Este panel se definirá como parte del HUD o visor frontal de datos.

Para crear la transparencia del fondo se deben utilizar las siguientes llamadas en la función `init()`.

```
glEnable(GL_BLEND);  
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

Con esto activamos la transparencia en los colores y provocamos que OpenGL tenga en cuenta el canal alfa de los mismos. Para que el fondo sea semitransparente hay que darle un color negro puro y una opacidad entre 0.6 y 0.8. Para ello hay que llamar a `glColor4f(0.0,0.0,0.0, 0.7)`.

Cuando el canal alfa vale 0, el color es totalmente *transparente*; con valor 1, es totalmente *opaco*.

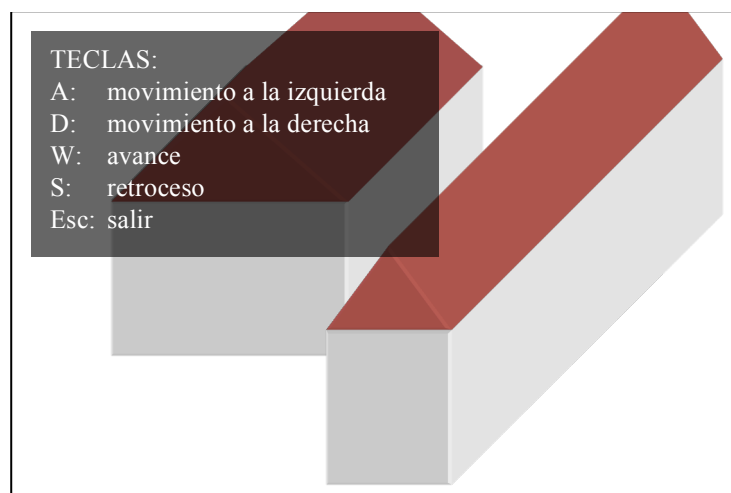


Figura 7. Apariencia del menú.

En el ejemplo de la figura 7 sólo se han puesto las teclas de los movimientos básicos. En la práctica hay que incluir todas las opciones de teclado y ratón implementadas.

Otra pulsación de la tecla 'H' debe quitar el menú.

Esta opción se valorará con **1 punto**.

Detección de colisiones

Opcionalmente, se podrá incluir un algoritmo de colisiones para evitar que se puedan atravesar las paredes y murallas, así como el resto de elementos de la escena. Se admite cualquier algoritmo y la cantidad de puntos que sume a la nota final dependerá de su complejidad.

Como recomendación se sugiere mantener un *array* de $M \times N$ celdas, dependiendo de las dimensiones del parque, en la que cada celda represente un rango de medidas. Por ejemplo, si el parque mide 100 x 50 metros (o cualquier otra unidad de longitud) y cada celda representa 1 m², entonces habrá un *array* de 100x50 celdas. De esta forma, los muros y demás objetos deberían tener dimensiones múltiplos de 1 m. Asociando la posición del observador a una

celda de este *array*, se puede saber si el siguiente movimiento del observador va a entrar o no en un obstáculo, bloqueando dicho movimiento si es incorrecto.

Implementar este algoritmo sumará **1 punto** sobre la nota final.

Grabación de la imagen en un fichero

Opcionalmente, se guardará la imagen del observador en el momento de pulsar la tecla ‘F’, en un fichero BMP. Obviamente, se deberán implementar las estructuras de los bloques de datos del BMP, crear el fichero y escribir en él estas estructuras; y no se permitirá hacer uso de las funciones del tipo `SaveToFile()` ya predefinidas en alguna librería como DevIL. No será necesario implementar el algoritmo que abra dicha imagen, utilizándose para su apertura la aplicación *Paint* de Windows, por ejemplo.

Implementar esta opción sumará **1 punto** sobre la nota final.

Música y sonidos

Opcionalmente, se puede reproducir un archivo como música de fondo. Pulsando la tecla ‘M’, se detendrá la música y pulsando de nuevo, se reanudará. Además cuando haya movimientos del observador con las teclas ‘WASD’, podrá sonar un sonido de pisadas.

Implementar esta opción sumará **1 punto** sobre la nota final.

Otras pulsaciones

También deben implementarse las siguientes acciones:

- Pulsando la tecla ‘R’ se colocará al observador en la posición inicial.
- Pulsando la tecla ‘Escape’, se terminará la aplicación.

IMPORTANTE: En las pulsaciones de letras hay que tener en cuenta que pueden ser **mayúsculas o minúsculas**.

Calificación final de la práctica

Todas las partes son opcionales. Si bien hay partes que dependen de la realización de alguna otra.

Resumiendo, éstas son las puntuaciones de cada parte:

- Definición de los objetos del parque, **1 o 2 puntos**.
- Incorporación de texturas, **1 punto**.
- Movimientos del ratón variando la vista del observador, **1 punto**.
- Vista en plano cenital, **1 punto**.
- Vista a través de cámaras de vigilancia, **1 punto**.

- Vista multicámara, **1 punto**.
- Panel de información, **1 punto**.
- Detección de colisiones, **1 punto** (con el algoritmo sugerido).
- Grabación de un fichero BMP, **1 punto**.
- Música y sonidos, **1 punto**.
- La claridad de la memoria a entregar y la modularidad del código se valorarán con hasta **2 puntos**.
- Cualquier otro añadido que quiera hacer el alumno deberá ser consultado al profesor y se valorará según su complejidad.

Entrega

Llamar a los archivos `practica4.cpp` y `practica4.cbp` y entregarlos comprimidos en un archivo ZIP o RAR junto con la **memoria** explicativa y cualquier otro fichero necesario (ASE, JPG, MP3...) para la correcta ejecución de la práctica.

Normas

1. La práctica principal es **obligatoria** y se podrá hacer en grupos de hasta **3 alumnos**.
2. Cada alumno o grupo entregará los trabajos a través del sistema Moodle mediante un único archivo comprimido (ZIP o RAR) que incluya **los ficheros fuente y todos los ficheros necesarios para su ejecución** directa desde cualquier directorio en el que se descomprima. Junto con los ficheros anteriores se entregará una **memoria** (en Word o PDF) de la práctica donde se detalle las aportaciones opcionales realizadas, la estructura de datos elegida y una explicación de las funciones y algoritmos principales incluidos.

Para los grupos de dos alumnos bastará con que entregue la práctica uno de ellos, indicando claramente en la memoria el nombre de los dos miembros del grupo.

3. Se concede libertad en la elección de herramienta para implementar. Preferentemente Code::Blocks, pero se admiten otros entornos como Eclipse, Visual C++, etc. para Windows, Linux, etc., siendo los alumnos quienes se encargarán de preparar una completa demostración de su desarrollo, en caso de ser requeridos para tal fin.
4. El **plazo de presentación** acaba el día **11 de enero de 2015 a las 23:55**. A partir del día de la entrega, se podrá solicitar a los grupos de prácticas que se considere oportuno una defensa de su desarrollo ante los profesores de la asignatura.
5. **La detección de copias o “colaboraciones” entre las prácticas presentadas, supondrá de inmediato el suspenso en la asignatura del presente curso tanto para el que copie como para el copiado.**
6. **Para resolver dudas** sobre la práctica, se deberá hacer uso de las **horas de tutoría** de la asignatura del profesor correspondiente o mediante el **sistema Moodle** en los foros habilitados para tal fin.
7. La presentación de la práctica supone la aceptación de las normas por parte de los alumnos.