

Videojuegos 2D

Jesús Alonso Alonso

PID_00183013



Universitat Oberta
de Catalunya

www.uoc.edu

Índice

Introducción	5
Objetivos	6
1. Estructura de un videojuego	7
1.1. Estructura	7
1.2. Sincronización	9
1.3. FPS	16
2. Tile based engine	18
2.1. Tipos	19
2.1.1. Perspectiva lateral	19
2.1.2. Perspectiva perpendicular	21
2.1.3. Perspectiva isométrica	22
2.2. Estructura de datos	23
2.3. Algoritmo de visualización	24
2.4. Selección e interacción	25
2.5. Navegación	26
2.5.1. Navegación directa	26
2.5.2. Navegación con <i>scroll</i> (incremental)	27
2.6. Variantes	28
2.6.1. <i>Isometric tile map</i>	29
2.6.2. <i>Hexagonal-based tile map</i>	34
2.6.3. <i>Tiles apilables</i>	35
3. Física	37
3.1. Conceptos	37
3.1.1. Cinemática	37
3.1.2. Fuerza	38
3.1.3. Gravedad	38
3.1.4. Resorte	39
3.2. Saltos	40
3.2.1. Aproximación	40
3.2.2. Trigonometría	43
3.2.3. <i>Gameplay</i>	45
3.3. Colisiones elásticas	47
3.3.1. Idea conceptual	48
3.3.2. Vectores normal y tangencial	50
3.3.3. Descomposición de velocidades	51
3.3.4. Componente tangencial final	52
3.3.5. Componente normal final	52

3.3.6.	Vectorización	52
3.4.	Proyectiles	53
3.5.	Detección de colisiones	56
3.5.1.	<i>Tile based</i>	56
3.5.2.	Por píxel	57
3.5.3.	<i>Bounding circles</i>	62
3.5.4.	<i>Bounding boxes</i>	63
3.5.5.	Jerarquías de objetos envolventes	65
4.	Motores de física	67
4.1.	Box2D	69
4.1.1.	Conceptos	70
4.1.2.	Arquitectura y diseño	71
4.1.3.	Funcionalidades	73
4.1.4.	Visualización y testeo	80
4.1.5.	Un caso práctico	82
4.2.	Torque 2D physics	87
5.	Programación gráfica 2D	90
5.1.	GLUT	90
5.2.	API Gráficas	95
5.2.1.	DirectX	95
5.2.2.	OpenGL	96
5.2.3.	SDL	97
5.2.4.	Allegro	98
5.3.	OpenGL	98
5.3.1.	Estructura	99
5.3.2.	Geometría	100
5.3.3.	La máquina de estados	103
5.3.4.	Transformaciones geométricas	105
5.3.5.	Matrices de proyección y de visión del modelo	106
5.3.6.	Definición de cámara	109
5.3.7.	Definición de vistas (<i>viewports</i>)	111
5.3.8.	Colorear el fondo	112
5.3.9.	Modos de visualización	112
5.3.10.	Superficies ocultas: <i>Z-buffer</i> y <i>Culling</i>	113
5.3.11.	Modelos de sombreado	114
5.3.12.	<i>Display lists</i>	115
5.3.13.	Creación de objetos jerárquicos	116
5.3.14.	Texturas	117
5.4.	DirectX	124
5.4.1.	Programación en Windows	124
5.4.2.	DirectX Graphics	127
5.4.3.	DirectInput	131

Introducción

En este módulo trataremos los aspectos involucrados en la realización del que será nuestro primer videojuego. Con este propósito se establecerán las bases de la programación de videojuegos ofreciendo su estructura principal, se describirá el motor *tile based engine*, daremos a conocer cómo realizar la gestión de eventos y ventanas y unos módulos iniciales de física y programación gráfica. De este modo seremos capaces de implementar un videojuego completo en su versión 2D.

La tecnología que utilizaremos a tal efecto se basará en la herramienta de desarrollo Visual Studio .net y las librerías GLUT para la gestión de ventanas y eventos, así como OpenGL como API gráfica. La elección de la primera herramienta viene dada por su amplia aceptación y utilización en el ámbito del videojuego de carácter profesional. Las librerías escogidas responden a una línea de programación basada en la filosofía multiplataforma y libres.

Para acabar, mostraremos la solución que nos brinda DirectX para la programación de videojuegos 2D.

Objetivos

En este módulo didáctico se presentan al alumnado los conocimientos necesarios para conseguir los siguientes objetivos:

1. Entender la estructura de un videojuego.
2. Distinguir y clasificar los diferentes tipos de videojuegos 2D basados en celdas.
3. Entender los diferentes módulos de física que intervienen en un videojuego 2D: cinemática, fuerza, gravedad, resorte, respuesta a colisiones y proyectiles.
4. Conocer las diferentes técnicas que se pueden utilizar para la detección de colisiones en este ámbito.
5. Implementar un videojuego 2D haciendo uso de un *tile based engine*.
6. Gestionar ventanas y eventos con la librería GLUT.
7. Conocer las API gráficas más importantes.
8. Conocer al detalle las funcionalidades básicas de OpenGL para implementar un videojuego 2D.
9. Tener una aproximación de la solución que proporciona DirectX mediante la interfaz ID3DXSprite.

1. Estructura de un videojuego

Cuando hablamos de videojuegos hacemos referencia a una forma completamente diferente de programar software. La diferencia principal entre una aplicación convencional y un videojuego radica en la gestión de eventos: un programa ofrece respuestas a eventos y, una vez son servidos, permanece a la espera de una nueva orden; en cambio, un videojuego es un programa que debe actuar en tiempo real, tiene que estar haciendo cálculos y dibujando en pantalla constantemente. No se puede esperar a que suceda un evento para poder actuar: aunque el jugador no haga nada, no presione una sola tecla, no mueva el ratón, el juego tiene que calcular el tiempo que lleva jugando en ese nivel, si le va a atacar algún enemigo, si está cargando energía y, por supuesto, dibujar en pantalla todo lo que va sucediendo.

1.1. Estructura

La mayoría de videojuegos están contruidos bajo una misma estructura básica sobre la cual corre el programa. Esta estructura puede variar, pero por lo general se presenta de la siguiente manera:

- **Inicialización.** Es la primera etapa y en ella el juego se sitúa en un estado inicial predefinido. Para ello deben efectuarse las inicializaciones por lo que respecta a las librerías o motores que vayan a utilizarse, de variables y estructuras de datos referentes a los atributos de las entidades o personajes, escenarios, configuraciones, etc., y de los diferentes recursos físicos que vayan a emplearse, tales como gráficos o sonidos.
- **Ciclo de juego.** Es la parte donde está toda la acción y es un punto que se irá repitiendo una y otra vez hasta que el jugador pierda, gane o se salga del juego. En general podríamos dividir esta sección en tres partes:
 - **Entrada.** Es donde se captura todo lo que hace el jugador, como presionar los botones del control, mover el ratón, presionar las flechas del teclado y el resto de información que recibe el juego.
 - **Proceso.** Es la parte donde se procesa toda la información que se recibió de entrada y tiene lugar la lógica del videojuego. Es donde se efectúan los módulos más importantes del videojuego junto al tratamiento gráfico. Se llevan a cabo los cálculos de la física del juego y la inteligencia artificial.
 - **Salida.** Se encarga de enviar al jugador toda la información que se procesó en el paso anterior, es decir, se le muestra la respuesta a lo que

hizo. Se visualiza en pantalla el estado actual, se hacen sonar músicas o efectos sonoros, etc.

- **Finalización.** Es el último punto y consiste en liberar los recursos y memoria utilizada durante el juego, dejando el sistema en un estado óptimo, esto es, tal y como lo habíamos encontrado. Previamente, también podremos realizar algunas tareas tales como el registro de la tabla de puntuaciones u otros valores.

Es importante diferenciar las tres etapas del ciclo del videojuego y ver qué tarea se delega en cada una de ellas: en la entrada tan sólo nos ocupamos de captar los posibles eventos; el proceso se encarga en primer lugar de procesar la entrada y más adelante de la lógica del videojuego, y, por último, la capa de salida se encarga de visualizar y reproducir la música.

Con los puntos anteriormente comentados ya podemos confeccionar lo que será la estructura principal de nuestro primer videojuego. Haremos uso de la notación inglesa, que es la utilizada por las librerías y motores en su inmensa mayoría.

```
void main()
{
    bool end = false;

    Initialization();

    do {

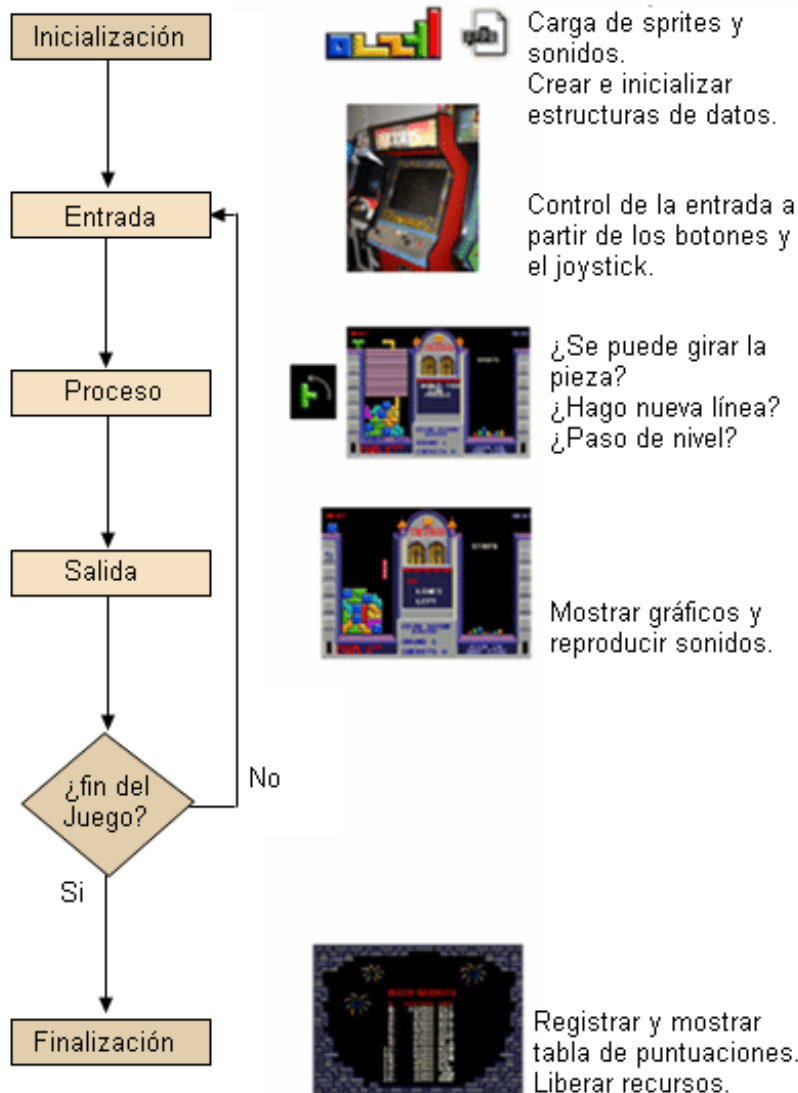
        Input();
        Process();
        Output();

    } while( !end );

    Finalization();
}
```

Tetris

Veamos un ejemplo de cada uno de los módulos de esta estructura haciendo uso de un juego conocido por todos: Tetris.



1.2. Sincronización

La estructura que hemos presentado hace referencia al esquema base de programación que suele tener un videojuego. Esta estructura, sin embargo, no tiene en cuenta un aspecto imprescindible: la potencia de la plataforma destino donde se ejecutará la aplicación. Esto es algo que hay que tener en cuenta sobre todo si la plataforma destino es un PC, dado que las diferencias de prestaciones técnicas entre ordenadores suelen ser muy diferentes (procesador, memoria, tarjeta gráfica, etc.).

Prestemos atención al siguiente ejemplo, en el que podremos ver cuál es el resultado de aplicar el algoritmo presentado sin más. Imaginemos que implementamos un videojuego muy simple o, mejor dicho, lo que podría ser el inicio de desarrollo de un videojuego en el cual tenemos un personaje caminante que va de izquierda a derecha de nuestra pantalla.

En dicho videojuego contaremos con una inicialización en la cual declaremos nuestras estructuras de datos necesarias (posición y velocidad del personaje) y les asignaremos un valor. La finalización del videojuego no contendrá código alguno, ya que no utilizaremos ningún recurso y tampoco haremos uso de memoria dinámica. El ciclo de juego contendrá la lógica de nuestro simple videojuego, esto es, avanzar la posición del personaje según la velocidad indicada. Por tanto, el código de esta implementación podría ser el siguiente:

```
//variables e inicializaciones

int x = 0; //posición del personaje

int vx = 2; //velocidad del personaje

//ciclo

do {

    x = x + vx; //actualizamos posición del personaje

    PintarPersonaje(x);

} while (1);

//finalización (nada, no es necesario)
```

Ahora supongamos que ejecutamos este programa en tres máquinas diferentes y obtenemos los resultados que figuran en la siguiente tabla:

Resultado de ejecutar nuestro videojuego en tres máquinas diferentes

	Procesador	Tiempo ciclo	Avance después de 1 segundo
PC1	250 MHz	0,040 ms	$25 \times 2 = 50$ px
PC2	500 MHz	0,020 ms	$50 \times 2 = 100$ px
PC3	2 GHz	0,005 ms	$200 \times 2 = 400$ px

En esta tabla podemos observar la ejecución de nuestra aplicación en tres ordenadores diferentes. Según el procesador de cada uno de ellos, el tiempo que tardará en ejecutarse el ciclo del juego será diferente y por tanto, el avance en píxeles de nuestro personaje también. Las columnas que hacen referencia al procesador y al tiempo de ciclo contienen valores simbólicos para ilustrar las diferencias que podrían llegar a producirse. Cada una de las pasadas de nuestro ciclo de juego se corresponde con lo que se suele denominar "un *tick* de juego". Más adelante utilizaremos este concepto.

Cuando implementamos un videojuego, es de esperar que este se pueda jugar con la misma velocidad, con independencia de la plataforma destino. En el caso que acabamos de mostrar, el movimiento del personaje depende de la potencia de la máquina y esto puede llegar a producir su completa no jugabilidad (porque va demasiado rápido y el control se hace difícil), o bien todo lo contrario (es decir, el movimiento es tan lento que se hace igualmente injugable).

Para solventar este problema, surge el concepto de **sincronización**. El objetivo no es otro que el de poder llegar a disfrutar de nuestro juego a una cierta velocidad con independencia del hardware donde corra. Este problema puede ser abordado de dos maneras diferentes: podemos optar por una sincronización por *framerate* (frecuencia de fotograma) o una sincronización por tiempo:

- **Sincronización por *framerate***. Consiste en fijar o acotar la velocidad mínima del *tick* de nuestro juego. Cada *tick* debe realizarse en un tiempo mínimo y hasta que este no haya transcurrido, no seguiremos con la ejecución del videojuego, esto es, con la siguiente pasada de nuestro ciclo de juego.
- **Sincronización por tiempo**. Consiste en ubicar (o actualizar, por lo general) los elementos del juego según su velocidad en aquella posición que toque según la diferencia de tiempos entre *ticks*.

Ambas filosofías son bastante diferentes entre sí. Observemos la implementación y el resultado de cada una de ellas siguiendo con el ejemplo anterior.

a) Sincronización por *framerate*

Para limitar el tiempo mínimo por *tick* de nuestro juego, será necesario obtener de alguna manera el tiempo inicial y final de cada pasada de nuestro bucle del juego. Para ello, un programador suele contar con funciones que nos pueden venir proporcionadas por la API del sistema operativo, o bien de la librería o motor que estemos utilizando. Estos métodos suelen devolvernos un valor que suele hacer referencia al tiempo en milisegundos transcurridos desde el inicio de la ejecución del programa o bien, el número de milisegundos transcurridos desde el año 1970. Con independencia de este hecho, nosotros haremos un mismo uso, ya que lo único que nos interesa es hacer una resta entre dos valores relativos y ver el tiempo que ha pasado. Citaremos dos ejemplos de métodos que nos podrían venir bien.

Para el caso de la librería multiplataforma GLUT, podemos hacer uso del método `glutGet` con el parámetro `GLUT_ELAPSED_TIME`. Con el sistema operativo Windows existe la función `GetTickCount`. Mostramos a continuación un ejemplo de uso de las dos funciones:

i) Implementación con GLUT

```
include <glut.h>

int t = glutGet(GLUT_ELAPSED_TIME);
```

ii) Implementación con Windows

```
include <windows.h>

int t = GetTickCount();
```

De manera genérica, para ilustrar el código resultante de implementar nuestro videojuego teniendo en cuenta la sincronización, haremos uso de una función llamada `ObtenerTiempo()` que nos proporcionará esta funcionalidad. La implementación de la sincronización por *framerate* de nuestro videojuego del personaje caminante es la siguiente:

```
//variables e inicializaciones

int x = 0; //posición del personaje

int vx = 2; //velocidad del personaje

int t1, t2;

//ciclo

do {

    t1 = ObtenerTiempo(); //obtenemos tiempo inicial

    x = x + vx; //actualizamos posición del personaje

    PintarPersonaje(x);

    //acotamos velocidad mínima del tick

    do { t2 = ObtenerTiempo();

        } while (t2-t1 < 20);

} while (1);

//finalización (nada, no es necesario)
```

En negrita podemos apreciar las líneas de código que han sido añadidas a la implementación inicial. Hacemos uso de dos variables nuevas (`t1` y `t2`), a partir de las cuales medimos el tiempo transcurrido en un *tick* del juego. Hasta que

este no sea superior a un cierto valor no dejamos proceder con el siguiente *tick*. El valor escogido es 20, ya que queremos que nuestro juego funcione a unos 50 fps ($1.000 \text{ ms}/20 \text{ ms} = 50$), es decir, tal y como lo hacía el PC2.

Observación

Para escoger a qué *framerate* queremos que funcione nuestro juego debemos tener en cuenta dos cosas. En primer lugar, nuestra limitación humana, capaz de diferenciar unas 25 imágenes por segundo, y en segundo lugar, la carga de recursos o gestión de memoria dinámica que puede hacer variar mucho el rendimiento en diferentes momentos del juego. Dado que un valor por debajo de los 25 fps hace que el jugador pierda la fluidez visual, debemos tener en cuenta que nuestro juego tendrá momentos donde sea necesario hacer más o menos cálculos y gestiones de carga y descarga de recursos, utilizar un valor superior es lo aconsejable (entre 30 y 60 fps, dependiendo del tipo de juego).

Este método de sincronización, por tanto, hará que en nuestro ejemplo el juego ejecutado desde el PC3 funcione de idéntica manera a como lo hace el PC2. Sin embargo, para el caso del PC1, el juego irá igual de lento, cuando llegue a la comprobación de tiempos nunca deberá esperar puesto que habrá pasado ese tiempo. Para este caso, la única posibilidad sería optimizar el juego. Esta sería una situación que debería abordar el *publisher*, debe hacer constar estos datos en forma de requerimientos mínimos o aconsejables para el buen funcionamiento y disfrute del juego.

b) Sincronización por tiempo

Este tipo de sincronización hace uso de una estrategia totalmente diferente. En lugar de acotar la velocidad de nuestros *ticks*, actualizamos las entidades con un valor proporcional al tiempo transcurrido. De esta manera si ha pasado más o menos tiempo, por ejemplo en el caso de desplazarnos, la distancia será mayor o menor.

Veamos su implementación utilizando el mismo ejemplo de nuestro personaje caminante.

```
//variables e inicializaciones

int x = 0; //posición del personaje

float vx = 0.100; //velocidad del personaje

int t1, t2;

t1 = ObtenerTiempo(); //inicializamos tiempo

//ciclo

do {

    t2 = ObtenerTiempo(); //obtenemos tiempo frame actual

    x = x + vx*(t2-t1); //actualizamos según tiempo transcurrido

    t1 = ObtenerTiempo(); //obtenemos tiempo anterior frame

    PintarPersonaje(x);

} while (1);

//finalización (nada, no es necesario)
```

Al igual que antes, en **negrita** podemos apreciar los puntos clave de la implementación de este tipo de sincronización. Volvemos a necesitar dos valores temporales. En este caso uno hace referencia al momento justo antes de hacer la actualización del estado de las entidades de nuestro juego y el otro inmediatamente después. También podemos apreciar una diferencia notoria en la velocidad del personaje. ¿Por qué 0.100? Si nos fijamos en la tabla de resultados de ejecución entre las tres máquinas y recordamos nuestra prioridad de ejecutarse tal y como nos brinda la potencia del PC2, podemos ver cómo el desplazamiento que tiene lugar al cabo de un segundo son 100 píxeles.

Por tanto, dado que nuestra función que nos devuelve el tiempo lo hace en milisegundos, veremos que cuando la diferencia entre los dos tiempos sea de 1.000 ms (1s), tendremos que $x = x + 0.100 * 1.000 = x + 100$, que es justo el desplazamiento que queremos. Si por ejemplo el tiempo transcurrido es de medio segundo, tendremos $x = x + 0.100 * 500 = x + 50$. Perfecto, lo que queríamos: la mitad de la distancia anterior en la mitad de tiempo.

Con este sistema, por tanto, podemos ver que el resultado esperado de la ejecución será el mismo con independencia de la máquina. Eso sí, dependiendo de la potencia de nuestra máquina, la visualización será más o menos fluida.

En el caso del PC1 anterior, el resultado será el esperado (el juego no irá más lento) pero sin embargo irá un poco "a saltos" con la consecuente pérdida de jugabilidad.

En esta tabla podemos resumir las ventajas y desventajas de utilizar estos dos métodos.

	Sincronización por <i>framerate</i>	Sincronización por tiempo
¿Soluciona el problema?	Sólo si la máquina es más potente.	Soluciona el caso con independencia de la máquina.
Resultado visual	El resultado visual es el mismo sea cual sea la máquina.	El resultado visual es más o menos fluido dependiendo de la máquina.

Llegados a este punto, se nos plantea una duda: ¿qué técnica de sincronización debemos utilizar? La respuesta, como en la mayoría de casos en los que nos planteamos una solución tecnológica, es un depende. Por lo general, lo que se suele hacer es una sincronización por *framerate*, que es la más sencilla, sobre todo para juegos 2D y, la sincronización por tiempo, es una técnica imprescindible para los juegos en red. En los juegos multijugador en red, es el valor de la latencia (el tiempo de espera de la comunicación entre cliente y servidor) el que marcará la actualización de nuestras entidades del juego.

Sirva como ejemplo didáctico plantearse por qué los jugadores que hacen uso de una mejor conexión están en clara ventaja respecto a los que lo hacen en circunstancias peores. El motivo de la frustración a la que pueden llegar los jugadores con peor conexión no es otro que el número de interacciones y representaciones del estado del juego por segundo que pueden tener: menor, cuanto mayor latencia tengan.



En los juegos en red, como el archiconocido "Quake III Arena" de ID Software, es necesario implementar el sincronismo por tiempo.

1.3. FPS

Visto el problema de la sincronización, éste puede ser un buen momento para conocer o medir el rendimiento de nuestro juego. El valor que suele tomarse como referencia es el número de muestros del estado del juego que se pueden hacer por segundo: los fotogramas por segundo¹.

⁽¹⁾En inglés, *frames per second* (FPS).

En un juego en dos dimensiones, es un valor que carece de importancia, ya que no es necesario realizar la mayoría de etapas del complejo flujo de visualización². Los efectos de sombras, los cálculos de iluminación, así como los de visualización o el filtrado de texturas, no tienen comparación en cuanto a complejidad y tiempo de procesado respecto a los juegos en tres dimensiones. Sin embargo, en los juegos 3D, es un valor que debe tenerse muy en cuenta para disfrutar al máximo del proceso de inmersión que debe ofrecernos.

⁽²⁾En inglés, *pipeline*.

Esta métrica es muy sencilla de implementar. Para ello, es necesario captar el tiempo al principio del *tick* de nuestro juego y al final de éste. La diferencia entre estos dos valores nos dará el tiempo necesario para calcular y mostrar un estado de nuestro juego. Veamos una posible implementación:

```
t1 = glutGet (GLUT_ELAPSED_TIME);

do {

    Input ();

    Logic ();
```



```
Render();

t2 = glutGet(GLUT_ELAPSED_TIME);

fps = 1000.0/(t2-t1);

t1 = t2;

} while(1);
```

Otra estrategia para calcular los FPS puede ser la siguiente. Primero obtenemos el tiempo inicial; a cada pasada (*tick*) de nuestro juego incrementamos un contador y, si ha transcurrido un segundo, actualizamos el valor que nos indica la frecuencia. Veamos aquí una posible implementación de esta segunda técnica:

```
timebase = glutGet(GLUT_ELAPSED_TIME);

frame = 0;

do {

    Input();

    Logic();

    Render();

    frame++;

    time = glutGet(GLUT_ELAPSED_TIME);

    if (time - timebase > 1000) {

        fps = frame*1000.0/(time-timebase);

        timebase = time;

        frame = 0;

    }

} while(1);
```

2. *Tile based engine*

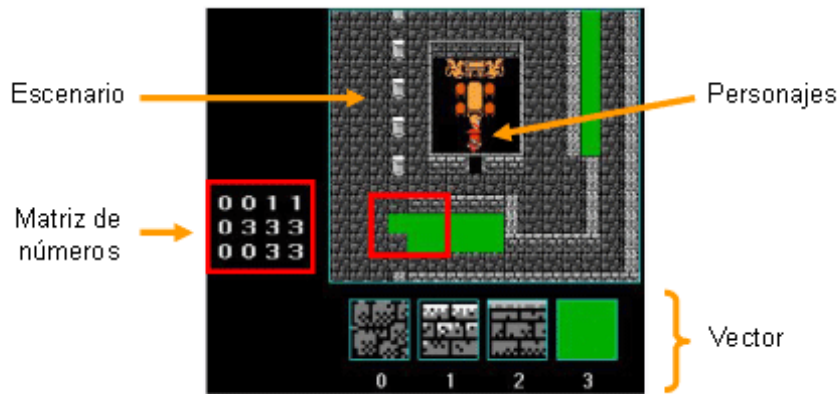
Para explicar la tecnología utilizada en los videojuegos de dos dimensiones, nos basaremos en los juegos basados en mapas, ya que el resto pueden ser considerados un subconjunto de éstos de menor complejidad. Haciendo uso de la terminología inglesa podemos referirnos a ellos como *tile based games* (juegos basados en celdas), que utilizan el motor precisamente llamado *tile based engine*.

Éste es un campo amplísimo que abarca quizás la mayoría de juegos que se han producido a lo largo de la historia, desde los juegos de plataformas al Tetris o el Diablo, por ejemplo. Existe bajo infinitas variantes, pero su esquema general se podría resumir en los elementos siguientes:

- Un escenario definido mediante una matriz de números. Cada número se usa para indexar un vector, que contiene los diferentes gráficos del mapa/fondo.
- Un personaje (o personajes) que puede moverse por ese mapa.
- Enemigos (opcional).
- Un algoritmo de detección de colisiones para averiguar por dónde puede o no puede pasar el personaje, así como la obtención de ítems o el alcance de proyectiles.

La idea es realmente simple. Partimos de un mapa que contiene una serie de números que conformarán los diferentes tipos de baldosas o celdas que tendremos, y cada uno de estos valores se identificará con un *bitmap*. Es decir, la representación física de estos números tendrá un significado semántico a partir de imágenes. El conjunto de estos números, los identificadores de *tiles* (azulejos), dispuestos estratégicamente conformarán la configuración y el aspecto de nuestro mapa.

Mostremos los diferentes aspectos que acabamos de citar haciendo uso de una imagen perteneciente al juego Final Fantasy:



Primera versión del juego Final Fantasy del año 1987

Este hecho nos permite corroborar por qué en la mayoría de los juegos que estamos acostumbrados a ver tiene lugar una repetición masiva de pequeñas imágenes. La configuración de mapas mediante *tiles* permite hacer un uso eficiente de la memoria en cuanto a espacio se refiere. Tan sólo es necesario guardar estas matrices de *tiles* y, por otra parte, contar con una serie de pequeñas imágenes.

2.1. Tipos

Como hemos comentado, existen infinidad de variantes a este sistema. A continuación, daremos a conocer en detalle las más extendidas:

- la perspectiva lateral con la técnica *parallax*
- la perspectiva perpendicular
- la perspectiva isométrica llamada popularmente *filmation*

2.1.1. Perspectiva lateral

Es la variante utilizada principalmente en los juegos de plataformas. En ella el escenario está compuesto por un fondo y aquellos elementos con los cuales podemos interactuar. El fondo suele estar compuesto por una o varias capas, en las cuales colocamos diferentes elementos gráficos a modo decorativo y se mueven en dirección contraria a la del jugador.

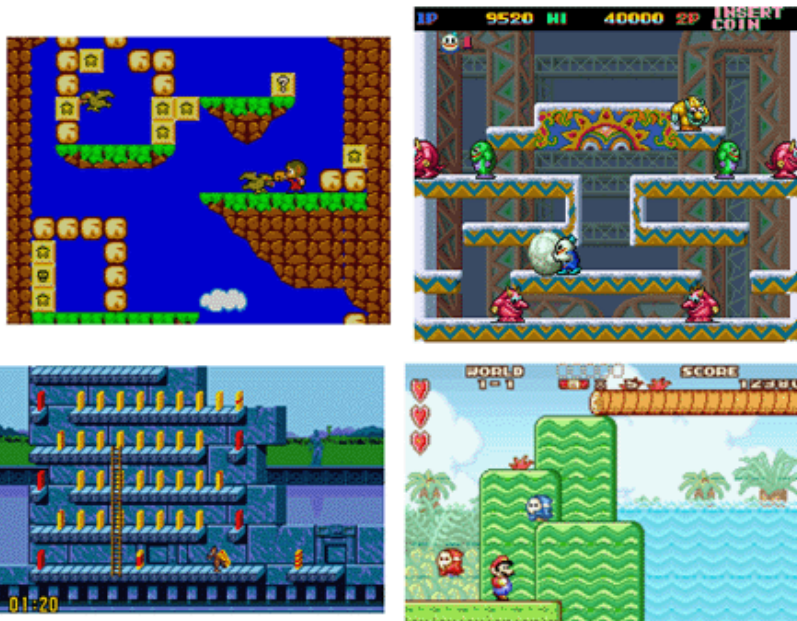
La posibilidad de trabajar con diferentes capas nos permite simular un cierto aspecto de profundidad dándole una cierta riqueza visual. Ésta es la técnica conocida como *parallax*, que consiste en mover cada una de estas capas a diferentes velocidades para obtener esta sensación de profundidad. Por ejemplo, podemos disponer de una capa compuesta por terrenos y montañas que transcurre a una cierta velocidad y otra más lenta formada por nubes.

Mediante la técnica **Parallax** podemos enriquecer el aspecto visual ofreciendo un cierto ambiente 3D gracias a las diferentes velocidades de cada una de las capas de fondo con las que trabajamos.

La lógica del juego se basa en el control de saltos y caídas desde diferentes plataformas, un control de movimiento de enemigos basado en su gran mayoría en patrones predefinidos y un sencillo algoritmo de detección de colisiones para impactos u obtención de ítems.

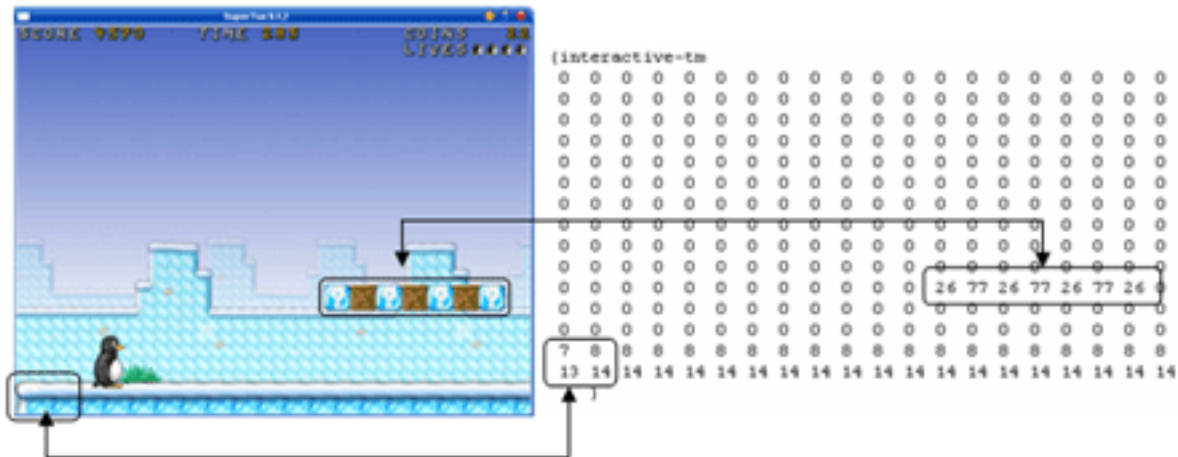
Ejemplos de juegos con perspectiva lateral

Veamos algunos juegos a modo de muestra e intentemos observar y diferenciar las diferentes *tiles* que aparecen en ellos:



De izquierda a derecha y de arriba abajo: "Alex Kid in the Miracle World", "Snow Bros", "Pushover" y "Mario Bros"

Veamos a continuación un ejemplo mediante el juego SuperTux, desarrollado mediante la API SDL y disponible en la dirección web: <http://supertux.lethargik.org/>.



La imagen de la izquierda corresponde al estado inicial de la primera pantalla y la imagen de la derecha a su mapa de *tiles* que está localizado en el fichero "`data\levels\world1\level1.stl`".

Actividad

En la imagen del juego SuperTux podemos apreciar la asociación que existe entre los números identificadores de celdas y su correspondiente *bitmap*. Se propone el siguiente ejercicio:

- 1) Modificar la configuración del mapa para que en todos aquellos espacios donde no tenga lugar ninguna *tile* con la cual interaccionar aparezcan monedas, y de esta manera podamos obtener una mayor puntuación.
- 2) Crear un plataforma superior, a partir de la cual poder pasar todo el nivel sin necesidad de esquivar los diferentes obstáculos.

Una vez conocida la técnica de los juegos basados en *tiles*, estamos en disposición de poder cambiar el aspecto de un videojuego si somos lo suficientemente hábiles, encontrando dónde y cómo se almacena la información de los escenarios.

2.1.2. Perspectiva perpendicular

Algunos juegos que utilizan esta vista son los del género de estrategia en tiempo real, los juegos de rol o algunos *shoot'em ups*. El manejo del escenario puede llegar a ser muy parecido al del caso anterior y, así como los de perspectiva lateral trataban con plataformas donde poder saltar, ahora dispondremos de celdas que serán transitables y otras que no. Otro elemento diferenciador es que en muchos casos se cuenta con un radar mediante el cual la navegación puede ser directa en lugar de incremental.

Ejemplos de juegos con perspectiva perpendicular

A continuación mostramos algunos ejemplos en los cuales, de nuevo, podemos corroborar la composición del mapa a partir de *tiles*:



De izquierda a derecha y de arriba abajo: "Cobra Mission", "Warcraft", "Dune 2" y "Zelda"

2.1.3. Perspectiva isométrica

Para el caso de los juegos con vista isométrica, también llamada *filmation*, el motor basado en celdas difiere en mayor medida. El motivo no es otro que el aspecto romboidal que en este caso presentan las celdas que mantienen una proporción 2:1. La idea de fondo, sin embargo, es exactamente la misma.

A lo largo de la historia este tipo de juegos se ha ido perfeccionando y podemos diferenciarlo en tres categorías:

- El *filmation* clásico: el aspecto del escenario era propiamente romboidal y la interfaz gráfica se utilizaba para rellenar el espacio libre.
- El *filmation* cuadrado: mostraba el escenario utilizando toda la pantalla.
- El *filmation* con *scroll*: la navegación se hace de manera que la cámara sigue constantemente al personaje.

Ejemplos de juegos con perspectiva isométrica

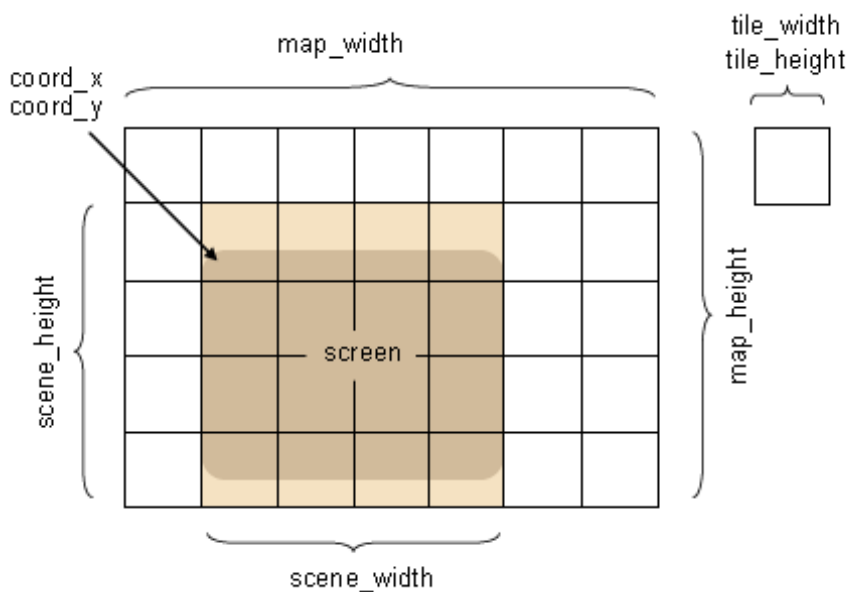
Una vez más, mostraremos unos ejemplos en los cuales se puede apreciar también el escenario compuesto de *tiles*, pero en este caso son romboidales.



De izquierda a derecha y de arriba abajo: "La abadía del crimen", "Knight Lore" y "Diablo"

2.2. Estructura de datos

La estructura de datos que permita almacenar este tipo de escenarios basados en celdas debe responder al contexto siguiente:



Usando una notación C, la estructura quedaría como sigue:

```
struct TileMap
{
    int tile_width, //Ancho y alto de las tiles
    tile_height; // (píxeles)
```

```
int map_height, //Ancho y alto de todo el mapa
    map_width;  //(tiles)

int scene_width, //Ancho y alto de la parte visible
    scene_height; //por pantalla del mapa (tiles)

int coord_x, //Coordenadas de la parte visible
    coord_y; //(píxeles)

Image *images; //Vector de imágenes para cada tile

int **data; //El mapa, la matriz de tiles
};
```

2.3. Algoritmo de visualización

El proceso de dibujado de nuestro mapa de *tiles* consiste en los siguientes pasos:

- 1) En primer lugar, para recorrer cada una de las celdas que conforman la parte visible del mapa, será necesario calcular los índices de la primera y última celda para cada uno de los dos ejes de la matriz.
- 2) A continuación, calculamos la coordenada de pantalla desde la cual empezaremos a pintar, teniendo en cuenta el desplazamiento de la coordenada de la escena visible. Si ésta no es múltiplo del tamaño de las celdas, será necesario pintar una celda más.
- 3) Por último, recorreremos la matriz resultante de la parte visible de nuestro mapa, y para cada paso, obtenemos el tipo de *tile*, la dibujamos en su posición e incrementamos este último valor para que vayan apareciendo consecutivamente en pantalla.

Por tanto, el código resultante sería el siguiente:

```
void RenderTileMap( TileMap *map )
{
    int tile;

    int Xo = map->coord_x / map->tile_width;
    int Yo = map->coord_y / map->tile_height;

    int Xf = Xo + map->scene_width - 1;
    int Yf = Yo + map->scene_height - 1;

    int pos_x = 0;
    int pos_y = 0;
```



```
for(int i=Y0; i<=Yf; i++)
{
    for(int j=X0; j<=Xf; j++)
    {
        tile = map->data[i][j];
        RenderImage(map->images[tile], pos_x, pos_y);
        pos_x += map->tile_width;
    }
    pos_x = 0;
    pos_y += map->tile_height;
}
}
```

2.4. Selección e interacción

En un motor basado en celdas, la selección de objetos o personajes y la interacción con ellos se realiza mediante un sencillo cálculo. Ambas operaciones deben implementarse teniendo en cuenta el siguiente aspecto.

En un instante determinado del juego nos encontraremos en una parte del mundo en el cual se desarrolla la historia. Ésta es la parte visible que nosotros mostramos en pantalla y que se encontrará en un sistema de coordenadas que irá desde la posición (0,0) hasta la resolución escogida (Rx-1,Ry-1). La selección e interacción tendrán que tener en cuenta que, cuando seleccionamos un objeto, lo hacemos a partir de este sistema de coordenadas y, por tanto, será necesario aplicar la transformación de coordenadas pertinente antes de hacer cualquier comprobación.

Con esta finalidad, siempre que queramos comprobar la posición de algún objeto o personaje, una vez hemos dado la orden mediante el ratón, por ejemplo, realizaremos los siguientes pasos:

- 1) Obtener la posición del ratón (mx, my) en pantalla. Por ejemplo, una vez se haya presionado, cuando interese, según cómo queremos que funcione nuestra interfaz.
- 2) Transformar el punto (mx, my) a coordenadas de mundo (del juego). Para ello le restamos el valor (coord_x, coord_y) que nos indica cuánto se encuentra desplazado nuestro mundo y cuál es el primer píxel que se está pintando en pantalla.
- 3) Realizar la comprobación oportuna. Por ejemplo, ver si tenemos algún personaje en esa posición, con el propósito de seleccionarlo para moverlo o darle una acción de atacar más adelante.

Para la selección múltiple de objetos o personajes, procederemos de igual manera. Transformamos punto inicial y final que conforman el recuadro de selección a coordenadas de mundo y comprobamos qué entidades se encuentran dentro de éste.

2.5. Navegación

La navegación sobre el escenario puede ser de dos tipos: directa o con *scroll*. Esto es, un acceso directo a una zona del mapa o paso a paso a través de él.

2.5.1. Navegación directa

Hacemos uso de esta navegación, por ejemplo, cuando seleccionamos un punto del área del radar para los juegos de estrategia en tiempo real o en algunos simuladores provistos de áreas de visualización en miniatura.

El problema que hay que resolver es el siguiente: actualizar la parte del mapa que va a ser visible por pantalla una vez el usuario haya pulsado el botón sobre el área del radar. A la variable, el punto de control de la parte visible de nuestro *tile based map* la hemos llamado (coord_x, coord_y). Por tanto, el objetivo será actualizar esta coordenada.

En este contexto es necesario conocer las siguientes variables:

- Posición del radar en la interfaz grafica (radar_x, radar_y), en coordenadas de pantalla.
- Proporción existente entre el tamaño del radar y el mapa del mundo. Es decir, cuántos píxeles en el área del radar conforman una celda.

Ejemplo de navegación directa

Nuestro mundo es de 32x32 celdas. El área del radar en pantalla mide 128x128 píxeles. Por lo tanto, 4x4 píxeles en el radar se corresponderán a una misma *tile*.

La fórmula para actualizar el punto (coord_x, coord_y) teniendo en cuenta los valores dados se realizará de la siguiente manera:

```
//Restamos offset de la IGU (valores en píxeles)
int x = mouse_x - radar_x;
int y = mouse_y - radar_y;

//Dividimos según la proporción existente (valores en tiles)
x = x >> 2;
y = y >> 2;

//Actualizamos variable de control de la parte visible
//de nuestro mundo
coord_x = x;
coord_y = y;
```

2.5.2. Navegación con *scroll* (incremental)

La navegación con *scroll* o incremental tiene lugar en la mayoría de juegos de plataformas al avanzar el personaje o en cualquier tipo de juego que permita navegar por el escenario posicionando el cursor sobre una de las zonas limítrofes de la pantalla. Detallaremos estos dos casos por separado: *scroll* provocado por el avance de un personaje y *scroll* provocado por la navegación vía ratón.

Scroll de personaje

Las opciones que se nos presentan son diversas. Pero en todas ellas el objetivo final es incrementar o decrementar el punto (coord_x, coord_y) según una constante que puede indicar la longitud del paso o avance del personaje o, por ejemplo, de forma variable según su velocidad.

Esta actualización puede realizarse de manera automática una vez pulsamos una tecla de dirección o cuando el personaje supera ciertos límites en pantalla, en una, dos, tres o hasta cuatro direcciones diferentes. Fijémonos cómo esta decisión definirá en gran parte la jugabilidad que va a ofrecer nuestro videojuego.

En este caso el código de visualización sufre una pequeña modificación, pues el *scroll* tiene lugar entre píxeles y no entre celdas o *tiles*. Los cambios a realizar radican en la inicialización del punto a partir del cual hay que pintar por pantalla.

```
int offset_x = map->coord_x % map->tile_width;
int offset_y = map->coord_y % map->tile_height;

int pos_x = -offset_x;
int pos_y = -offset_y;

if(offset_x!=0) Xf++;
if(offset_y!=0) Yf++;
```

Si nos fijamos en el código que acabamos de presentar, existe un pequeño detalle en lo que se refiere a la variable *offset*, que debemos tener en cuenta. Cuando el desplazamiento no es múltiplo del tamaño de la celda, la primera y última posición donde pintamos, por sus características, pueden salir de la pantalla. La mayoría de API gráficas realizan la operación de *clipping* por defecto, por lo que no es necesario añadir complejidad al algoritmo y el recorte de la imagen resulta automático. Asimismo, debemos destacar que cuando nos encontramos en este caso, es necesario pintar una celda más y, por tanto, debemos incrementar las variables Xf y/o Yf, según sea el caso.

Scroll vía ratón

El *scroll* vía ratón es más sencillo que el de personaje. En este caso queremos brindar al usuario la posibilidad de navegar por el mapa del juego cuando éste coloca el ratón sobre una de las zonas limítrofes de la pantalla y pulsa (o no) el botón.

Para ello, primero será necesario que definamos estas áreas. Si queremos permitir navegabilidad total, deberemos definir ocho zonas y obtener así todas las direcciones posibles. El tamaño de estas zonas rectangulares no debe ser muy grande, es suficiente con 4, 2 o incluso un único píxel, pues estamos forzando al usuario a desplazar el ratón hasta el límite de la pantalla.

Una vez detectada la dirección deseada, procederemos a actualizar la variable del punto de control de la zona visible. Para ello deberemos tener en cuenta, además, el control de los límites. El código resultante podría ser como el que se muestra a continuación:

```
void MoveScene(int dir)
{
    //Up
    if((dir==N) || (dir==NW) || (dir==NE))
    {
        if(coord_y > 0) coord_y--;
    }
    //South
    else if((dir==S) || (dir==SW) || (dir==SE))
    {
        if(coord_y < map_height-scene_height) coord_y++;
    }
    //West
    if((dir==W) || (dir==NW) || (dir==SW))
    {
        if(coord_x > 0) coord_x--;
    }
    //East
    else if((dir==E) || (dir==NE) || (dir==SE))
    {
        if(coord_x < map_width-scene_width) coord_x++;
    }
}
```

2.6. Variantes

La mayoría de juegos que trabajan bajo un *tile based engine* lo hacen mediante *tiles* cuadradas o rectangulares. Sin embargo, surgieron diferentes variantes para dotar de mayor realismo este tipo de juegos 2D. La confección de *ti-*

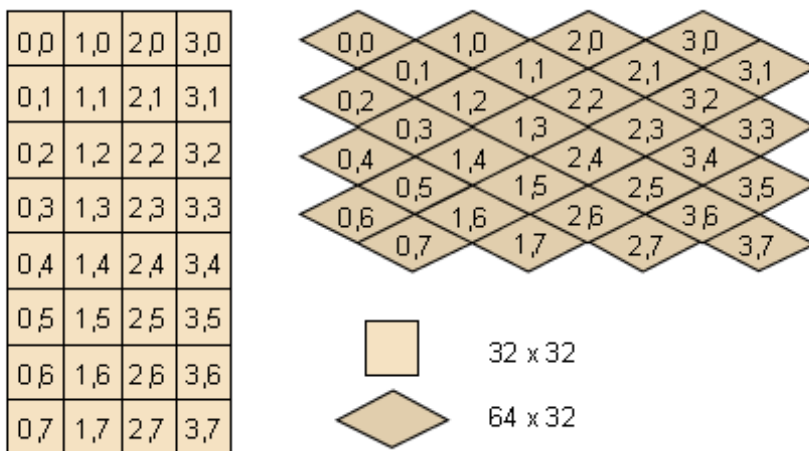
les rectangulares que simularan profundidad mediante sombras o un diseño más perfeccionado proporciona un aspecto más profesional o más cercano a la realidad. Sin embargo, a lo largo de la historia se han elaborado otras soluciones que modifican directamente el tipo de *tile* pasando a ser romboidales y hexagonales.

A continuación, daremos a conocer las principales modificaciones a tener en cuenta para estos dos casos. Para ello, en primer lugar veremos las diferencias entre la versión perpendicular (la vista hasta ahora) con la isométrica y esta última con la hexagonal.

2.6.1. *Isometric tile map*

A primera vista podemos apreciar dos grandes diferencias entre los juegos en vista isométrica y perpendicular: el tamaño de las *tiles* y la configuración del mapa, las coordenadas de cada celda. El tamaño de las *tiles* isométricas mantiene una proporción de 2 a 1, y las coordenadas de cada una en el mapa difieren en gran medida.

Podemos observar este hecho en la siguiente ilustración. A modo de ejemplo, escogeremos uno de los tamaños de celdas más utilizados: 32x32 y 64x32.



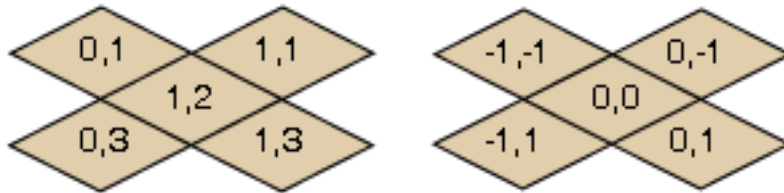
A la izquierda, el *tile based map* convencional. A la derecha, su versión isométrica

Como podemos ver se nos presentan dos nuevos problemas:

- La selección de celda ya no es tan trivial como antes, donde tan sólo era necesario hacer una división por el tamaño de la celda para saber de cuál se trata.
- El algoritmo de pintado también debe ser modificado.

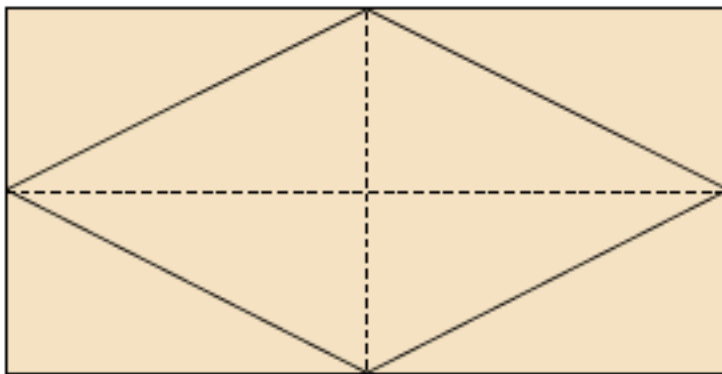
Selección de celda

Si observamos de nuevo la imagen anterior, podemos apreciar cómo se establece una relación de vecindad entre cada una de las celdas. Ésta puede ser descrita de la manera siguiente.



Esta imagen nos puede dar una primera pista de cómo solucionar el problema. Al elegir una celda haciendo la misma división que para el caso de celdas rectangulares, podemos acertar en el 50% de los casos y, en caso de error, sabemos que será una de las cuatro posibilidades restantes.

El siguiente paso, pues, consistirá en saber cuándo la división es suficiente y diferenciar y tratar cada uno de los cuatro casos posibles. Para ello proseguiremos el estudio con la siguiente imagen:



La técnica consiste en dividir la celda candidata en cuatro cuadrantes tal y como se acaba de mostrar. Al hacer esto podemos ver cómo han quedado perfectamente definidas cuatro rectas (las aristas del rombo). Primero debemos ubicar el punto en cuestión sobre el cuadrante pertinente y, a continuación, calcular si éste se encuentra por encima o por debajo de la recta. En caso positivo o negativo, se aplicará el incremento o decremento en una unidad según el cuadrante que sea, respecto la coordenada de la celda candidata.

El código resultante quedaría de la siguiente manera. El punto (mx, my) representa la posición del ratón.

```
//Calculamos la tile candidata
```

```
tx = mx >> 6;
ty = (my >> 5) << 1;

//Calculamos el resto
rx = mx % 64;
ry = my % 32;

//Comprobamos los cuatro cuadrantes
//Left
if(rx < 32)
{
    //Up
    if(ry < 16)
    {
        if( (16-(rx>>1)) > ry )
        {
            dx = -1;
            dy = -1;
        }
    }
    //Down
    else
    {
        if( (16+(rx>>1)) < ry )
        {
            dx = -1;
            dy = 1;
        }
    }
}
//Right
else
{
    rx -= 32;
    //Up
    if(ry < 16)
    {
        if( (rx>>1) > ry )
        {
            dx = 0;
            dy = -1;
        }
    }
    //Down
    else
    {
        if( (32-(rx>>1)) < ry )
```

```

    {
        dx = 0;
        dy = 1;
    }
}
}
//Actualizamos posible incremento
tx += dx;
ty += dy;

```

En muchas ocasiones, un cálculo en tiempo real puede ser ahorrado y venir preprocesado. Pensemos que estamos haciendo un juego de estrategia en tiempo real (RTS). En este contexto, la selección de unidades será una operación que el usuario realizará de manera continuada y por ello, puede ser justificado tenerla precalculada. ¿Cómo? Pues por ejemplo, mediante la imagen que mostramos a continuación:



Imagen con información relativa a la selección de celda isométrica precalculada.

Este recurso debería ser del tamaño idéntico al de nuestra *tile* (por ejemplo, de 64×32 píxeles) y debería utilizarse mediante la siguiente técnica. Lo primero que deberíamos hacer es cargar esta imagen en una matriz que podría ser de enteros, por ejemplo, asignando un valor a cada color (0 = blanco, 1 = rojo, etc.). A la hora de implementar la selección de celda, calcularíamos la división tal y como hacíamos antes y con el resto de la división accederíamos a esta nueva matriz. Según el color (valor de la matriz) resultante ahora sabremos en qué área nos encontramos. Sabremos si el cálculo de la *tile* candidata es final (nos indexamos en la zona blanca de la imagen) o qué modificación deberemos realizar tal y como hacíamos antes (el resto de la división apunta a uno de los cuatro colores). Este podría ser un ejemplo de implementación:

```

//Calculamos la tile candidata

tx = mx >> 6;

ty = (my >> 5) << 1;

//Calculamos el resto

rx = mx % 64;

ry = my % 32;

```



```
//Comprobamos las diferentes areas

switch(mapa[rx][ry])

{

    case BLANCO:    dx = 0;  dy = 0;  break;

    case ROJO:      dx = -1; dy = -1; break;

    case VERDE:     dx = -1; dy = 1;  break;

    case AMARILLO: dx = 0;  dy = -1; break;

    case AZUL:      dx = 0;  dy = 1;  break;

}

//Actualizamos posible incremento

tx += dx;

ty += dy;
```

Otra posible solución, utilizando esta misma técnica, podría ser inicializar la matriz en cuestión directamente en código. Es decir, sin hacer uso de un recurso externo (nos ahorramos la imagen).

Algoritmo de pintado

El algoritmo de pintado sufre una pequeña modificación. Debemos tener en cuenta si la fila es par o impar, para efectuar el desplazamiento de media celda y, además, tener en cuenta que el incremento de las variables para cada uno de los ejes ya no es el mismo.

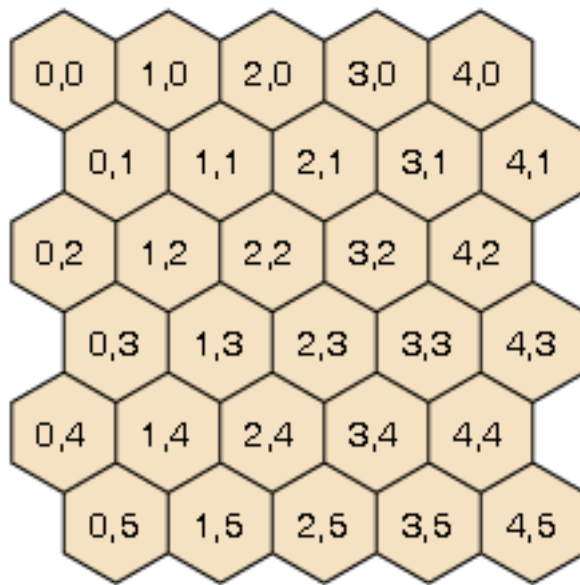
El recorrido del algoritmo queda como sigue:

```
for(int j=Yo; j<=Yf; j++)
{
    for(int i=Xo; i<=Xf; i++)
    {
        tile = map->data[i][j];
        RenderImage( map->images[tile], (i<<6) + ((j%2)<<5), (j<<4));
    }
}
```

En el código presentado podemos ver cómo en el eje de las X nos desplazamos 32 píxeles (media anchura de *tile*) según la fila es par o impar y las *tiles* se sitúan cada 64 píxeles (la anchura de la *tile*). En cambio, en el eje Y el incremento es de 16 píxeles (media altura de *tile*).

2.6.2. Hexagonal-based tile map

Para el caso de *tiles* hexagonales, se procede siguiendo la misma técnica que el isométrico pero teniendo en cuenta sus peculiaridades. De este modo, ahora la indexación de celdas es como presentamos en la ilustración siguiente.

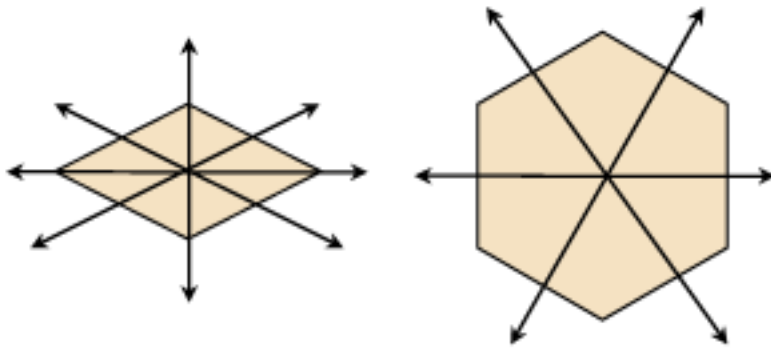


El algoritmo de selección de celda es muy similar al anterior; sin embargo, ahora no es suficiente con los cuatro cuadrantes. Asimismo, el caso del pintado resulta también muy similar.

Actividad

Pensar los cambios que deberán efectuarse en el código anteriormente presentado para ambos casos.

Finalmente, debemos tener en cuenta la limitación de movimientos que tiene lugar en este tipo de mapas con celdas hexagonales. Tanto en el caso perpendicular como en el isométrico disponemos de ocho posibles direcciones para ir de celda en celda. Sin embargo, la versión hexagonal limita este valor a seis, como se aprecia en la imagen siguiente:



En la imagen de la izquierda, una *tile* isométrica, con las ocho posibles direcciones que se pueden tomar desde ésta. En la derecha, una *tile* hexagonal con sus seis posibles direcciones.

2.6.3. Tiles apilables

Con independencia, además, de que el juego contenga *tiles* rectangulares, isométricas o hexagonales, el escenario puede ser modelado mediante diferentes capas de *tiles*. Es una técnica de visualización 2D aún más enriquecida, para acercarnos un poco más al 3D. Mediante diferentes capas de *tiles* apiladas unas sobre otras, podemos simular el volumen y ofrecer en un entorno de jugabilidad 2D (el juego sigue siendo completamente 2D), una simulación visual en 3D, sin deformación en los ejes, con una proyección ortoédrica.

Veamos un ejemplo de ello mediante el siguiente ejemplo del juego "Piranhas":



"Piranhas", un ejemplo de juego con vista isométrica y *tiles* apiladas.

Como podemos ver en la imagen, el tratamiento del pintado no difiere mucho de la versión original que hemos comentado con anterioridad. Ahora, además de realizar un recorrido en el orden correcto, es necesario apilar las *tiles* de las sucesivas capas también en el orden correcto. El objetivo no es otro que el de superponer los objetos convenientemente para que la sensación de profundidad y volumen sea correcta.

3. Física

En este apartado daremos a conocer los principales aspectos de la física que tienen lugar en el desarrollo de videojuegos 2D. En primer lugar se mostrarán los aspectos de cinemática, fuerza, gravedad, resorte y colisiones elásticas. A continuación daremos a conocer diferentes técnicas para la detección de colisiones. En el material de "Videojuegos 3D", trataremos este tema con más detalle y mostraremos algunas referencias de librerías y motores físicos a tal efecto.

3.1. Conceptos

3.1.1. Cinemática

La cinemática es la parte de la mecánica clásica que estudia el movimiento de los cuerpos sin tener en cuenta las causas que lo producen, limitándose, esencialmente, al estudio de la trayectoria en función del tiempo. A continuación caracterizaremos las diferentes variables que tienen lugar: posición, velocidad, aceleración y tiempo.

La posición de un objeto es:

$$\vec{r} = \vec{r}_0 + \vec{v} t + \frac{1}{2} \vec{a} t^2$$

siendo \vec{r}_0 un vector desde el origen de coordenadas hasta la posición inicial del objeto, \vec{v} la velocidad a la que se desplaza, \vec{a} la aceleración que sufre y t el tiempo transcurrido desde que nos encontrábamos en \vec{r}_0 hasta ahora, que nos encontramos en la posición \vec{r} (siendo éste un vector desde el origen de coordenadas hasta la posición actual).

Cuando hay aceleración, debemos actualizar la velocidad:

$$\vec{v} = \vec{v}_0 + \vec{a} t$$

siendo \vec{v}_0 la velocidad inicial, \vec{a} la aceleración que sufre y t el tiempo transcurrido.

La aceleración nos vendrá determinada por las fuerzas que se apliquen sobre el objeto, que en las siguientes secciones están detalladas. Si un objeto sufre más de una aceleración (por ejemplo, un objeto colgado de un muelle recibe una aceleración por la tracción del muelle y otra por la gravedad) simplemente se han de sumar para obtener la aceleración total:

$$a = \sum_{i=1}^N \vec{a}_i$$

3.1.2. Fuerza

Todo videojuego que incluya valores físicos, necesitará hacer uso de la ecuación que permite traducir una fuerza ejercida sobre un objeto a una aceleración del movimiento del mismo. Como la exactitud no es un requisito, lo más correcto es hacer uso de la famosa "segunda ley de Newton" que ofrece una aproximación muy buena en los casos en que los objetos se desplacen a velocidades muy inferiores a las de la luz. Dicha ecuación es:

$$\sum_{i=1}^N \vec{F}_i = m \vec{a}$$

que aislando la componente que nos interesa, resulta:

$$\vec{a} = \frac{\sum_{i=1}^N \vec{F}_i}{m}$$

donde $\sum_{i=1}^N \vec{F}_i$ es la suma de todas las fuerzas que son ejercidas sobre el objeto y m es su masa. Esto provoca la aceleración \vec{a} sobre el objeto, que condicionará su movimiento.

3.1.3. Gravedad

Como es bien sabido, cualquier masa atrae a toda otra masa debido a la gravedad que ejerce la primera sobre el resto. Usando la "teoría de la gravitación universal" de Newton, podemos calcular el valor de la fuerza de atracción ejercida por un objeto 1 sobre otro objeto 2:

$$\vec{F}_{12} = -G \frac{m_1 m_2}{r^2} \hat{r}_{12}$$

donde G es la constante de gravitación, cuyo valor es $6,673 \times 10^{-11} \text{N} \cdot \text{m}^2 \text{kg}^{-2}$, m_1 y m_2 son las masas de los dos objetos, r es la distancia que los separa (es decir, $\|\vec{r}_2 - \vec{r}_1\|$ si \vec{r}_1 es la posición del objeto 1 y \vec{r}_2 la del objeto 2) y \hat{r}_{12} es el vector unitario que se dirige del objeto 1 al objeto 2:

$$\left(\hat{r}_{12} = \frac{\vec{r}_2 - \vec{r}_1}{\|\vec{r}_2 - \vec{r}_1\|} \right).$$

Si no nos interesa obtener directamente la fuerza, también podemos calcular únicamente el campo gravitatorio del objeto 1 en la región del objeto 2 de la siguiente manera:

$$\vec{g}_{12} = -G \frac{m_1}{r^2} \hat{r}_{12}$$

La única diferencia con la fórmula previa es la desaparición del término m_2 .

Sabiendo esto, se deduce que, una vez obtenida \vec{g}_{12} , calcular \vec{F}_{12} es sencillamente:

$$\vec{F}_{12} = m_2 \vec{g}_{12}$$

Esta fórmula es idéntica a la ecuación expuesta en el apartado anterior:

$$\left(\vec{F} = m \cdot \vec{a} \right).$$

Se ve que el campo gravitatorio \vec{g}_{12} actúa en el objeto 2 de igual manera que una aceleración, es decir, un campo gravitatorio es una aceleración.

3.1.4. Resorte

El comportamiento de los resortes es realmente complejo. Dependiendo del material, el método de fabricación, la forma, el estado y una infinidad más de parámetros, un resorte se comportará de una u otra manera. Resulta aún más impredecible cuando se le somete a deformaciones extremas (extenderlo

o comprimirlo mucho), caso en el que incluso se produce una modificación permanente de la forma del resorte. Esto es lo que se conoce como el límite de elasticidad.

Afortunadamente, desde la posición de equilibrio (cuando no actúa ninguna fuerza) hasta dicho límite de elasticidad, un resorte se comporta de manera bastante simple siguiendo la aproximación lineal de la famosa ley de Hooke:

$$\vec{F} = -k \vec{x}$$

donde k es la constante de rigidez propia de cada resorte y \vec{x} es el desplazamiento efectuado desde la posición de equilibrio. El signo negativo indica que la fuerza ejercida por el resorte siempre está en oposición a la dirección del desplazamiento: es una fuerza de "restauración", ya que tiende al equilibrio. Si no se contemplan pérdidas de energía, al enganchar una masa en el extremo libre del resorte, tensarlo y luego soltarlo, se producirá en dicho resorte un movimiento oscilatorio perpetuo, con una frecuencia de:

$$\omega = \sqrt{\frac{k}{m}}$$

3.2. Saltos

Una vez hemos visto los conceptos físicos básicos, trataremos uno de los temas que más suele aparecer en los juegos 2D, sobre todo en lo que se refiere al género plataformas: los saltos.

Para implementar el salto de una entidad en el escenario, podemos optar por hacer uso de trigonometría o, como suele solventarse en muchas situaciones, mediante una aproximación. Empezaremos con esta segunda opción.

3.2.1. Aproximación

Dado que los juegos 2D suelen trabajar en un espacio discreto –el que nos proporcionan los píxeles de la pantalla–, éstos suelen manipular datos casi siempre de tipo entero, en lugar de coma flotante. Por temas de simplicidad, como de buen principio solemos antepoñernos esta limitación, puede ser razonable utilizar un método aproximado para implementar los saltos en lugar de una solución exacta que debiéramos obtener a partir de números reales. Pues bien, esta simulación es fácil de conseguir dando los pasos siguientes:

Para calcular el salto de un personaje, necesitamos conocer su posición, indicar una velocidad y el valor de la gravedad del mundo físico donde se encuentra. A partir de estos valores, en cada *tick* del juego debemos actualizar la nueva posición según su velocidad (en este caso proporcionada por el impulso del salto) y esta misma velocidad (o fuerza del salto) que se ve afectada por la gravedad.

Veremos esto con un ejemplo. Sin embargo, antes es necesario establecer nuestro sistema de coordenadas. En las API de los juegos en dos dimensiones, lo normal es encontrarse que la posición (0, 0) se encuentra en la esquina superior izquierda o en la esquina inferior izquierda. Vamos a suponer que nos hallamos en la segunda situación planteada. Esto repercutirá sobre los signos del ejemplo que a continuación tratamos, pero, sin embargo, es fácil de adaptar en caso de que nos encontremos en la otra situación.

Dado que nuestro origen de coordenadas se encuentra en la parte inferior izquierda, hacer un salto significa tomar una velocidad en el eje Y positiva (subiremos). Y por tanto, la fuerza de la gravedad será negativa (nos empujará hacia abajo). Por ejemplo, si la velocidad inicial del personaje fuera de 10 unidades (de píxeles dada la situación planteada), y una gravedad de valor -2, el proceso mencionado anteriormente en código quedaría como sigue.

En primer lugar, cuando detectamos que hay que realizar el salto, por ejemplo al pulsar la tecla de espacio, efectuaríamos una inicialización como ésta:

```
//inicialización

jump = true; //variable de control

speed = 10; //velocidad inicial del salto

gravity = -2; //gravedad de nuestro mundo físico
```

Y en el método que se encarga de la lógica del personaje, añadiríamos el código siguiente:

```
//lógica del salto

if(jump) {

    //actualizamos nueva posición según velocidad

    y = y + speed;

    //la velocidad queda atenuada por la fuerza de la

    //gravedad a cada tick
```

```

    speed = speed + gravity;

}

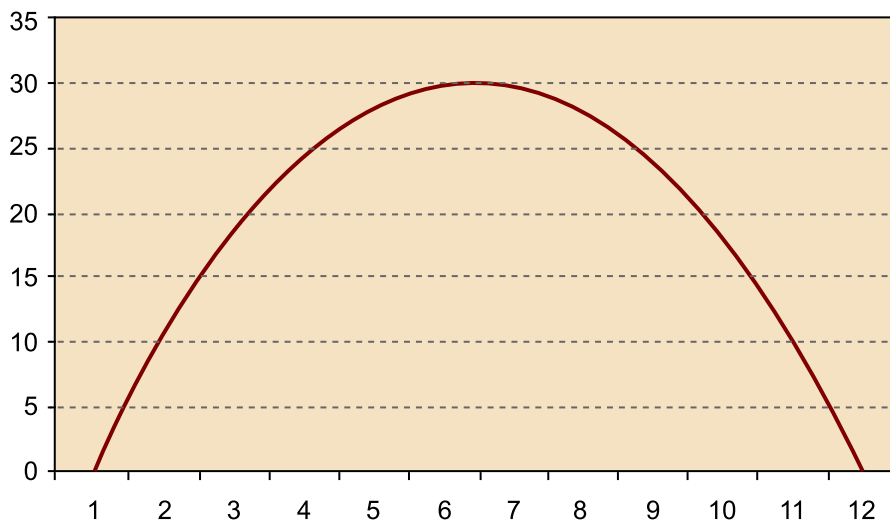
```

Para ver el funcionamiento de esta implementación, en la siguiente tabla mostraremos la reproducción del salto para cada *tick* del juego.

Iteración	1	2	3	4	5	6	7	8	9	10	11	12
Y	0	10	18	24	28	30	30	28	24	18	10	0
Speed	10	8	6	4	2	0	-2	-4	-6	-8	-10	0

En la tabla anterior puede apreciarse cómo en cada iteración la posición varía como si de una función periódica se tratase. En el último paso, se produciría la colisión de nuevo con la plataforma de donde procedemos y, por tanto, la velocidad en el eje Y pasaría a ser nula. Si esto no ocurriera, lo que se suele hacer, por términos de jugabilidad, es que a partir de este punto, la velocidad en Y es constante según el último valor adquirido (es decir, -10 unidades).

Una representación gráfica de este movimiento, tomando como movimiento en el eje X una velocidad constante, esto es, una situación normal en la que el jugador pulsa la tecla de salto y sigue apretando una de las teclas de movimiento horizontal, podría ser la que se muestra en la siguiente figura:



Representación gráfica del salto haciendo uso de un procedimiento de aproximación. En el eje Y, se puede apreciar la posición o altura en la que se encontrará el personaje, y mediante el eje X, su fluctuación entre cada paso de la iteración.

Tal y como puede apreciarse, a pesar de tratarse de una representación en el espacio discreto, la simulación es bastante precisa y suficiente para su implantación en un juego de dos dimensiones.

3.2.2. Trigonometría

Otra alternativa a la implementación del salto sería lo que podríamos denominar como la simulación exacta a partir de trigonometría, haciendo uso de una función sinusoidal. La estrategia a seguir consiste en definir la altura del personaje durante el salto mediante la función seno entre los ángulos 0° y 180° , multiplicando su valor por la altura máxima que puede alcanzar la entidad en cuestión.

Al igual que en el caso anterior, deberemos tener una inicialización en la cual se da a conocer que nos encontramos en el estado salto. Además, será necesario almacenar la posición altura inicial del salto e inicializaremos un atributo para el ángulo a cero. La implementación quedaría como sigue:

```
//inicialización

jump = true; //variable de control

alpha = 0; //ángulo inicial

y_base = y; //altura base sobre la que se hace el salto

y_max = 30;
```

Ahora nos falta añadir el código pertinente a la lógica para implementar el paso a paso de nuestro salto. Esta actualización deberá incrementar el valor del ángulo sobre el cual, con la función matemática *seno*, calculamos nuestra altura y asignaremos la nueva altura del personaje. Nótese cómo para hacer un salto simétrico, el valor constante con el cual incrementamos nuestro ángulo debe ser divisor de 180. De hecho, el resultado de hacer la división entre 180 y el incremento del ángulo nos dará el número de pasos con el cual se hará el salto, es decir, la fluidez con la cual se efectuará. Aquí tenemos una posible implementación:

```
//lógica del salto

if(jump) {

    //actualizamos paso del salto

    alpha = alpha + 15;

    //calculamos nueva altura

    y = y_base + y_max*sin(radians(alpha));

    //comprobamos si fin de salto
```

```

    jump = alpha == 180;

}

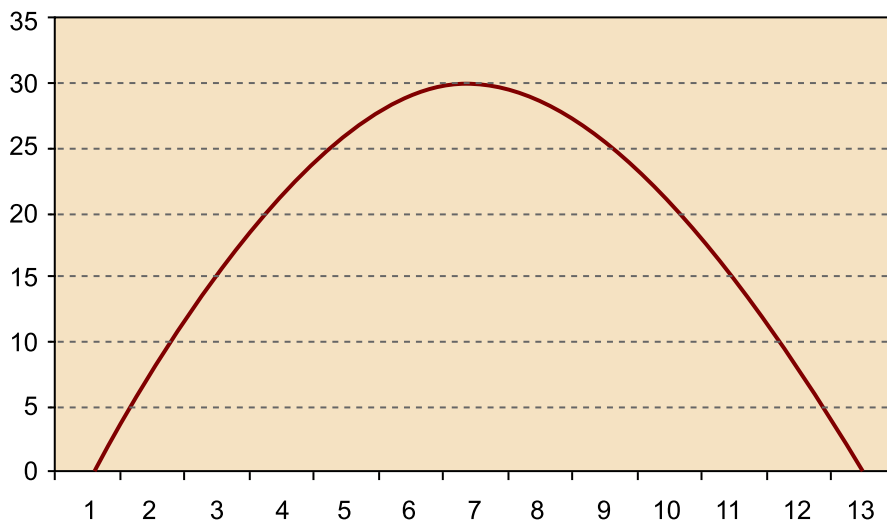
```

En la siguiente tabla mostramos el funcionamiento paso a paso del conjunto de iteraciones de la implementación del salto mediante la función *sinus*.

Iteración	1	2	3	4	5	6	7	8	9	10	11	12	13
Alpha	0	15	30	45	60	75	90	105	120	135	150	165	180
Y	0	7,8	15	21	26	29	30	29	26	21	15	7,8	0

Como podemos apreciar, las diferencias respecto al modelo de cálculo aproximado anterior son pequeñas. Antiguamente, era una práctica común no utilizar funciones trigonométricas dado su coste en número de ciclos. En su lugar, se utilizaban tablas precalculadas, con lo cual se reducía el coste a un simple acceso a memoria. Es una optimización que apenas ocupa espacio, y de esta tabla, luego se hace un uso frecuente, ya sea para saltos o cálculos de trayectorias, disparos, etc.

Una representación gráfica –como hemos hecho en el caso anterior y en la misma situación– tomando como movimiento en el eje X una velocidad constante, podría ser la que se muestra en la siguiente figura:



Representación gráfica del salto haciendo uso de la función trigonométrica *seno*. En el eje Y se puede apreciar la posición o altura en la que se encontrará el personaje y mediante el eje X, su fluctuación entre cada paso de la iteración.

Como podemos apreciar las diferencias no son muy notorias, si bien para este segundo caso hemos escogido un valor de incremento del ángulo de magnitud 15, para provocar la similitud con el caso anterior.

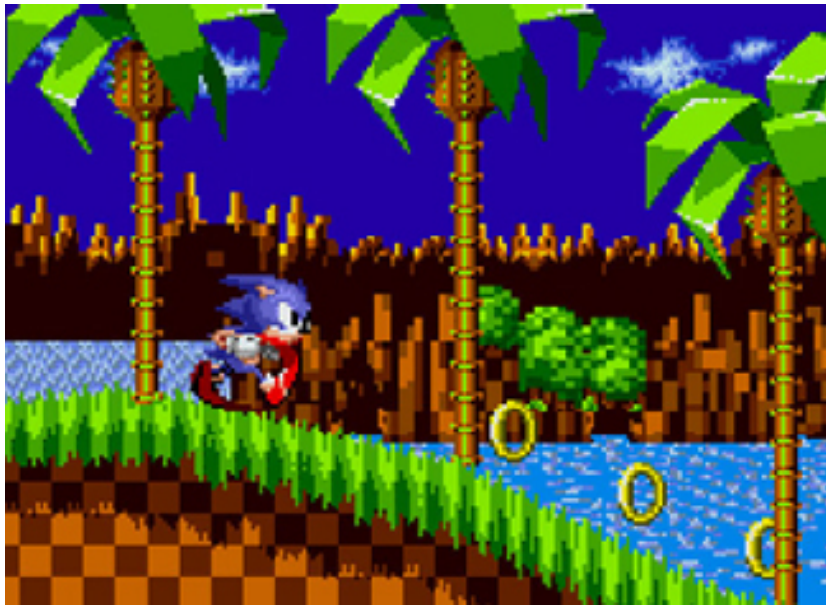
3.2.3. *Gameplay*

A la implementación del salto ahora sería necesario añadir las reglas de interacción propias de nuestro *gameplay* con el personaje. A partir de cualquiera de las dos implementaciones escogidas, podremos incorporar ciertos aspectos que implicarán retoques en los algoritmos planteados. Algunas posibilidades que nos podemos plantear son las que enumeramos en el siguiente listado:

- Posibilidad de hacer un doble salto, o saltos múltiples, a partir de una barra de energía o un valor determinado.
- Control de colisiones con el escenario: sólo mientras el personaje desciende o tanto cuando sube (con la cabeza) como cuando baja (con los pies).
- Velocidad constante máxima en caída o fuerza de la gravedad.
- Los saltos no siempre son voluntarios (producidos al pulsar una tecla), pueden venir dados por impactos (nos dañan) o mediante una plataforma que nos hace rebotar, por ejemplo. Son reacciones cuya implementación es idéntica a la del saltar.

Por lo tanto, sirva como ejemplo el salto, no implementado con una velocidad constante en los dos ejes, sino con un poco de gracia, como un aspecto importantísimo de la física que puede cambiar por completo la jugabilidad y agrado de nuestro proyecto. Esto puede ser extrapolado e implementado con la misma facilidad a otros contextos, como podrían ser las acciones de acelerar, derrapar, frenar, atraer, etc. Todas ellas enriquecerán muchísimo la experiencia de nuestro juego y fundamentarán las claves del éxito y aceptación por parte del público.

Un buen ejemplo de ello puede ser "Sonic the Hedgehog" de Sega, un juego de plataformas con un altísimo grado de jugabilidad que permite realizar acciones básicas como son acelerar, frenar, correr, impactar, deslizarse, velocidad hipersónica (de aquí el origen del nombre del juego), derrapar, saltar, rebotar, nadar, etc. Acciones todas ellas fáciles de implementar y que dotan al juego de una gran interacción.





"Sonic the Hedgehog" de Sega, uno de los juegos por excelencia de plataformas más conocidos gracias a sus divertidas pruebas físicas. En las imágenes se puede apreciar, en este orden, la implementación física del rebote al alcanzar un ítem dañino, la aceleración y el derrape.

3.3. Colisiones elásticas

Una colisión elástica es aquella en la que no hay pérdida de energía cinética y, por tanto, se conserva la cantidad de movimiento. Esto no es del todo real, pues siempre hay pérdidas en forma de sonido, calor, deformación, etc.; pero aun así, ofrece la base que permitiría implementar colisiones inelásticas.

Empecemos, pues, con la definición de la cantidad de movimiento:

$$\vec{p} = m \vec{v}$$

que es lo que conceptualmente puede entenderse como el impulso que lleva un objeto debido a su movimiento y masa.

También necesitamos la energía cinética:

$$E_c = \frac{1}{2} m \|\vec{v}\|^2$$

que es la energía extra que tiene un objeto debido a su movimiento, la energía que ha ido ganando durante su aceleración desde el reposo hasta su velocidad actual.

Para aclarar de antemano las próximas ecuaciones, exponemos aquí la sintaxis especial que será usada en ellas: con un ₁ o un ₂ como subíndice, nos referiremos siempre al mismo objeto de los dos que intervienen en la colisión; y con

una prima, expresada con una comilla simple ('), denotaremos un valor después del choque. Con lo anteriormente dicho, la conservación de la cantidad de movimiento se expresa como:

$$m_1 v_1 + m_2 v_2 = m_1 v_1' + m_2 v_2'$$

Y la conservación de la energía cinética es:

$$\frac{1}{2} m_1 v_1^2 + \frac{1}{2} m_2 v_2^2 = \frac{1}{2} m_1 v_1'^2 + \frac{1}{2} m_2 v_2'^2$$

Haciendo uso de mucha álgebra, pueden combinarse ambas ecuaciones para encontrar la solución de los valores que nos interesan, a saber: las velocidades finales v_1' y v_2' . Dichas ecuaciones son:

$$v_1' = \frac{v_1(m_1 - m_2) + 2m_2 v_2}{m_1 + m_2}$$

$$v_2' = \frac{v_2(m_2 - m_1) + 2m_1 v_1}{m_1 + m_2}$$

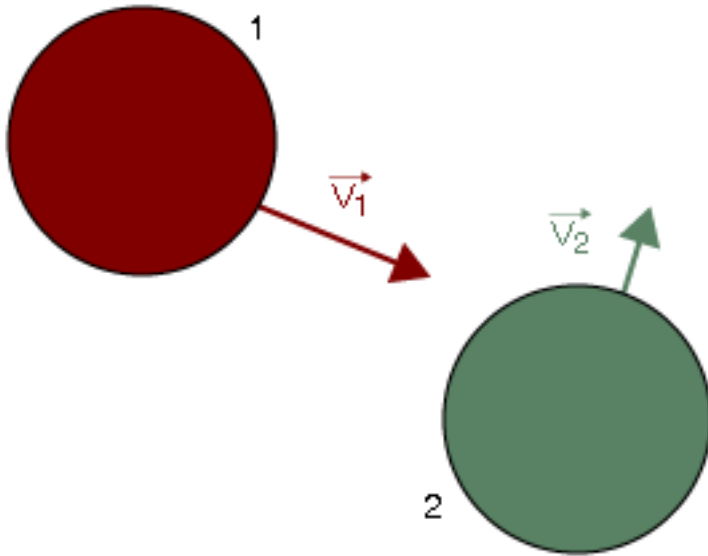
Como puede observarse, sólo hemos obtenido las magnitudes de las velocidades resultantes en una única dimensión; aún nos resta extrapolar a colisiones en dos dimensiones usando velocidades con direccionalidad, para así poder expresar completamente a \vec{v}_1' y \vec{v}_2' . En los siguientes pasos se detalla el método a seguir para la resolución.

3.3.1. Idea conceptual

En la siguiente figura puede observarse una situación modelo de dos objetos que van a colisionar.

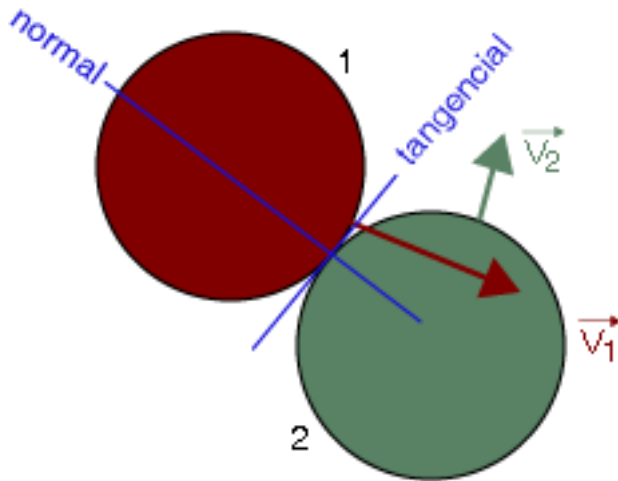
m1 y m2 iguales

Nótese la peculiaridad del caso en el que m_1 y m_2 sean iguales. Véase como ejemplo la ecuación de v_1' : el término v_1 ($m_1 - m_2$) se anula, el divisor puede expresarse como $2m_2$ y simplificando únicamente nos queda v_2 . Así pues, con objetos de igual masa, las velocidades simplemente se intercambian.



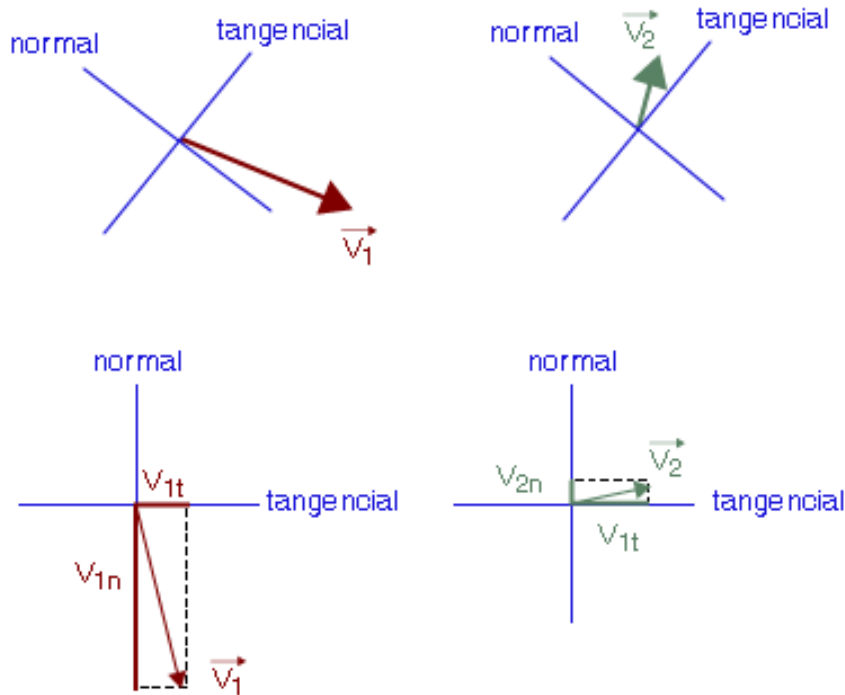
Disposición original de los objetos que van a colisionar

El procedimiento que seguiremos para el cálculo de sus velocidades finales consistirá en proyectar las velocidades iniciales de cada uno de los objetos en los vectores normal (perpendicular) y tangencial al punto de colisión. La siguiente figura muestra la disposición de los vectores normal y tangencial según el modelo de ejemplo.



Objetos en el momento de la colisión

Y en la siguiente figura pueden verse las proyecciones de \vec{v}_1^i y \vec{v}_2^i sobre ellos.



Descomposición de las velocidades iniciales

Esta descomposición en normal y tangencial la haremos debido a que la componente tangencial de las velocidades no se ve afectada en absoluto por el choque, mientras que la componente normal actúa como si se tratase de una colisión en una única dimensión, tipo de colisión que ya sabemos resolver con las fórmulas antes enunciadas.

3.3.2. Vectores normal y tangencial

Para obtener un vector que sea normal a la superficie de colisión de los círculos 1 y 2, simplemente usaremos las coordenadas de sus centros de la manera siguiente:

$$\vec{n} = \langle x_2 - x_1, y_2 - y_1 \rangle$$

donde las coordenadas (x_1, y_1) son el centro del círculo 1 y (x_2, y_2) , las del círculo 2. El resultado \vec{n} obtenido, sin embargo, es un vector que se dirige del centro del círculo 1 al centro del círculo 2. Nosotros sólo necesitamos su direccionalidad, no su módulo, y por ello lo transformamos a un vector unitario:

$$\hat{u}_n = \frac{\vec{n}}{\|\vec{n}\|}$$

Para la obtención del vector unitario tangente a la superficie de colisión, sólo necesitamos el vector unitario normal a la misma superficie de colisión que acabamos de calcular:

$$\hat{u}_t = \langle -\hat{u}_{n,y}, \hat{u}_{n,x} \rangle$$

donde el subíndice x indica la dimensión x del vector \hat{u}_n , y el subíndice y la dimensión y del mismo.

3.3.3. Descomposición de velocidades

Para separar los componentes normal y tangencial de \vec{v}_1 y \vec{v}_2 , debemos hacer sendos productos escalares:

$$v_{1n} = \vec{v}_1 \cdot \hat{u}_n$$

$$v_{1t} = \vec{v}_1 \cdot \hat{u}_t$$

$$v_{2n} = \vec{v}_2 \cdot \hat{u}_n$$

$$v_{2t} = \vec{v}_2 \cdot \hat{u}_t$$

donde se entiende que los subíndices se han usado de la forma obvia: v_{1n} se refiere a la componente normal de \vec{v}_1 , mientras que v_{2t} a la componente tangencial de \vec{v}_2 ; el resto no requiere más explicación.

Nótese la ausencia de notación vectorial para v_{1n} , v_{1t} , v_{2n} y v_{2t} ya que son resultado de un producto escalar que, recordemos, siempre devuelve una magnitud escalar (un único número invariable en cualquier sistema de referencia).

Para llevar a cabo un producto escalar de un vector \vec{a} sobre un vector \vec{b} , se procede de la manera siguiente:

$$\vec{a} \cdot \vec{b} = \|\vec{a}\| \|\vec{b}\| \cos \theta$$

dónde θ es el ángulo que separa ambos vectores. Para calcularlo, puede hacerse de la manera siguiente:

$$\theta = \theta_a - \theta_b$$

Y para obtener la θ de un vector tenemos que hacer una arcotangente de la división de sus componentes x e y (y/x), teniendo muy en cuenta sus signos para no errar el cuadrante en que se halla el vector. Afortunadamente, para los programadores de videojuegos, los lenguajes más comúnmente usados (C, C++, Java) disponen de una función llamada `atan2` que realiza este cálculo para nosotros, devolviendo un ángulo en radianes con valor en el rango $[-\pi, \pi]$.

3.3.4. Componente tangencial final

Como ya se ha dicho anteriormente, la componente tangencial de las velocidades no se ve afectada en una colisión, por lo que su cálculo es inmediato:

$$v'_{1t} = v_{1t}$$

$$v'_{2t} = v_{2t}$$

3.3.5. Componente normal final

Como también se ha dicho ya, la componente normal de las velocidades actúa como si se tratase de una colisión en una única dimensión, por lo que las ecuaciones son iguales a las anteriormente presentadas:

$$v'_{1n} = \frac{v_{1n}(m_1 - m_2) + 2m_2v_{2n}}{m_1 + m_2}$$

$$v'_{2n} = \frac{v_{2n}(m_2 - m_1) + 2m_1v_{1n}}{m_1 + m_2}$$

3.3.6. Vectorización

Llegados a este punto, disponemos de las magnitudes de las componentes de las velocidades finales, pero son sólo valores escalares, por lo que necesitamos convertirlos en vectores:

$$\vec{v}'_{1x} = v'_{1x} \hat{u}_x$$

$$\vec{v}'_{1t} = v'_{1t} \hat{u}_t$$

$$\vec{v}'_{2x} = v'_{2x} \hat{u}_x$$

$$\vec{v}'_{2t} = v'_{2t} \hat{u}_t$$

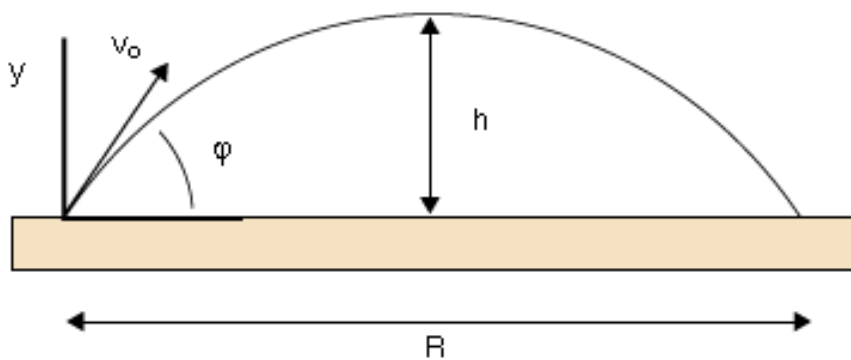
Y ahora que ya hemos obtenido las dos componentes de las velocidades finales, podemos construir los vectores que las representan:

$$\vec{v}'_1 = \vec{v}'_{1x} + \vec{v}'_{1t}$$

$$\vec{v}'_2 = \vec{v}'_{2x} + \vec{v}'_{2t}$$

3.4. Projectiles

En videojuegos sencillos suelen utilizarse proyectiles. Por lo que mostraremos con detalle las diferentes ecuaciones que nos pueden interesar al respecto.



Punto de origen e impacto a la misma altura

Las fórmulas que resultan de este contexto que debemos tener en cuenta son las siguientes:

$$x(t) = (v_0 \cos \varphi) t$$

$$y(t) = (v_0 \sin \varphi) t - \frac{g t^2}{2}$$

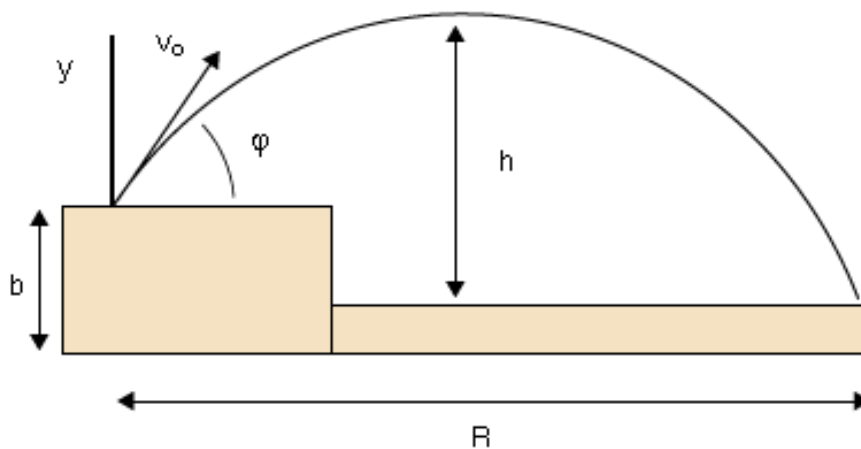
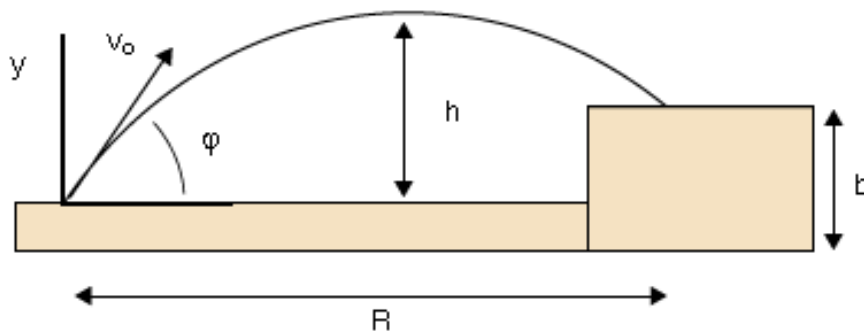
$$v_x(t) = v_0 \cos \varphi$$

$$v_y(t) = v_0 \sin \varphi - g t$$

$$v(t) = \sqrt{v_0^2 - 2g t v_0 \sin \varphi + g^2 t^2}$$

$$R = v_0 T \cos \varphi$$

Para calcular la altura máxima o el tiempo total hasta el pertinente impacto, debemos tener en cuenta, además, los dos siguientes casos.

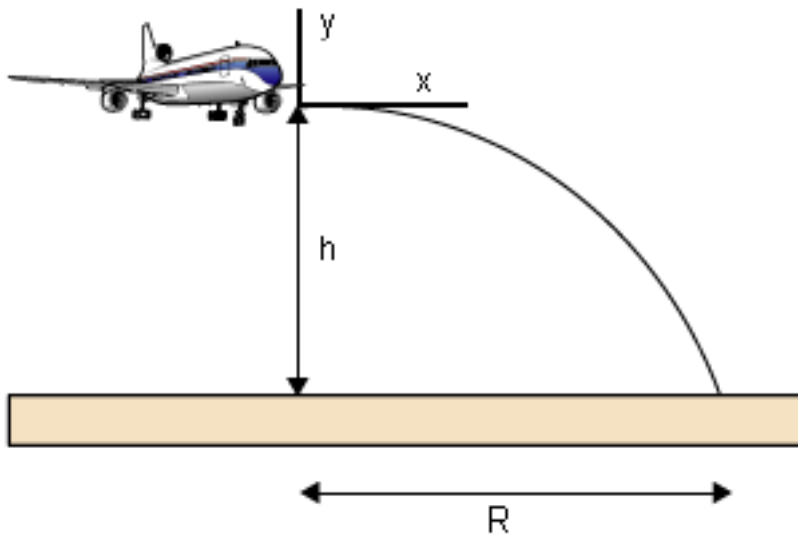


Punto de origen por encima del punto de impacto

Presentadas las diferentes posibilidades que podemos encontrar, las variables altura y tiempo para cada uno de los diferentes casos se muestran en la siguiente tabla:

	Caso 1	Caso 2	Caso 3
h	$\frac{v_0^2 \sin^2 \varphi}{2g}$	$\frac{v_0^2 \sin^2 \varphi}{2g}$	$b + \frac{v_0^2 \sin^2 \varphi}{2g}$
T	$\frac{2v_0 \sin \varphi}{g}$	$\frac{v_0 \sin \varphi}{g} + \sqrt{\frac{2(h-b)}{g}}$	$\frac{v_0 \sin \varphi}{g} + \sqrt{\frac{2h}{g}}$

Finalmente, se presenta un cuarto caso referido al lanzamiento de proyectiles desde un sistema en movimiento (por ejemplo, desde un avión). Nótese cómo la velocidad inicial del proyectil es horizontal e igual a la velocidad del vehículo desde donde es lanzado. La siguiente figura ilustra dicho caso y posteriormente se presenta el conjunto de ecuaciones necesarias para resolver este problema:



Proyectil lanzado desde un sistema en movimiento

$$x(t) = v_0 t$$

$$y(t) = h - \frac{g t^2}{2}$$

$$v_x(t) = v_0$$

$$v_y(t) = -g t$$

$$v(t) = \sqrt{v_0^2 + g^2 t^2}$$

$$h = \frac{g t^2}{2}$$

$$R = v_0 T$$

$$T = \sqrt{\frac{2h}{g}}$$

Nota

Cabe destacar que cuando v_0 es igual a cero, el problema se reduce a un caso de caída simple, en el que el proyectil se limita a caer verticalmente.

3.5. Detección de colisiones

Estudiaremos las técnicas de detección de colisiones más utilizadas hasta la fecha en los videojuegos en su versión 2D.

3.5.1. *Tile based*

Sin lugar a dudas, es la manera más simple y utilizada en los juegos antiguos. Se trata de comprobar si los objetos se encuentran en *tiles* adyacentes o en la misma, depende de lo que se desee.

En la próxima imagen del juego Dyna Blaster se puede apreciar la técnica presentada. En este caso las bombas se expanden hasta encontrar un muro, el personaje se mueve siempre que la próxima celda sea transitable y recoge bonificaciones cuando llega a situarse en una celda que contenga algún ítem.



3.5.2. Por píxel

Éste es el método más preciso, pero también es el menos eficiente. Se comprueba si los píxeles de un *bitmap* coinciden con los de otro.

En esta ilustración de ejemplo, podemos comprobar cómo, mediante la técnica de *pixel perfect*, el fantasma ha cazado a nuestro PacMan.

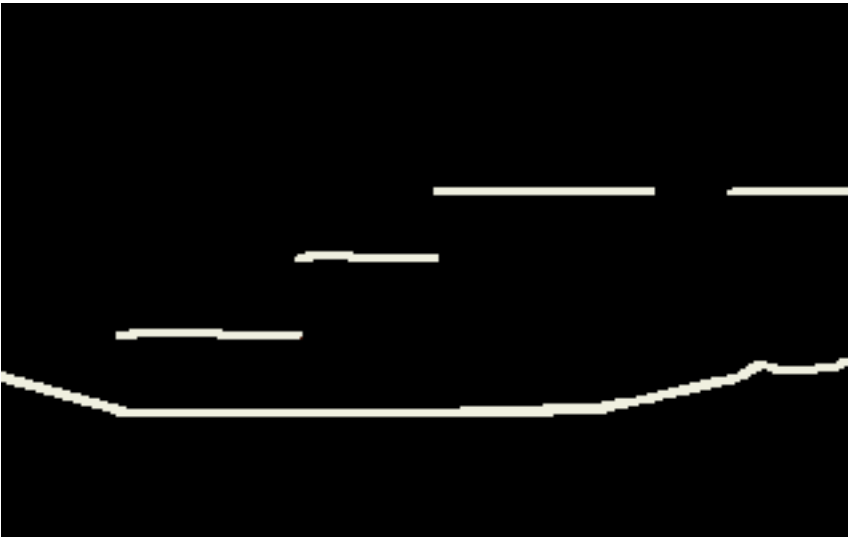


Sin embargo, este sistema puede ser de gran utilidad cuando un determinado personaje debe interactuar con un escenario de frontera irregular. Por ejemplo, dos casos totalmente diferentes: los juegos de plataformas y los de conducción.

Según qué tipo de juegos basados en plataformas pueden disponer de un mapa cuyo escenario, si bien puede ser descrito mediante *tiles*, el contorno sobre el cual debemos interactuar sea irregular. En estos casos, la ayuda de un *bitmap* auxiliar resulta imprescindible si queremos ofrecer un correcto sistema de colisiones. Este recurso gráfico nos indicará cuándo tenemos una zona con la cual debemos apoyarnos, codificada con un cierto color y, en caso contrario, utilizando otro color. Veamos un ejemplo gráfico.



Pantalla generada, ya sea a través de la parte del *bitmap* que forma toda la escena, o por la composición de diferentes *bitmaps* mapeados.



Bitmap de superficies: si el píxel en una determinada posición es blanco, el jugador permanecerá en su misma altura; en caso contrario, su altura se verá decrementada hasta que nos encontremos en la situación del primer punto.



En esta situación, el punto de control (necesariamente situado en la parte inferior del personaje) se encuentra sobre un píxel de color blanco. Por lo tanto, el personaje no caerá.

La detección de colisiones en el caso del género de conducción en modo perspectiva perpendicular también puede beneficiarse de esta técnica. En estos casos, incluso podemos llegar a contar con otro recurso gráfico que actúa de máscara y que describe el área del vehículo en cada una de las orientaciones en la que se nos presenta. De esta manera, no sólo contamos con la máscara que nos indica si hay colisión con el entorno, sino que también disponemos de la del propio jugador que puede colisionar.

Este algoritmo de detección de colisiones se hace imprescindible para juegos tipo "Lemmings", de Psygnosis, o "Worms", de Team17, en los cuales el píxel como unidad física atómica tiene una importancia vital, es decir, tiene consecuencias sobre el devenir de la partida.

Ejemplo

La presencia o no de un píxel del modelo físico puede hacer que un proyectil nos alcance (un píxel está entre medio de una entidad y la trayectoria de un proyectil en movimiento) o puede hacer posible alcanzar una plataforma, por citar dos posibles ejemplos.

Presentamos a continuación en la siguiente figura, dos ejemplos del estado de la máscara que podríamos tener para sendos juegos en dos momentos diferentes de la partida. En color blanco simbolizamos las áreas colisionables y en color negro las zonas no colisionables.



Juego "Lemmings" de Psygnosis, donde la detección de colisión por píxel es imprescindible.



Posible modelo físico del estado de la partida de la figura anterior.



Juego "Worms" de Team17, donde podemos apreciar la necesidad de detección de colisión por pixel.



Posible modelo físico del estado de la partida de la figura anterior.

En ambos ejemplos podemos comentar varias cosas. En primer lugar, la diferenciación que es necesario realizar a la hora de computar las físicas. Por una parte tenemos el cálculo con el escenario y aquellos objetos estáticos y, por otra, las entidades dinámicas que harían referencia a los diferentes seres vivos que tenemos en el juego. Son dos fases que es necesario diferenciar y tratar con independencia. Para este tipo de juegos, la solución idónea para el cálculo de colisión con el escenario sería la del *pixel perfect* a partir de las máscaras comentadas. Para los diferentes elementos dinámicos, sería mejor la técnica de objetos envolventes que veremos en los siguientes apartados.

Otra cosa que podemos comentar es sobre la máscara en sí. Fijémonos cómo estamos utilizando tan sólo una información binaria: ausencia (color negro) o presencia (color blanco) de cuerpo sólido. Esta información se puede ampliar mediante el uso de otros colores (por ejemplo, para indicar otras zonas de interés). Podría ser útil para definir un área mortal por ejemplo para los casos anteriores donde teníamos agua, o para definir las zonas accesibles de las entidades (camino, llanos, puentes, etc.), lo que en inglés se denota como el *walkability map*³, allá por donde podemos pasar. Las posibilidades que nos brinda esta técnica son muchísimas. En cierta manera, incluso, la edición de esta imagen (máscara) podría corresponderse al diseño de un nivel donde podremos ubicar la configuración o posición inicial de todos los elementos de nuestro juego. Este recurso se cargaría al iniciarse el juego, se parsearía y se crearían todas las estructuras de datos necesarias tal y como se indicase en él.

⁽³⁾Una posible traducción al castellano podría ser mapa de accesibilidad.

Otro aspecto a tener en cuenta es el tratamiento de estas máscaras. Como podemos apreciar, si debemos construir una imagen como el escenario, será necesario construir imágenes muy grandes y deberemos reservar una gran cantidad de memoria para poder almacenarlas y tratarlas.

Para solventar este problema, podemos hacer uso, por ejemplo, de un conocido sistema de codificación y compresión de datos llamado *run-length encoding* (RLE), utilizado también por el antiguo formato de imágenes PCX. Este sistema de codificación trata secuencias consecutivas de un mismo valor almacenando un único dato y su recuento. De esta manera, una imagen como la que se ilustra en la siguiente figura, de 8×8 píxeles y haciendo uso de la notación b (azul), g (verde), r (rojo) e y (amarillo) se almacenaría de esta forma: 4b, 4g, 4b, 4g, 32r, 16y.



Imagen de muestra para entender la ventaja de utilizar un sistema RLE para su almacenamiento.

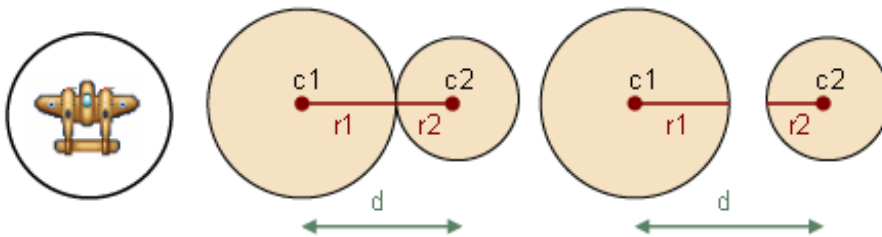
Si la imagen anterior se hubiera guardado texel a texel (que es un sistema que por ejemplo utiliza el formato Windows BMP), utilizando 8 bits para representar el color, hubieran sido necesarios $8 \times 8 \times 8 = 512$ bytes. Con el método RLE, han sido necesarios tan sólo 12 bytes. Como podemos apreciar, las ventajas son notorias y la diferencia será aún mucho mayor cuanto más grande sea nuestra imagen que hace de máscara física.

Una vez escogida esta técnica de almacenamiento, nos faltaría construir un tipo abstracto de datos (TAD) que nos ofreciera la posibilidad de obtener y actualizar nuestra información. Para poder acceder a ella, basta con convertir el índice (i, j) de la matriz a un índice de vector, siendo N el ancho de la matriz, mediante la fórmula $j*N + i$, para luego hacer un recorrido acumulando la magnitud hasta la posición deseada.

3.5.3. *Bounding circles*

Llamamos *bounding circle* a la circunferencia mínima que envuelve completamente un objeto. Ésta es una de las maneras más precisas y eficientes de representar el área de colisión de un objeto.

Veamos un ejemplo claro y sencillo de una *bounding circle* sobre un objeto nave y la estrategia a seguir para detectar la colisión:



Como podemos apreciar en la ilustración, la colisión tendrá lugar cuando la distancia entre los centros de los dos objetos sea igual o menor a la suma de los radios de cada uno. La estructura de la que debemos disponer de un objeto para llevar a cabo esta técnica deberá contar con los atributos posición, tamaño y radio, tal y como se muestra en el siguiente ejemplo:

```
struct object
{
    int x, y;
    int width;
    int height;
    int radius;
};
```

Dadas dos entidades, el algoritmo que detectará la colisión entre ambas, deberá efectuar los siguientes pasos:

- Calcular el centro de cada objeto
- Calcular la distancia entre ellos
- Compararla con la suma de los radios

El algoritmo resultante, pues, quedará como sigue:

```
bool collision( struct object *obj1, struct object *obj2 )
{
    int center_x1 = obj1->x + (obj1->width >> 1);
    int center_y1 = obj1->y + (obj1->height >> 1);
    int center_x2 = obj2->x + (obj2->width >> 1);
    int center_y2 = obj2->y + (obj2->height >> 1);

    int delta = distance( center_x1, center_y1, center_x2, center_y2);
    int sum_radius = obj1->radius + obj2->radius;

    if( delta <= sum_radius )    return true;
    else                          return false;
}

int distance(int x1, int y1, int x2, int y2)
{
    return sqrt( (x2-x1)*(x2-x1) + (y2-y1)*(y2-y1) );
}
```

Este código es bastante sencillo. Sin embargo, la función raíz cuadrada tiene un coste computacional considerable. Los videojuegos, en general, suelen aplicar diferentes técnicas de optimización. Para el caso de esta función, podemos utilizar la aproximación de McLaurin, que, con una precisión del 97%, nos brinda un código mucho más eficiente. Es el que indicamos a continuación:

```
int distance(int x1, int y1, int x2, int y2)
{
    int x = abs( x2 - x1 );
    int y = abs( y2 - y1 );
    int min = x < y? x : y;
    return ( x+y - (min>>1) - (min>>2) + (min>>4) );
}
```

3.5.4. *Bounding boxes*

Es una técnica que utiliza la misma filosofía que la *bounding circles*, pero, en este caso, el área mínima envolvente es un rectángulo.

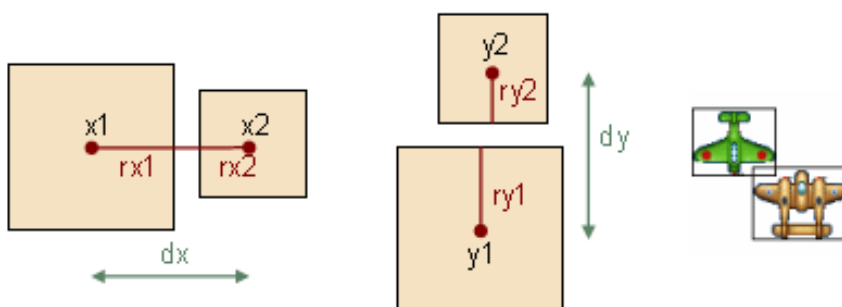
En muchos casos puede resultar práctico utilizar círculos como área de colisión. Sin embargo, en otras ocasiones puede no resultar adecuado. Imaginemos, por ejemplo, que estamos desarrollando un juego en el cual tuvieran que interactuar humanoides, tal y como presenta la siguiente ilustración.



Podemos apreciar claramente cómo, en este contexto, la precisión de las circunferencias envolventes no resultan muy exactas y, si no actuamos de otro modo, tendremos colisiones entre estos personajes cuando la distancia entre ellos sea considerable. Definitivamente, el papel de la detección de colisiones vía rectángulos, ofrecerá un test mucho más preciso.

La base para la detección de colisiones mediante rectángulos es muy similar a la de las circunferencias. En este caso, sin embargo, deberemos abordar la comprobación entre las distancias tanto para el eje X como para el eje Y. Haciendo un símil con el algoritmo anterior, podríamos decir que es necesario comprobar la distancia mediante los radios de los dos ejes.

Por tanto, la estrategia a seguir será la que podemos apreciar en la siguiente representación gráfica:



Rediseñamos la estructura presentada previamente para los objetos y describimos el nuevo algoritmo de test de colisión.

```
struct object
{
    int x, y;
    int width;
    int height;
    int radius_x;
    int radius_y;
}
```



```
};
```

Nuestra función de colisión quedaría de la manera siguiente:

```
bool collision( struct object *obj1, struct object *obj2 )
{
    int center_x1 = obj1->x + (obj1->width >> 1);
    int center_y1 = obj1->y + (obj1->height >> 1);
    int center_x2 = obj2->x + (obj2->width >> 1);
    int center_y2 = obj2->y + (obj2->height >> 1);

    int delta_x = abs( center_x1 - center_x2 );
    int delta_y = abs( center_y1 - center_y2 );

    int sum_x = obj1->radius_x + obj2->radius_x;
    int sum_y = obj1->radius_y + obj2->radius_y;

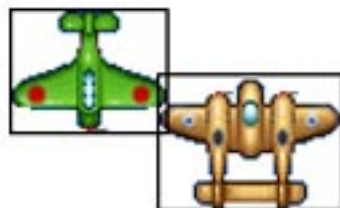
    if( delta_x <= sum_x && delta_y <= sum_y )
        return true;
    else
        return false;
}
```

3.5.5. Jerarquías de objetos envolventes

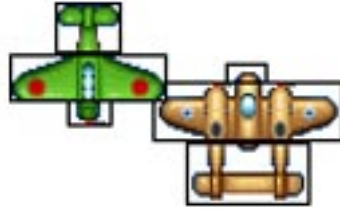
Las dos técnicas que acabamos de presentar, *bounding circles* y *bounding boxes*, resultan suficientemente efectivas y su coste de computación es óptimo. Aun así, podemos conseguir un mayor grado de precisión sin añadir complejidad al algoritmo haciendo uso de estructuras jerárquicas envolventes.

El procedimiento es idéntico al mostrado con anterioridad. La única diferencia es que ahora dispondremos de tests que se realizarán con niveles de detalle diferente, de menor a mayor precisión, profundizando en otro nivel según el test anterior responda afirmativamente.

Mostremos un ejemplo práctico de dos niveles de *bounding boxes*:



Primer nivel



Segundo nivel

El algoritmo funcionaría de la siguiente manera: en primer lugar, realizamos el test haciendo uso del primer nivel de *bounding boxes*; si la respuesta es afirmativa, hacemos el test de cada uno de los rectángulos envolventes del siguiente nivel de una entidad con los de la otra; si alguna de estas consultas responde afirmativamente, daremos por hecho que se ha producido la colisión.

4. Motores de física

A la hora de hacer un juego 2D, el aspecto de la física es un concepto muy importante. Es algo que facilita mucho la inmersión en el juego y de por sí es divertido. Acciones básicas como saltar o un doble salto, la reacción al producirse la colisión entre dos entidades y la aceleración o desaceleración producida cuando una entidad corre o frena es algo muy agradable en términos de *jugabilidad*. Una mala implementación o una solución abrupta o seca de estas acciones puede causar una imagen muy negativa cuando justo estamos empezando a probar un juego o, al contrario, puede invitarnos a continuar con él.

Quizás cuando estamos desarrollando nuestro primer videojuego es lo primero que dejamos como algo mejorable para más adelante, pero sin lugar a dudas, la física es un elemento imprescindible que definirá el éxito de nuestro juego a partir de su *jugabilidad*. Tan importante es este hecho que incluso juegos como los etiquetados *casual games* (juegos de poca complejidad y reglas sencillas con los que se puede jugar en muy poco tiempo) pueden ser desarrollados centrándose en la física, dando menor importancia a otros aspectos, como los gráficos y el propio decorado, y convertirse en auténticos éxitos a nivel mundial.

Ejemplos

Sirva como ejemplo de este hecho el conocido "Angry Birds" de Mikael Hed, que en el año 2010 ya contaba con 12 millones de descargas.



"Angry Birds", juego de físicas 2D con un argumento muy sencillo que ha sido todo un éxito a nivel mundial

Otro juego sencillo que hace uso de las físicas para crear un divertido juego de habilidad mental puede ser el juego flash "Cover Orange 2" de Johnny-K. A continuación presentamos una imagen de este juego:



"Cover Orange 2", otro divertido juego de físicas 2D en el cual hay que resolver diferentes rompecabezas

Teniendo en cuenta esto, y una vez vistos los elementos básicos de física, es hora de introducir el concepto de *motor de física*. Para no reinventar la rueda y precisamente para hacer uso del concepto de ingeniería del software, existen varias soluciones que nos brindan la simulación de física para entornos 2D. Para según qué contextos, el cálculo no se diferencia mucho de lo que sería un motor de físicas en 3D (las leyes son las mismas), pero sin embargo, para otras situaciones, como la división espacial, las diferencias son abismales y para ello contamos con tecnologías simplificadas que dan respuesta únicamente al contexto 2D.

El motor Box2D se ha convertido en uno de los motores de física más importantes en la actualidad para hacer juegos 2D y por eso lo estudiaremos con detenimiento más adelante. Por último, dentro ya del mercado, otro motor que podríamos destacar es Torque 2D y será la solución de pago que comentaremos. Pasemos a continuación a mostrar una breve descripción de cada uno de ellos.

a) Box2D

Box2D es un motor de físicas 2D de código abierto escrito, en C++ por Erin Catto y publicado bajo licencia Zlib. Ha sido utilizado en juegos como los anteriormente mencionados "Angry Birds" y "Cover Orange 2", u otros como "Crayon Physics Deluxe", "Rolando", "Fantastic Contraption" o "Incredibots

Ejemplo

Entre los diversos motores de física 2D existentes, podrían citarse los ejemplos siguientes: Box2D, Chipmunk, Farseer y Physics2D.Net.



Box2D

Transformice", así como para otros muchos juegos desarrollados en Flash y para iPhone, iPad y Android. Empezó a desarrollarse en el año 2006, y en marzo del 2008 alcanzó su versión 2.0.

Os recomendamos probar cada uno de estos juegos que acabamos de mencionar. Son de reconocido éxito y agrado de masas y seguro que podrá influirlos positivamente cuando llevéis a cabo un proyecto propio. Una primera toma de contacto mediante una búsqueda de estos títulos, acompañándolos de la palabra *gameplay* en *Youtube*, por ejemplo, puede ser suficiente.

b) Chipmunk

Chipmunk es un motor físicas 2D portable de código abierto, escrito en C por Scott Lembcke y publicado bajo licencia MIT. Concretamente, se trata de un motor físico en tiempo real de cuerpos rígidos (*rigid bodies*) diseñado para ser rápido, poco pesado y fácil de utilizar. En agosto del 2010, fue lanzada la versión 5.3.



c) Farseer

Farseer es un motor de físicas de código abierto para la plataforma Microsoft .Net. Existen versiones para Silverlight, así como para XNA. Fue creado por Jeff Weber y actualmente mantenido por Ian Qvist. En abril del 2011 fue publicada su versión 3.3.1.



d) Physics2D.Net

Physics 2D.Net es un motor de físicas 2D para cuerpos rígidos escrito en C# para Silverlight, XNA y Mono. Fue creado por Jonathan Porter, y en junio del 2008 fue publicada la versión 2.0.



Physics 2D.Net

e) Torque 2D

Torque 2D es mucho más que un motor de físicas. Se trata de un completo motor de videojuegos 2D escrito en C++ que incluye muchas funcionalidades, como veremos más adelante, desarrollado por GarageGames en el año 2009 y disponible a un precio realmente asequible.

Comentamos a continuación lo que podríamos considerar como las dos muestras más representativas en la actualidad referente a motores de físicas 2D: Box2D y Torque 2D Physics.



4.1. Box2D

Box2D es un motor de videojuegos orientado a la simulación de física realista en sistemas de dos dimensiones. Es de código abierto, distribuyéndose bajo la licencia Zlib, que es compatible con la licencia GNU GPL y por lo tanto,

puede considerarse software libre. Está escrito en C++ de forma independiente a la plataforma y puede usarse, por tanto, en cualquier sistema que disponga de compilador de C++. A día de hoy, se ha utilizado en juegos para Nintendo DS, Wii, diversos teléfonos móviles y para la mayoría de sistemas operativos. Asimismo, el motor ha sido portado completamente a otros lenguajes de programación como Java, Flash, C#, JavaScript y D y linkado a otros como Python o DarkBasic.

Su autor es Erin Catto, programador de los motores físicos de la saga "Tomb Raider" desde el título "Tomb Raider: Legend" y del motor físico del juego de Blizzard "Diablo III", que se ha reutilizado en otros títulos posteriores de la misma compañía.

Inicialmente, Box2D fue presentado como una demo en el Game Developers Conference (GDC) del 2006. El 11 de septiembre del 2007 fue publicado como código abierto y alojado en SourceForge. El 17 de enero del 2010 el proyecto se trasladó a Google Code, donde permanece hasta ahora. La versión 2.0 fue lanzada el 6 de marzo del 2008, renovando la API e incluyendo otras novedades como la detección de colisiones continua.

4.1.1. Conceptos

El núcleo del motor trabaja con los siguientes objetos fundamentales:

- **Shape** (forma). Permite definir objetos geométricos en 2D, como podría ser un círculo o cualquier polígono.
- **Rigid body** (cuerpo rígido). Permite definir un trozo de materia, un objeto que es tan rígido, que la distancia entre dos partes de materia se considera constante.
- **Fixture**. Asigna una forma a un cuerpo, y propiedades tales como la densidad, la fricción o la restitución.
- **Constraint** (restricción de integridad). Permite definir una conexión física que restringe los grados de libertad de las formas.
- **Contact constraint**. Este objeto es un *constraint* especialmente diseñado para prevenir la penetración de cuerpos rígidos, simulando así fricción y restitución. Este tipo de objeto es creado de forma automática por el motor.
- **Joint** (unión). Es un *constraint* que nos permite tener la unión entre dos o más cuerpos. El motor Box2D soporta varios tipos de uniones como revoluciones, prismáticas, a distancia y algunos otros más.

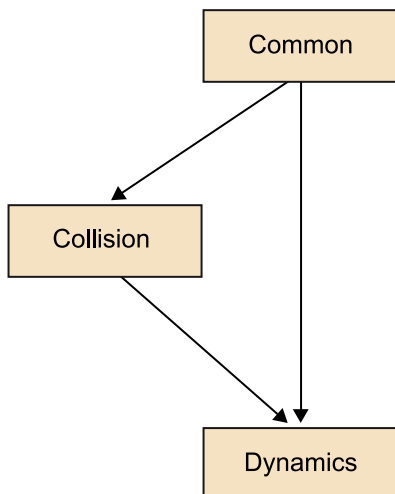
- **Joint limit.** Permite restringir el rango de movimiento de dos o más cuerpos unidos.
- **Joint motor.** Permite controlar el movimiento de conexión de cuerpos de acuerdo a los grados de libertad de las *joint*.
- **World.** Colección de cuerpos, *fixtures* y *constraints* que interactúan entre ellos. Box2D soporta la creación de múltiples mundos, aunque esto no es habitual o deseable.

4.1.2. Arquitectura y diseño

La estructura de Box2D está compuesta por tres módulos:

- a) **Common.** Se encarga de la gestión de memoria, las operaciones matemáticas y la configuración.
- b) **Collision.** Define formas, contiene el algoritmo de fase ancha o *broad-phase*, que calcula colisiones a partir de *bounding boxes* para mejorar la eficiencia y define funciones de colisiones.
- c) **Dynamics.** Proporciona el entorno de simulación, los cuerpos, los *fixtures* o unión de cuerpo y forma, y las articulaciones.

Módulos principales del motor Box2D



La detección de colisiones de Box2D y su manera de resolver los sistemas de ecuaciones que describen el movimiento de los cuerpos se basa en tres algoritmos principales:

- a) uno de **barrido y poda** (*sweep and prune*) incremental,
- b) otro de **detección de colisiones continua** y

c) un tercero de **solución de contactos** que tiene un coste lineal.

Estos algoritmos permiten la simulación eficiente de cuerpos veloces y de grandes pilas de elementos sin que se pierda ninguna colisión ni se produzcan situaciones inestables.

Por el sistema de representación numérica empleado, Box2D está optimizado para trabajar con objetos dinámicos de entre 0,1 y 10 metros y con objetos estáticos de hasta 50 metros. Esto se debe tener muy en cuenta a la hora de definir los cuerpos que habitan en nuestro mundo físico.

La gestión de la memoria tiene un papel importante en el diseño de la API Box2D. Toda instanciación de un objeto debe realizarse mediante llamadas a la factoría que proporciona la API y de ninguna otra forma. Concretamente, tenemos funciones creadoras como por ejemplo:

Ejemplo

```
b2Body* b2World::CreateBody(const b2BodyDef* def)
```

```
b2Joint* b2World::CreateJoint(const b2JointDef* def)
```

Y éstas son sus correspondientes funciones destructoras:

```
void b2World::DestroyBody(b2Body* body)
```

```
void b2World::DestroyJoint(b2Joint* joint)
```

Cuando creamos un objeto o una unión, debemos proporcionarle una definición. Esta definición contendrá toda su información necesaria para ser construido. Veamos aquí un ejemplo:

Ejemplo

```
b2Fixture* b2Body::CreateFixture(const b2FixtureDef* def)
```

```
void b2Body::DestroyFixture(b2Fixture* fixture)
```

Box2D permite simular el movimiento de los sólidos rígidos sujetos a restricciones. Estas restricciones normalmente vienen dadas por algún tipo de sistema articulado del que el sólido forma parte. El motor puede simular cuerpos formados por polígonos convexos, círculos y formas estrelladas. Estos cuerpos pueden estar unidos entre sí mediante articulaciones y pueden sufrir el efecto de fuerzas cualesquiera, entre las que cabe destacar la gravedad, el rozamiento o el coeficiente de restitución, que indica cuánto rebota un objeto tras un impacto.

Box2D incluye también una completa API y un banco de pruebas en el que testar nuestros códigos sin necesidad de implementar una interfaz gráfica.

4.1.3. Funcionalidades

Pasaremos a especificar las funcionalidades del motor, divididas en los módulos que lo componen.

Módulo Common

En primer lugar explicaremos las funcionalidades del módulo Common, este módulo contiene las configuraciones, la gestión de memoria y un paquete matemático.

a) Configuraciones

La cabecera `b2Settings.h` contiene:

- Tipos de datos como `int32`, `float32`, `int8`, etc., para hacer más fácil determinar el tamaño de las estructuras.
- **Constantes predefinidas** que normalmente no son necesarias reajustar. Box2D utiliza coma flotante para el sistema de colisiones y la simulación.
- **Asignación de memoria.** El fichero de configuración define los llamados `b2Alloc` y `b2Free` para grandes peticiones de memoria. Estas llamadas pueden ser enviadas a nuestro propio sistema de gestión de memoria.
- **El número de versión.** La estructura `b2Version` contiene la versión actual y la podemos solicitar en tiempo de ejecución.
- **Funciones de fricción y restitución.** Estas serán almacenadas en el fichero de configuración únicamente en caso de que queramos personalizarlas.

b) Gestión de memoria

Muchas decisiones que ciñen el diseño de Box2D están orientadas a la necesidad del cómputo rápido y un uso eficiente de la memoria. Es por ello por lo que debemos delegar en Box2D todo el tema de gestión y asignación de memoria.

Box2D suele tener en memoria un gran número de objetos pequeños, de entre 50 y 300 bytes. El uso que podamos hacer personalmente de las funciones del sistema `new` o `malloc` para la gestión de objetos pequeños es altamente ineficiente y puede ocasionar fragmentación.

La solución que ofrece Box2D para este problema es utilizar un *small object allocator* (SOA), el `b2BlockAllocator`. El SOA mantiene una serie de contenedores ampliables a diferentes tamaños. Cuando se hace una solicitud de me-

moria, el SOA devuelve el bloque de memoria que mejor se adapte al tamaño solicitado. Cuando un bloque se libera, se devuelve al contenedor. Ambas operaciones son rápidas.

Dado que Box2D utiliza el SOA, nosotros no debemos hacer *new* o *malloc* de un cuerpo, *fixture* o *joint*. Debemos cargar un objeto *b2World* y, a partir de esta clase, utilizar los métodos pertinentes para la creación de las entidades que acabamos de citar. Así es como Box2D nos facilita el hecho de no tener que preocuparnos por la gestión de la memoria.

Para la ejecución de un paso (*tick*) de la simulación, Box2D necesita un pequeño espacio de memoria temporal. Con este fin utiliza un objeto llamado *b2StackAllocator*. Para evitar peticiones de memoria innecesarias, no debemos interactuar tampoco con esta pila.

c) Paquete Math

Box2D incluye un módulo para trabajar con vectores y matrices. Este ha sido diseñado para satisfacer las necesidades internas de Box2D. Todos los métodos del módulo están expuestos de forma pública, con el fin de que puedan ser usados desde la aplicación.

Módulo Collision

Ahora pasaremos a explicar las funcionalidades del módulo Collision. Este módulo contiene las formas (shapes) y las funciones para poder operar entre ellas. Además contiene un árbol dinámico y el paso llamado *broad-phase* para la aceleración del procesado de colisiones de grandes sistemas. Veamos estos tres puntos por separado.

a) Shapes

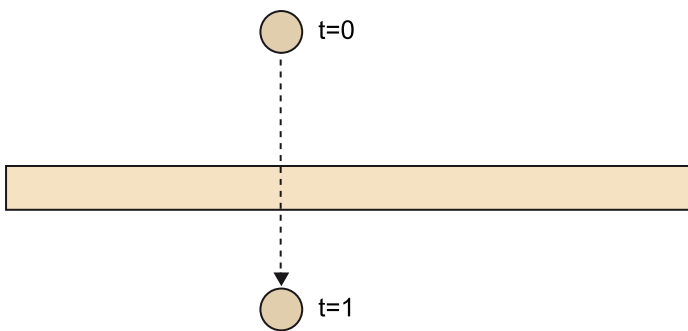
Box2D tiene la clase base *b2Shape*, que define las siguientes funciones:

- Test punto interior a una forma.
- Intersección recta-forma.
- Calcular la caja envolvente alineada a los ejes⁴ de una forma.
- Calcular las propiedades de masa de una forma.
- Operaciones con circunferencias y polígonos.

⁽⁴⁾En inglés, *axis aligned bounding box* (AABB).

El módulo Collision contiene algunas funciones bilaterales que computa algunos resultados a partir de dos formas, como son:

- Contact manifolds. Nos proporciona los puntos de contacto entre dos formas superpuestas.
- Distancia entre dos formas. Esta función necesita que las formas se conviertan a un objeto `b2DistanceProxy`.
- Tiempo de impacto. El objeto `b2TimeOfImpact` se usa para determinar el momento en que dos formas colisionarán. El uso principal de esta función es evitar el efecto tunneling. Dependiendo de las características de las formas, sus velocidades y trayectorias, puede pasar que dos objetos no colisionen cuando sí debieran hacerlo. Este hecho puede observarse en la siguiente ilustración:



El objeto circular traspasa el objeto rectangular, pero sin embargo en los dos instantes de tiempo el algoritmo de colisión no da el resultado esperado. En cuanto al efecto tunneling, debería producirse colisión, pues el objeto rectangular está en la trayectoria del objeto circular.

b) Árbol dinámico

La clase `b2DynamicTree` es usada por el motor `Box2D` para organizar grandes cantidades de *shapes* de forma eficiente. El árbol dinámico es un árbol jerárquico de AABB, donde cada nodo puede tener un máximo de dos hijos. Esta estructura permite hacer tests de intersección mediante rayos de forma eficiente.

c) Broad-phase

Todo cómputo físico suele realizarse en dos pasos, uno de mayor granularidad y otro de menor granularidad. En inglés, se suele referenciar a estos dos pasos como *broad-phase* y *narrow-phase*. El paso de mayor granularidad es muy importante pues evita el proceso de cálculo con todos los elementos del escenario, evita cálculos innecesarios. Hace posible que no tengamos que calcular el test entre todas las *shapes*. Mediante un primer filtro en principio no muy costoso, descartamos un gran número de entidades y, con el conjunto de objetos restantes, ahora sí, se prosigue con el cálculo físico de mayor detalle y más costoso.

El objeto `b2BroadPhase` es el encargado de reducir la carga de trabajo del cálculo mediante el uso de un árbol dinámico.

Módulo Dynamics

Por último, definiremos las funcionalidades que nos ofrece el módulo Dynamics. Este módulo es la parte más compleja del motor y es la parte con la que más interactuamos. Concretamente contiene los siguientes módulos:

a) *Shape Fixture Class*

Nos permite fijar las formas de los cuerpos y contiene los atributos siguientes:

- ***Single shape***. Para la creación y almacenamiento de datos de la forma
- ***Density, friction and restitution***
 - *Density*: la densidad es usada para computar las propiedades de masa del nodo padre.
 - *Friction*: la fricción es usada para hacer que los objetos se desplacen a lo largo de los otros de forma realista.
 - *Restitution*: la restitución es usada para hacer que los objetos reboten.
- ***Collision filtering flags***. Este es un sistema para prevenir la colisión entre figuras. Box2D soporta un total de 16 categorías diferentes de colisión.
- ***Back pointer***. Puntero al cuerpo del nodo superior (padre).
- ***User data***.
- ***Sensor flag***. Usado para permitir a la lógica del juego saber que dos figuras se superponen pero no debe haber respuesta.

b) *Rigid Body Class*

Los cuerpos tienen una posición y una velocidad, podemos aplicar fuerzas, torsiones e impulsos. Existen tres tipos de cuerpos:

- ***Static***. Los cuerpos estáticos no pueden moverse, haciéndonos creer que su masa es infinita. No pueden colisionar con otros *static bodies* ni con *kinematic bodies*.
- ***Kinematic***. Estos cuerpos se mueven bajo una simulación de acuerdo a su velocidad y no responden a fuerzas. Éstos no puede colisionar con *static* u otros *kinematic bodies*.

- **Dynamic.** Se trata de cuerpos completamente simulables, puede aplicársele fuerzas y pueden colisionar con todos los otros cuerpos.

Un cuerpo (*body*) se define mediante los siguientes atributos:

- **Body type.** Hay que escoger entre *static*, *kinematic* y *dynamic*.
- **Position y angle.** La posición nos permite definir el punto original del cuerpo y el ángulo nos permite definir el ángulo del polígono.
- **Damping** (amortiguación). Este parámetro es usado para reducir la velocidad del cuerpo.
- **Sleep parameters.** Parámetro usado para definir que el cuerpo se encuentra en reposo cuando sus cambios de estado son mínimos y pueda ser despertado por ejemplo cuando un objeto impacta con él.
- **Fixed rotation.** Nos permite definir una inercia rotacional sobre el cuerpo.
- **Bullets.** Son usados para implementar el CCD (*continuous collision detection*) con *static* y *dynamic bodies*.
- **Activation.** Nos permite definir que un cuerpo que ha sido creado no participe en la colisión.
- **User data.** *Void pointer* para ser enlazado a un objeto de la aplicación del usuario.

Para crear o destruir cuerpos, debemos usar la *body factory*, proporcionada por la clase *world*. Esta nos permite crear cuerpos dinámicamente de forma eficiente y añadirlos a la estructura de datos del mundo. Una vez creados, pueden ser usados en tiempo de ejecución y podemos modificar o consultar algunos parámetros como son su masa, estado, posición y velocidad.

c) **Contact Class**

Se trata de objetos usados en el motor Box2D para manejar las colisiones. Esta clase nos permite crear y destruir objetos. Para poder acceder a los contactos disponemos de vías diferentes: directamente desde el mundo o desde la estructura de los cuerpos.

Podemos recibir datos de la colisión a partir del objeto *b2ContactListener*, que nos informará de los contactos que hayan sucedido. Podemos filtrar aquellos eventos que no nos interesen que se den o que tengamos mayor interés por usar, mediante el objeto *b2ContactFiltering*.

d) *Joint Classes*

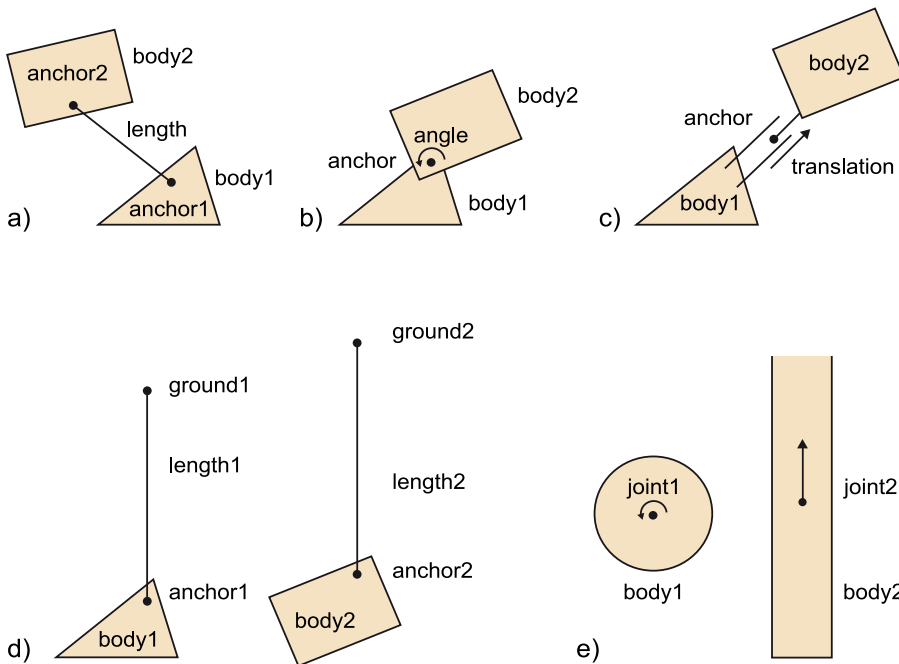
Los objetos de esta clase son usados para definir restricciones a los cuerpos, ya sean respecto al mundo o respecto a otros cuerpos. Para definir *joints*, debemos tener en cuenta los siguientes puntos:

- Todas las *joints* deben estar conectadas entre dos cuerpos, siendo uno de estos de tipo estático. Las *joints* entre *static* y *kinematic* a pesar de estar permitidas, no tienen efecto.
- Podemos definir *flags* para prevenir las colisiones entre nuestros cuerpos con los demás.
- La mayoría de definiciones de *joints* requieren que se proporcionen algunos datos geométricos.

De igual forma que con los cuerpos, para crear estas uniones debemos usar el *joint factory*, que se encargará de crearlos usando los métodos de la clase mundo.

Tenemos diferentes tipos de *joints*. He aquí una pequeña descripción de cada una de ellas:

- ***Distance joint***: indica que la distancia entre dos puntos en dos cuerpos debe ser constante.
- ***Revolute joint***: fuerza a que dos cuerpos compartan un punto de anclaje.
- ***Prismatic joint***: con esta joint impedimos la rotación relativa entre dos cuerpos y permitimos su traslación.
- ***Pulley joint***: usada para crear una polea, conecta dos cuerpos a tierra y entre sí de manera que, si uno sube, el otro baja.
- ***Gear joint***: se usa para la creación de artilugios mecánicos sofisticados.
- ***Mouse joint***: usada para manipular cuerpos con nuestro ratón, conduciéndolos a partir del cursor.
- ***Line joint***: se usa para modelar la rueda de un vehículo con suspensión.
- ***Weld joint***: usada para tratar de limitar todo el movimiento relativo entre dos cuerpos.

Tipos de *joint*a) *Distance joint*; b) *revolute joint*; c) *prismatic joint*; d) *pulley joint*; e) *gear joint*.e) **World Class**

Esta clase contiene los cuerpos y los tipos de uniones anteriormente mencionados. Esta clase es la encargada de manejar todos los aspectos de la simulación y permite realizar consultas como AABB y *ray-casts*. La mayoría de nuestra interacción como programador se realizará a partir de esta clase.

Para crear el objeto mundo, basta con indicar un vector de gravedades y un booleano indicando si los objetos pueden entrar o no en el estado *sleep*. He aquí un ejemplo de ello:

Ejemplo

```
b2World* myWorld = new b2World(gravity, doSleep);
```

```
... nuestra pruebas ...
```

```
delete myWorld;
```

A partir de esta clase es como nosotros podemos llevar a cabo nuestra simulación. Para ello, es necesario especificar una magnitud de tiempo (*time step*), y un contador de iteraciones de velocidad y de posición. Un ejemplo de ello sería el que sigue a continuación:

Ejemplo

```
float32 timeStep = 1.0f / 60.f;
```

```
int32 velocityIterations = 10;
```

```
int32 positionIterations = 8;
```

```
myWorld->Step(timeStep, velocityIterations, positionIterations);
```

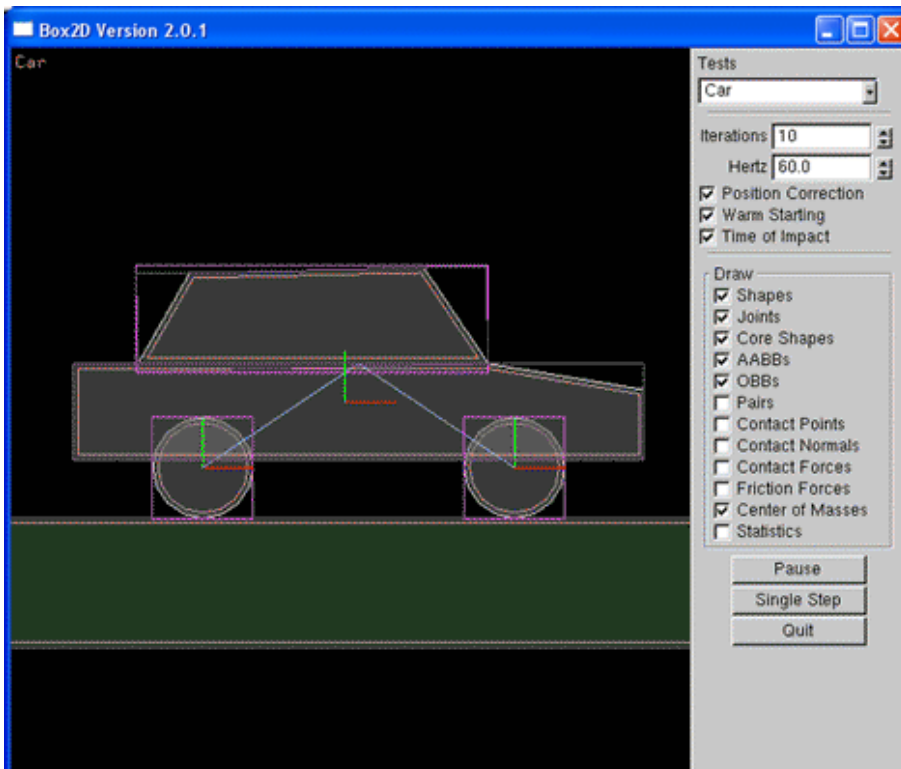
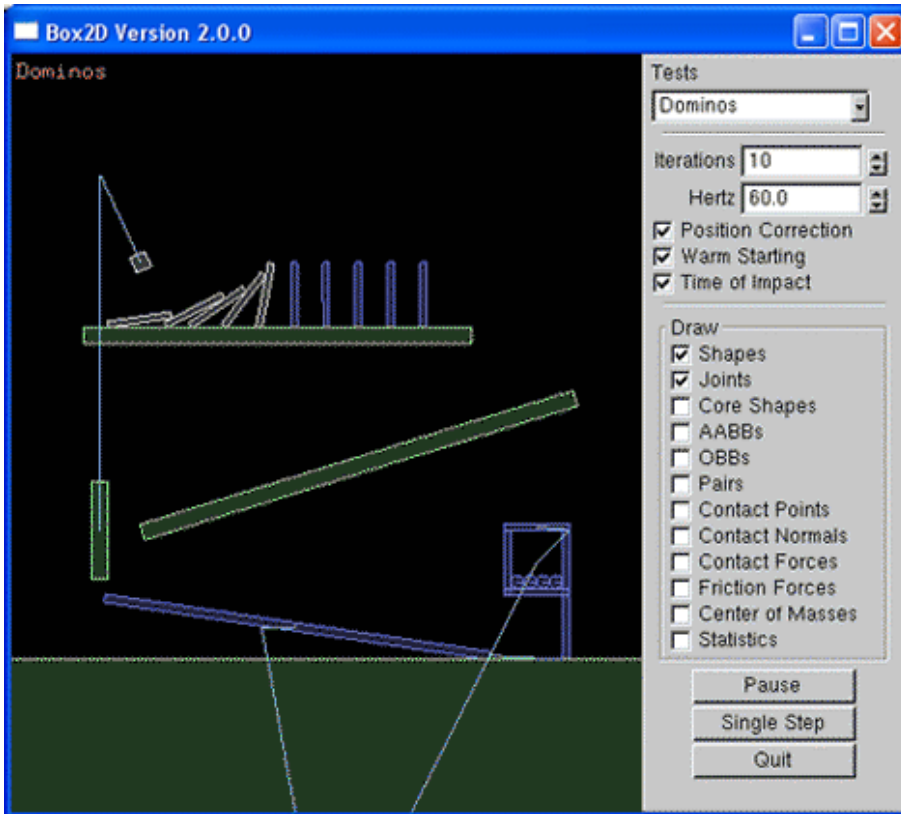
Además de realizar la simulación, es posible:

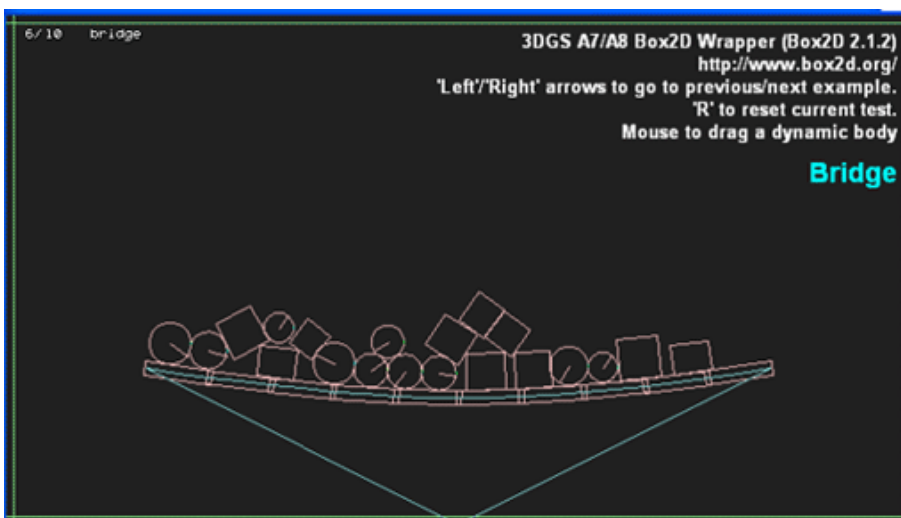
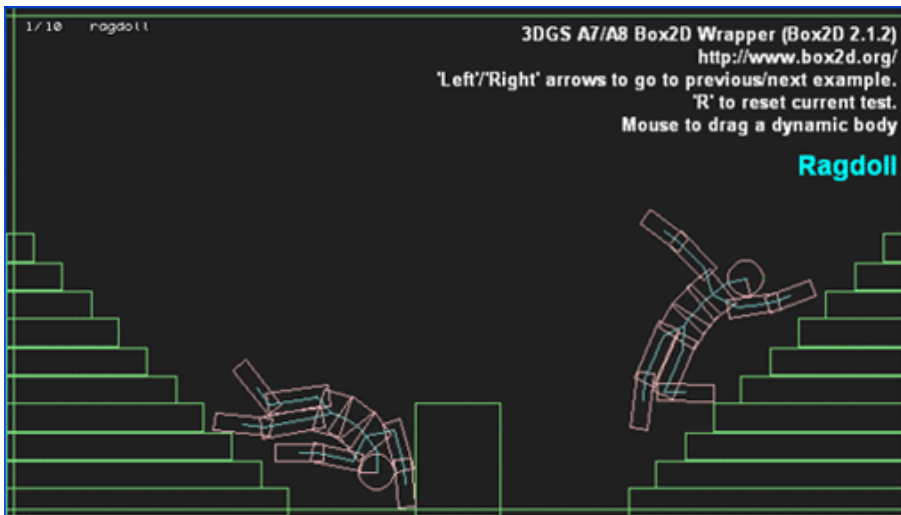
- Explorar el estado del sistema, es decir, recorrer la lista de cuerpos, contactos y uniones e iterar sobre éstos.
- Consultas AABB, para determinar qué formas (*shapes*) se encuentran en un área determinada.
- Hacer *tests* de *ray-casts* (por ejemplo, para implementar disparos o proyectiles).
- Aplicar fuerzas, torsiones e impulsos a los cuerpos.
- Realizar transformaciones de coordenadas locales y de mundo para puntos y vectores.
- Iterar sobre las *fixtures* de cada uno de los cuerpos.

4.1.4. Visualización y testeo

Este motor proporciona una clase llamada `b2DebugDraw`, que permite obtener con toda clase de detalle una visualización de nuestro mundo físico. Las entidades que se pueden ilustrar son: contornos de formas, conexiones de uniones, núcleo de las formas (para colisiones continuas), AABB, OBB, contactos de la fase *broad-phase* y los centros de masa.

En las siguientes imágenes mostramos la visualización del mundo físico de cuatro juegos de prueba diferentes.





Box2D permite testear y visualizar en tiempo real el estado actual de nuestro mundo físico.

4.1.5. Un caso práctico

Veamos a continuación un ejemplo escrito en ActionScript (para Flash) y que utiliza el port de Box2D a dicho lenguaje de programación. Se trata de una demostración de colisión entre cajas tal y como puede verse en la figura que tenemos más adelante.

```
package {
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.utils.Timer;
    import flash.events.TimerEvent;

    //Librerías de Box2D que utilizaremos
    import Box2D.Dynamics.*;
    import Box2D.Collision.*;
    import Box2D.Collision.Shapes.*;
    import Box2D.Common.Math.*;
}
```

```
public class demo extends Sprite {
    /*La clase b2world es el objeto más importante a tener cuenta
    cuando se trabaja con Box2D. Guarda todos los cuerpos y uniones
    entre cuerpos, tiene los listeners y es responsable del
    desarrollo de la simulación:*/
    public var the_world:b2World;
    var time_count:Timer=new Timer(1000);

    public function demo() {
        /*La clase b2AABB define el marco donde las reglas físicas
        de Box2D son aplicables. En este caso, un cuadrado de
        100x100 metros (1 metro = 30 píxeles).*/
        var environment:b2AABB = new b2AABB();
        environment.lowerBound.Set(-100.0, -100.0);
        environment.upperBound.Set(100.0, 100.0);

        /*La variable gravity es un vector que representa la
        fuerza de la gravedad en la superficie terrestre.*/
        var gravity:b2Vec2=new b2Vec2(0.0,10.0);

        /*Inicializamos nuestro mundo Box2D. "Environment" indica
        las dimensiones, "gravity" la fuerza de la gravedad y el
        booleano indica que no se deben simular los cuerpos en
        reposo, lo cual mejora la eficiencia.*/
        the_world=new b2World(environment,gravity,true);

        /*El uso de la clase b2DebugDraw permite obtener
        información detallada de la física de los objetos que
        pueblan el mundo: centros de masa, conectividad entre las
        diferentes partes de un cuerpo, bounding boxes, posibles
        contactos, etc. En este ejemplo lo usamos para mostrar el
        atributo shape de todo cuerpo.*/
        var debug_draw:b2DebugDraw = new b2DebugDraw();
        var debug_sprite:Sprite = new Sprite();
        addChild(debug_sprite);
        debug_draw.m_sprite=debug_sprite;
        debug_draw.m_drawScale=30;
        debug_draw.m_fillAlpha=0.5;
        debug_draw.m_lineThickness=1;
        debug_draw.m_drawFlags=b2DebugDraw.e_shapeBit;
        the_world.SetDebugDraw(debug_draw);

        /*Crea un cuerpo y lo añade al "world".
        b2BodyDef es la definición del cuerpo, a partir de la
        cual luego se crea el cuerpo en cuestión con la clase
        b2Body. b2PolygonDef es la definición de un polígono. De la
        definición del cuerpo sólo configuramos su posición,
```

```

    mientras que de la del polígono configuramos sus dimensiones y
    otros atributos físicos, como el coeficiente de fricción o
    la densidad. Por último, ambas definiciones se añaden al
    cuerpo final y se infiere la masa.*/
    var final_body:b2Body;
    var the_body:b2BodyDef;
    var the_box:b2PolygonDef;
    the_body = new b2BodyDef();
    the_body.position.Set(8.5, 13);
    the_box = new b2PolygonDef();
    the_box.SetAsBox(8.5, 0.5);
    the_box.friction=0.3;
    the_box.density=0;
    final_body=the_world.CreateBody(the_body);
    final_body.CreateShape(the_box);
    final_body.SetMassFromShapes();
    addEventListener(Event.ENTER_FRAME, on_enter_frame);
    time_count.addEventListener(TimerEvent.TIMER, on_time);
    time_count.start();
}

public function on_time(e:Event) {
    /*Creación de cajas de tamaño y posición aleatorios cada
    cierto tiempo. El proceso de definición y creación de los
    cuerpos es, en este caso, idéntico al descrito más arriba.*/
    var final_body:b2Body;
    var the_body:b2BodyDef;
    var the_box:b2PolygonDef;
    the_body = new b2BodyDef();
    the_body.position.Set(Math.random()*10+2, 0);
    the_box = new b2PolygonDef();
    the_box.SetAsBox(Math.random()+0.1,Math.random()+0.1);
    the_box.friction=0.3;
    the_box.density=1;
    final_body=the_world.CreateBody(the_body);
    final_body.CreateShape(the_box);
    final_body.SetMassFromShapes();
}

public function on_enter_frame(e:Event) {
    the_world.Step(1/30, 10);
}
}
}

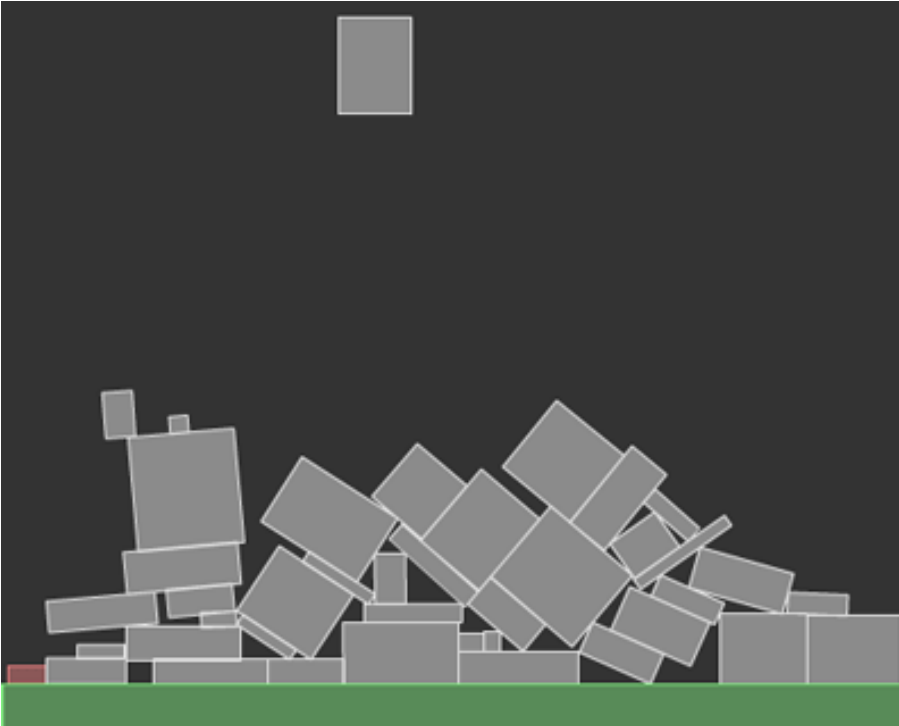
```

En este ejemplo, cada cierto tiempo, aparece en la parte superior del *world* (nuestro mundo) un cuerpo de forma rectangular y dimensiones variables. Este cuerpo se ve afectado por la fuerza de gravedad que hemos definido al crear

el mundo, y cuando choca con la montaña de bloques caídos anteriormente se ve afectado por la fricción de los cuerpos que hemos definido. El efecto del choque ocurre de forma transparente al programador: es inherente al mundo que hemos creado.

En este ejemplo podemos ver, por tanto, cómo definiendo un marco (el mundo físico) y unos cuerpos que en él habitan, es suficiente, el motor Box2D se encarga de toda la simulación. Veamos una captura de la película que se genera:

Ejemplo de aplicación haciendo uso del motor de físicas Box2D.

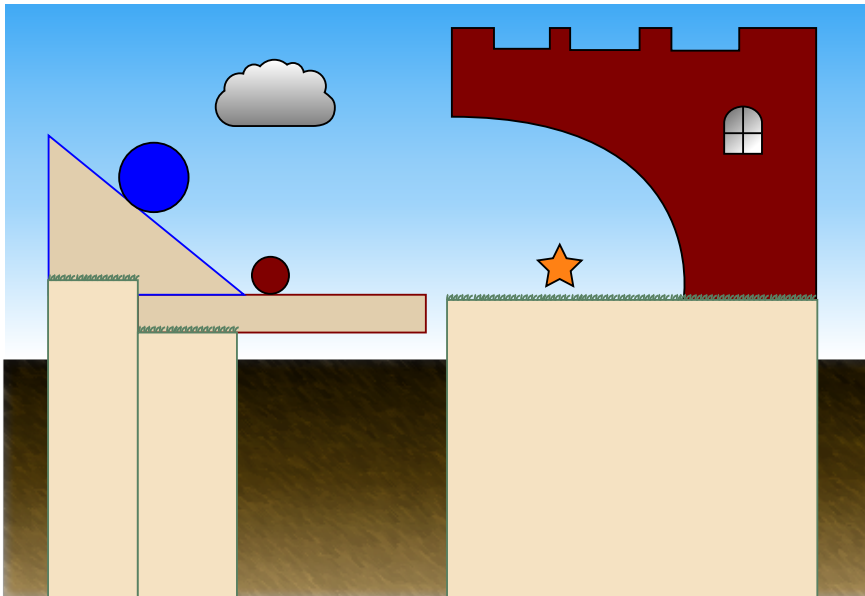
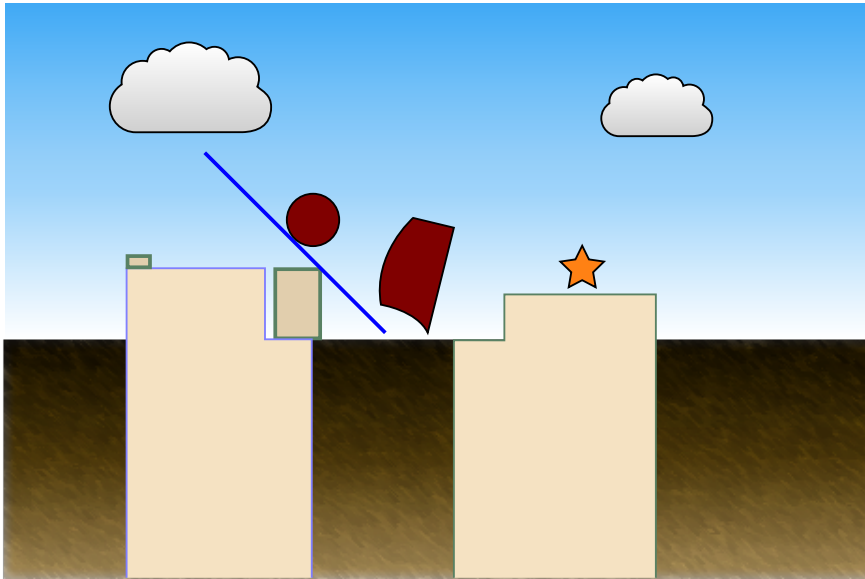


El ejemplo anterior puede servirnos de base para la creación de un videojuego. En él deberemos definir igualmente un mundo físico (*world*) y en este colocar los diferentes elementos de nuestro juego (*bodies*). Para determinar el movimiento de estas entidades, podemos cambiar las posiciones de estos cuerpos cuando se pulsa una tecla, por ejemplo; o, para darle un mayor realismo, aplicarle una fuerza y conseguir un salto realista de un modelo articulado. De toda la interacción entre los elementos colisionables se encargará Box2D de forma transparente al programador.

Terminaremos con este motor mostrando dos nuevos ejemplos implementados con esta tecnología para demostrar qué tipo de juegos y qué tipo de entretenimiento podemos llegar a ofrecer.

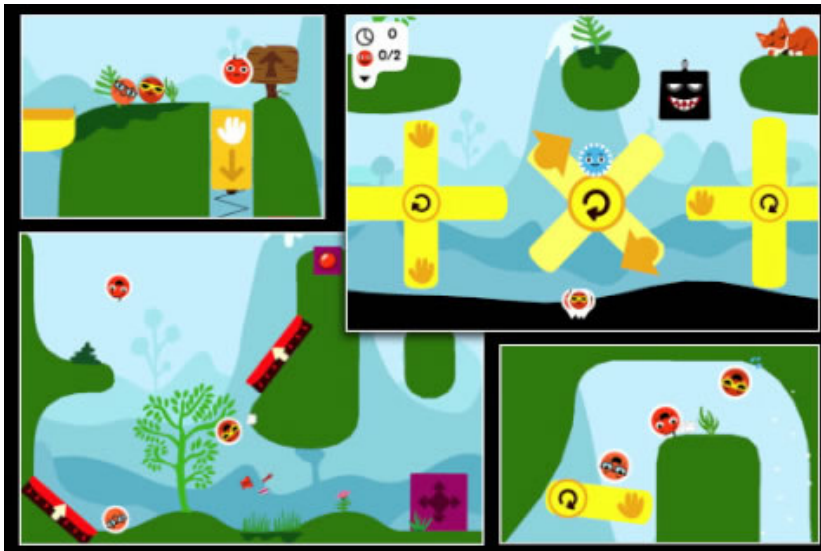
Ejemplos

El primero de ellos es "Crayon Physics Deluxe", de Petri Purho, uno de los proyectos ganadores del concurso Independent Games Festival del año 2008. Destaca por su sencillez y atractivo y su objetivo es guiar un objeto esférico a lo largo de un nivel. El jugador puede influir en el movimiento de la bola dibujando literalmente formas en la pantalla. He aquí unas figuras que ilustran dos momentos de este juego:



El juego "Crayon Physics Deluxe" utiliza el motor Box2D.

El otro juego que queremos mostrar es "Rolando", de ngmoco:). Otro sencillo y atractivo juego que gracias a la física ofrece una forma de entretenimiento basada en el movimiento de los personajes, que se consigue inclinando el dispositivo respecto al eje vertical, simulando el deslizamiento en un plano inclinado y la interacción con los dedos para mover según qué plataformas. Veamos aquí una captura de pantalla con el juego en acción:



El juego "Rolando" utiliza el motor Box2D.

4.2. Torque 2D physics

El motor de física Torque 2D physics es un módulo del motor de videojuegos Torque 2D, desarrollado por el estudio GarageGames, que en mayo del 2011 publicó la versión 1.5.

Según el propio estudio, se basa en el motor más potente y fácil de utilizar para desarrollar juegos 2D. Está construido sobre el núcleo de la arquitectura del motor para su versión genérica en 3D y está optimizado para el gameplay 2D. Permite desarrollarlo para las plataformas Windows, Mac, Xbox 360, Wii y iPhone.

Consolas

Para el caso de las consolas, es necesaria una licencia adicional.

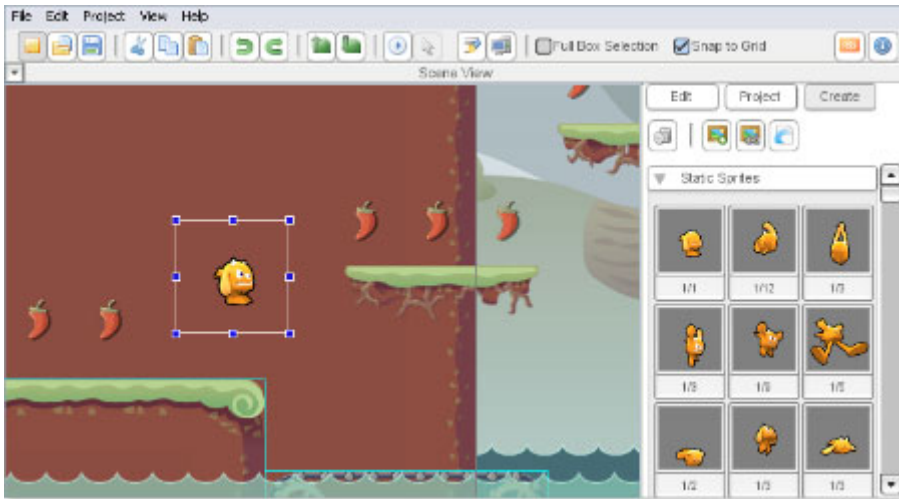
El motor de físicas aporta numerosas funcionalidades, entre las cuales encontramos:

- las posibilidades de aplicación de inercia,
- velocidad lineal,
- velocidad angular,
- fricción,
- restitución o amortiguación y
- otros conjuntos de atributos físicos que podemos aplicar a nuestros objetos del juego.

Además del motor físico, este *engine* proporciona un editor de niveles, una potente API gráfica 2D que se adapta a las diferentes arquitecturas hardware, definición de comportamientos para la inteligencia artificial, un sistema de

red para el modo multijugador, *scripting*, un sistema personalizado de creación de interfaces gráficas de usuario y un módulo para el sonido. En la siguiente ilustración, podemos apreciar una captura del editor de niveles.

Editor de niveles del motor de videojuegos Torque 2D.



Aparte de todo esto, Torque 2D, mediante licencias individuales, incorpora todo un conjunto de *kits* de desarrollo pensados para implementar juegos específicos a un género, como son los juegos de aventuras, plataformas y conducción.

A continuación, presentamos varios proyectos de ejemplos realizados con este motor con el fin de poder tener una idea del tipo de proyecto que podemos desarrollar con él y las posibilidades que ofrece.

Ejemplos de proyectos desarrollados con el motor Torque 2D.





5. Programación gráfica 2D

En este apartado daremos a conocer diferentes API gráficas con las cuales poder trabajar. Nos centraremos sobre todo en la solución OpenGL haciendo uso de la librería GLUT para el tratamiento de ventanas y eventos. Veremos las funciones que incorporan, y finalizaremos mostrando la API DirectX. Ambas API gráficas se explicarán teniendo en cuenta tan sólo el aspecto 2D al cual se refiere este módulo.

5.1. GLUT

Hoy en día y más en el caso de los videojuegos, casi toda aplicación corre bajo una ventana en la cual pueden tener lugar una serie de eventos efectuados por el usuario. Por tanto, para implementar una aplicación, debemos contemplar cómo gestionar una ventana y el *input* que se pueda generar en ella.

Para realizar esta tarea utilizaremos GLUT (*OpenGL utility toolkit*), que es una librería multiplataforma, libre y específica para OpenGL, sencilla y fácil de manejar, realizada por Mark J. Kilgard.

Webs

A continuación mostramos unos enlaces que pueden ser de especial interés:

- Página web
<ftp://ftp.sgi.com/sgi/opengl/glut/index.html>
- Especificación
<http://www.opengl.org/resources/libraries/glut/spec3/spec3.html>
- Versión de Nate Robins para Win32 con múltiples tutoriales especialmente didácticos para tratar las principales funcionalidades de OpenGL en tiempo real.
<http://www.xmission.com/~nate/glut.html>

Con esta librería el proceso de implementación es realmente sencillo:

- En primer lugar, es necesario registrar una serie de funciones que se encargarán de tratar los diferentes eventos o interrupciones que tienen lugar en la ventana donde corra el juego. Registraremos tantas funciones como diferentes eventos queramos tratar.
- Acto seguido, daremos el control de la aplicación a la librería y ésta, se encargará de llamar a las funciones definidas según sea necesario hasta que se desee finalizar la aplicación.

El objetivo de esta obra no es la programación de interfaces gráficas mediante una librería específica. Por eso, no entraremos en detalle en la programación con GLUT. Presentaremos dos ejemplos de código para, en primer lugar, ofre-

cer el aspecto general de ésta y, en segundo lugar, darle cabida en el contexto de desarrollo al que se quiere dirigir. El alumno podrá contrastar el significado de cada una de las funciones utilizadas mediante la especificación de GLUT indicada previamente.

A continuación, presentamos lo que sería una aplicación base utilizando GLUT, con el mínimo código posible.

```
#include <glut.h>

void render(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glutSwapBuffers();
}

int main(int argc, char *argv[])
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE);
    glutCreateWindow("First GLUT main program");
    glutDisplayFunc(render);
    glutMainLoop();
    return 0;
}
```

Utilizando el código que acabamos de mostrar y el de la estructura de un videojuego presentada con anterioridad, construiremos nuestro propio proyecto para desarrollar videojuegos.

```
//—GLUTmain.cpp—//

#include <glut.h>
#include "cGame.h"

cGame Game;

void AppRender() {
    Game.Render();
}

void AppKeyboard(unsigned char key, int x, int y) {
    Game.ReadKeyboard(key, x, y, true);
}

void AppKeyboardUp(unsigned char key, int x, int y) {
    Game.ReadKeyboard(key, x, y, false);
}

void AppSpecialKeys(int key, int x, int y) {
```

```
    Game.ReadKeyboard(key,x,y,true);
}
void AppSpecialKeysUp(int key, int x, int y) {
    Game.ReadKeyboard(key,x,y,false);
}
void AppMouse(int button, int state, int x, int y) {
    Game.ReadMouse(button,state,x,y);
}
void AppIdle() {
    if(!Game.Loop()) exit(0);
}

void main(int argc, char *argv[])
{
    //GLUT initialization
    glutInit(&argc, argv);

    //RGBA with double buffer
    glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE);

    glutInitWindowPosition(GAME_Xo,GAME_Yo);
    glutInitWindowSize(GAME_WIDTH,GAME_HEIGHT);
    glutCreateWindow("My Game");

    //Register callback functions
    glutDisplayFunc(AppRender);
    glutKeyboardFunc(AppKeyboard);
    glutKeyboardUpFunc(AppKeyboardUp);
    glutSpecialFunc(AppSpecialKeys);
    glutSpecialUpFunc(AppSpecialKeysUp);
    glutIdleFunc(AppIdle);

    //Game initializations
    Game.Init();

    //Application loop
    glutMainLoop();
}
//— cGame.h —//

#define GAME_Xo          100
#define GAME_Yo          100
#define GAME_WIDTH      640
#define GAME_HEIGHT     480

class cGame
{
```

```
public:
    //Main functions
    bool Init();
    bool Loop();
    void Finalize();

    //Input
    void ReadKeyboard(unsigned char key, int x, int y,
                     bool press);
    void ReadMouse(int button,int state, int x, int y);

    //Process
    bool Process();

    //Output
    void Render();

private:
    unsigned char keys[256];
};
//— cGame.cpp —//

#include "cGame.h"
#include <glut.h>

//Game initializations
bool cGame::Init()
{
}

//Game loop
bool cGame::Loop()
{
    bool end = false;

    end = Process();
    if(!end) Render();
    else Finalize();

    return end;
}

//Game finalizations
void cGame::Finalize()
{
}
```

```
//Game input
void cGame::ReadKeyboard(unsigned char key, int x, int y, bool press)
{
    keys[key] = press;
}

void cGame::ReadMouse(int button, int state, int x, int y)
{
}

//Game process
bool cGame::Process()
{
    bool end = false;

    //Process Input
    if(keys[27]) return true; //27=ESC

    //Game Logic
    //...

    return end;
}

//Game output
void cGame::Render()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glutSwapBuffers();
}
```

Este código base genera una aplicación con una ventana que es capaz de tratar los eventos de teclado y ratón, y presenta una estructura de código propia a la de un videojuego, sobre la cual podremos ir añadiendo nuestro código. Observemos con detenimiento cómo la clase *cGame* contiene las funciones principales de la estructura de un videojuego y en qué lugar del proyecto son llamadas cada una de ellas (en el código se encuentran en **negrita**). Fijémonos cómo mediante esta clase principal conseguimos la abstracción o independencia de nuestro proyecto al sistema de ventanas y sistema operativo utilizado.

Por último, es interesante señalar que esta aplicación se presenta en modo *windowed*, es decir, mediante una ventana con marcos. También es posible hacerlo en modo *fullscreen*, ocupando el tamaño completo de la pantalla.

Presentamos ambos códigos para cada uno de los dos casos:

- Windowed

```
glutInitWindowPosition(pos_x,pos_y);  
glutInitWindowSize(width,height);  
glutCreateWindow("Window caption");
```

- Fullscreen

```
glutGameModeString("800x600:32");  
glutEnterGameMode();
```

5.2. API Gráficas

Para llevar a cabo el desarrollo de un videojuego, es imprescindible apoyarse en una API gráfica. Este conjunto de funciones se ocupan de brindarnos el soporte necesario para poder visualizar en pantalla el estado del videojuego. Con esta finalidad podemos escoger entre diferentes opciones. Actualmente, las API más utilizadas son OpenGL y DirectX. Existen otras soluciones como SDL o Allegro dirigidas a un público menos exigente que busca mayor sencillez y simplicidad.

5.2.1. DirectX

DirectX es una colección de API creadas para facilitar la programación y dar acceso a las diferentes funcionalidades y ventajas hardware en la plataforma Microsoft Windows. Data de 1995 y actualmente disponemos de la versión 10.0 exclusiva para Windows Vista y 9.0c para versiones anteriores de Windows.

A continuación presentamos las diferentes API de las que está compuesto DirectX:

- Direct Graphics: para dibujado de imágenes en dos dimensiones (Direct-Draw) y representación de imágenes en tres dimensiones (Direct3D).
- DirectInput: utilizado para procesar datos del teclado, ratón, *joystick* y otros controles para juegos.
- DirectPlay: para comunicaciones en red.
- DirectSound: para la reproducción y grabación de sonidos.
- DirectMusic: para la reproducción de pistas musicales compuestas con DirectMusic Producer.
- DirectShow: para reproducir audio y vídeo.

Web

La página oficial de DirectX es <http://msdn.microsoft.com/directx/>

- DirectSetup: para la instalación de componentes DirectX (el instalable de un juego).

El aspecto más positivo de este conjunto de librerías es su amplio abanico de servicios al programador, así como el número de funcionalidades, tutoriales, ejemplos y ayuda en general, para cada uno de los módulos de los que consta.

Sin embargo, DirectX tiene dos grandes desventajas: no es portable y su continua actualización. En la mayoría de los casos el surgimiento de una nueva versión ha venido acompañado de un rediseño de la estructura de los módulos, así como el de los objetos y métodos implicados en cada uno de ellos.

5.2.2. OpenGL

OpenGL es una especificación estándar que define una API multilenguaje y multiplataforma para escribir aplicaciones que produzcan gráficos 2D y 3D. Fue desarrollada por Silicon Graphics Inc. (SGI) en 1992. Su nombre viene del inglés Open Graphics Library, cuya traducción es biblioteca de gráficos abierta (o mejor, libre, teniendo en cuenta su política de licencias).

OpenGL se utiliza en campos como CAD, realidad virtual, representación científica y de información, simulación de vuelo o desarrollo de videojuegos.

A grandes rasgos, OpenGL es una especificación, es decir, un documento que describe un conjunto de funciones y su comportamiento exacto. A partir de ella, los fabricantes de hardware crean implementaciones (bibliotecas de funciones creadas para enlazar con las funciones de la especificación OpenGL, utilizando aceleración hardware cuando sea posible). Dichos fabricantes tienen que superar pruebas específicas que les permitan calificar su implementación como una implementación de OpenGL.

Existen implementaciones OpenGL suministradas por fabricantes para Mac OS, Microsoft Windows, Linux, varias plataformas Unix, y PlayStation 3. Existen también varias implementaciones *software* que permiten que OpenGL esté disponible para diversas plataformas sin soporte de fabricante.

Por este motivo, si estamos desarrollando bajo OpenGL, lo primero que debemos hacer es actualizar el *driver* de nuestra tarjeta gráfica. Para ello deberemos dirigirnos a la página web del fabricante en cuestión, descargarlo e instalarlo. Por defecto, Microsoft Windows XP y versiones anteriores tan sólo incorporan la versión OpenGL 1.1, por lo que su actualización es primordial, si queremos obtener las ventajas de aceleración *hardware* de la tarjeta gráfica.

Aparte de la portabilidad, otro aspecto realmente positivo de OpenGL es la evolución que ha tenido desde su primera versión. OpenGL 2.1 fue lanzado el 2 de agosto del 2006 y se trata de la séptima revisión desde que viera la luz en el año 1992. A pesar de haber pasado de la versión 1.X a la 2.X (tan

Web

La página oficial de OpenGL es:
<http://www.sgi.com/products/software/opengl/overview.html>

Recomendación

En el momento de redacción, NO se recomienda la utilización de Microsoft Windows Vista para el desarrollo con OpenGL.

sólo para indicar el soporte a la programación con *shaders*), la versión actual es totalmente compatible con todas las anteriores (2.0, 1.5, 1.4, 1.3, 1.2, 1.1 y 1.0) con las ventajas que eso conlleva al programador. Si a ello sumamos la simplicidad y estructura con la cual OpenGL nos brinda las diferentes funcionalidades gráficas, obtenemos de ésta una herramienta eficiente, profesionalmente hablando, e ideal para la docencia.

Por otra parte, es necesario recalcar que OpenGL tan sólo es API gráfica, no da soporte a temas de sonido, captura de eventos o carga de imágenes de cualquier formato (sólo bmp). Este hecho no representa ningún problema añadido. Para obtener el soporte de funcionalidades no gráficas, deberemos utilizar otras librerías a tal efecto.

5.2.3. SDL

SDL es un conjunto de librerías desarrolladas con el lenguaje C que proporcionan funciones básicas para realizar operaciones de dibujo 2D, gestión de efectos de sonido y música, y carga y gestión de imágenes. SDL es una abreviatura en inglés de Simple DirectMedia Layer.

Una de sus grandes virtudes es que se trata de una librería multiplataforma, soportando oficialmente los sistemas Windows, Linux, MacOS y QNX, además de otras arquitecturas o sistemas como Dreamcast, GP32 o GP2X. De ahí le vienen las siglas SDL (*simple directmedia layer*) que más o menos alude a capa de abstracción multimedia.

Fue desarrollada inicialmente por Sam Lantinga, desarrollador de videojuegos para la plataforma Linux. La librería se distribuye bajo la licencia LGPL, que es la que ha provocado el gran avance y evolución de las SDL.

Existen una serie de librerías adicionales que complementan las funcionalidades y capacidades de la librería base. Las enumeramos a continuación:

- **SDL_Mixer.** Extiende las capacidades de SDL para la gestión y uso de sonido y música en aplicaciones y juegos. Soporta formatos de sonido como Wave, MP3 y OGG, y formatos de música como MOD, S3M, IT, y XM.
- **SDL_Image.** Extiende notablemente las capacidades para trabajar con diferentes formatos de imagen. Los formatos soportados son los siguientes: BMP, JPEG, TIFF, PNG, PNM, PCX, XPM, LBM, GIF, y TGA.
- **SDL_Net.** Proporciona funciones y tipos de dato multiplataforma para programar aplicaciones que trabajen con redes.
- **SDL_RTF.** Posibilita el abrir para leer en aplicaciones SDL archivos de texto usando el formato Rich Text Format RTF.

Ejemplo de librerías

A continuación se citan unas cuantas librerías a modo de ejemplo.

- Ventanas y eventos: GLUT, Qt, Tcl/Tk, GTK+, wxWidgets, FLTK, Win32
- Sonido: FMOD, OpenAL
- Texturas: Corona, glpng, DevIL, FreeImage
- Parser XML: TinyXML

Web

La página oficial de SDL es <http://www.libsdl.org/>

- `SDL_TTF`. Permite usar fuentes TrueType en aplicaciones SDL.

5.2.4. Allegro

Allegro es una librería para programadores de C/C++ orientada al desarrollo de videojuegos, originalmente escrita por Shawn Hargreaves para el compilador DJGPP y distribuida libremente.

Funciona en las siguientes plataformas: DOS, Unix (Linux, FreeBSD, Irix, Solaris), Windows, QNX, BeOS y MacOS X. Ofrece funcionalidades de gráficos, sonidos, entrada del usuario (teclado, ratón y *joystick*) y temporizadores. También dispone de funciones matemáticas en punto fijo y coma flotante, funciones 3d, funciones para manejar ficheros, ficheros de datos comprimidos y una interfaz gráfica.

Es intuitiva y fácil de usar, sin embargo, igual que la librería SDL, no se utiliza en el ámbito profesional de la creación de videojuegos.

5.3. OpenGL

Como hemos dicho, OpenGL es una interfaz software independiente del sistema operativo para el acceso directo al hardware gráfico. Fue diseñada por Silicon Graphics basándose en una década de experiencia en hardware y software de visualización. Las siglas GL vienen de *graphics library*, pues precisamente lo que OpenGL suministra al programador es una interfaz compuesta por varios centenares de procedimientos y funciones, divididas en dos grupos: las de interacción con el *buffer* gráfico (ubicación de la memoria de almacenamiento temporal) y las de control. Estas rutinas permiten al programador generar imágenes de alta calidad de forma eficiente. Si bien el diseño de OpenGL se orienta a los gráficos 3D interactivos, también proporciona funcionalidades de gráficos 2D de calidad.

OpenGL supone una capa de abstracción entre el hardware gráfico y las aplicaciones. Globalmente, las rutinas de OpenGL conforman una interfaz de programación de aplicaciones. Estas llamadas permiten el dibujo de primitivas gráficas (puntos, líneas, polígonos, *bitmaps* e imágenes) en el *frame-buffer*. Empleando las primitivas disponibles y las operaciones de control de su dibujo, podremos generar imágenes a pleno color 3D de forma sencilla.

OpenGL fue diseñado con los lenguajes C y C++ en mente, y con ellos se consigue un mayor acoplamiento. Sin embargo, adaptaciones para lenguajes como Tcl/Tk, Fortran, Java o ADA se encuentran a disposición del público.

Web

La página web oficial en su versión en castellano es la siguiente:
<http://alleg.sourceforge.net/index.es.html>

Web

La especificación de la versión actual OpenGL 2.1 la podemos encontrar en el enlace siguiente:
<http://www.opengl.org/documentation/specs/version2.1/glspec21.pdf>

Lenguaje de referencia

En este apartado tomaremos el C como lenguaje de referencia.

Una característica importante de OpenGL es que se trata de un estándar, a diferencia de otras API gráficas que son dependientes de un único fabricante.

OpenGL fue diseñado por Silicon Graphics, pero actualmente es un estándar gobernado por un conjunto de fabricantes. Esta característica garantiza la disponibilidad del software para cualquier plataforma conocida, y por ello el código OpenGL es multiplataforma, y resulta simple de portar a los más diversos entornos. Tan sólo es necesario cambiar aquella parte de nuestro programa que interactúe con el sistema operativo. No olvidemos que OpenGL es independiente de estos detalles.

En consecuencia, esta estandarización conlleva que OpenGL define un conjunto cerrado de funcionalidades e interfaces de uso: un sistema que implemente OpenGL lo ha de hacer al 100%, hasta la última de las llamadas. De hecho, no se puede comercializar una implementación de OpenGL hasta que el gabinete de arquitectura de OpenGL (su máximo órgano de gobierno) certifique que efectivamente se cumplen todos los requisitos.

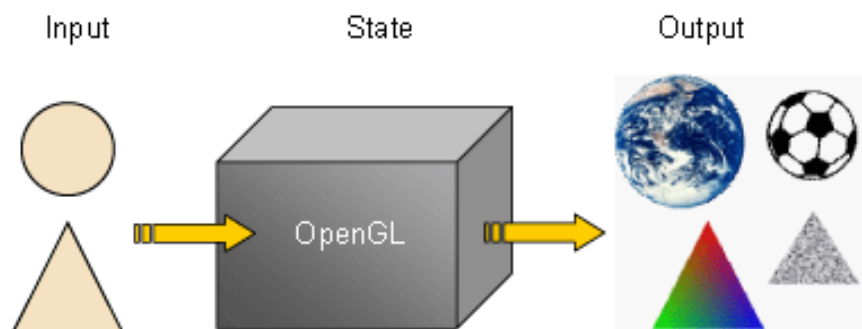
5.3.1. Estructura

OpenGL especifica el sistema gráfico en forma de una máquina de estados que controla un conjunto conocido de operaciones de dibujado. Las rutinas de control que OpenGL proporciona son un medio para modificar el estado de la máquina, y así alterar la forma en la que las operaciones de dibujado generan sus resultados.

Podríamos pensar en OpenGL como en una máquina compuesta por palancas y botones, con una entrada y una salida. Nosotros introducimos datos gráficos en la entrada, y la máquina los procesa y los devuelve, transformados, a través de la salida. En función de los controles, la salida será una u otra. Esta salida siempre es una descripción 2D rasterizada (discretizada en pantalla mediante píxeles) de la escena de entrada, pero las palancas y botones controlarán si se le aplica iluminación, color o cualquier otro efecto deseado.

Además, la máquina de estado de OpenGL no es opaca: en cualquier momento podemos consultar el estado de cualquier variable, del mismo modo que podemos alterar su valor. Esta arquitectura es especialmente intuitiva, al tiempo que permite escribir código breve pero potente.

En la siguiente ilustración podemos observar este tipo de funcionamiento, basado en una máquina de estados. Con idéntica geometría pero estados diferentes se pueden obtener resultados totalmente distintos:



El modelo para interpretar comandos OpenGL es el de cliente-servidor. De hecho, OpenGL permite ejecución remota y, lo más habitual, local.

OpenGL es una librería de bajo nivel que opera en lo que se llama modo inmediato. El modo inmediato significa que, en el momento en que se ejecuta una llamada a OpenGL, se actualiza el *buffer* gráfico adecuado con los resultados. En este sentido, OpenGL es similar a DirectDraw o Direct3D de Microsoft.

Por contra, las librerías de modo retenido (Inventor, Performer) se usan por encima de las de modo inmediato, para generar descripciones de escenas, operaciones, etc. Usualmente, las librerías retenidas no son más que *wrappers* de librerías inmediatas más sencillas.

Finalmente, debemos comentar que OpenGL se entrega como dos componentes:

- Por un lado, la librería en sí, cuya cabecera se presenta en el fichero GL/gl.h.
- Por otro lado, la librería de utilidades GLU (*OpenGL utility library*) en el fichero GL/glu.h. Esta librería extra ofrece *wrappers* para algunas funciones de OpenGL, simplificando su uso de manera significativa.

5.3.2. Geometría

La definición de geometría en OpenGL no puede ser más simple. Su sintaxis es la siguiente:

```
glBegin(TIPO_DE_OBJETO)
    datos del objeto
glEnd();
```

Puede verse como todas las llamadas de OpenGL empiezan con las siglas "gl". A continuación, veremos el código para pintar un triángulo con interpolación suave de colores entre los vértices:

```
glBegin(GL_TRIANGLES);
    glColor3f(1.0,1.0,1.0);
    glVertex3f(0.0,0.0,0.0);
    glColor3f(1.0,0.0,0.0);
    glVertex3f(3.0,5.0,0.0);
    glColor3f(0.0,1.0,0.0);
    glVertex3f(-3.0,5.0,0.0);
glEnd();
```

Con este código definimos un triángulo (GL_TRIANGLES) con vértices en (0,0,0), (3,5,0) y (-3,5,0). Cada vértice es de un color diferente y OpenGL hará una gradación de colores para los puntos intermedios. Éste es el comportamiento por defecto. Más adelante veremos cómo puede ser modificado.

La nomenclatura estándar de colores en OpenGL es la RGB, pudiendo ser cada valor un real comprendido entre 0.0 y 1.0 o un entero entre 0 y 255 según el tipo de llamada utilizada. También se soporta el modo RGBA, y modos paletizados a 256 colores.

Nomenclatura de colores de referencia

En este apartado nos basaremos en el modo RGB por ser el más extendido.

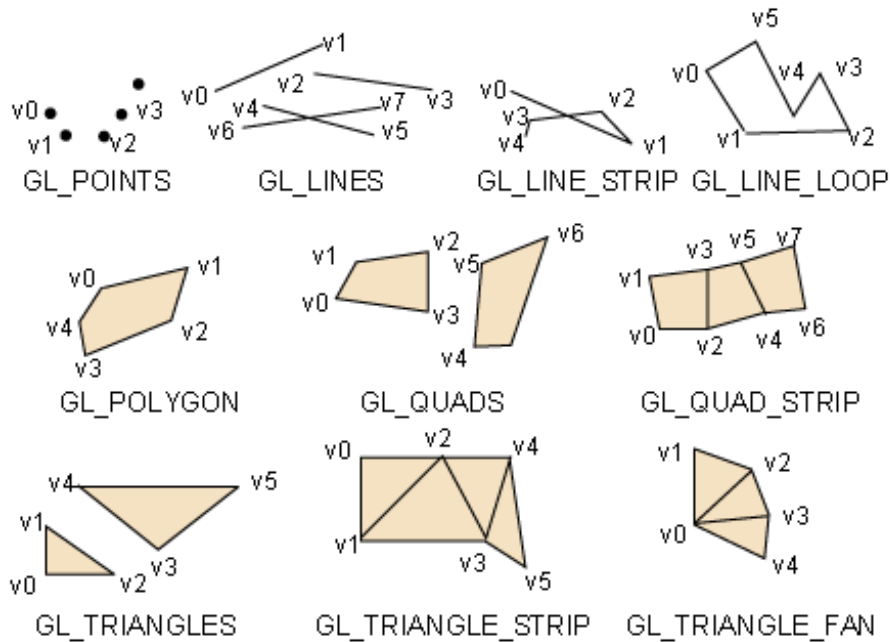
De este ejemplo de código que acabamos de ver podemos observar lo siguiente:

- Todas las constantes de OpenGL poseen el prefijo GL_ (como GL_TRIANGLES).
- Las llamadas con parámetros numéricos acaban con el número de parámetros y su tipo. Así, la llamada "glVertex3f" recibe 3 parámetros de coma flotante. Los tipos disponibles son:

Tipo de OpenGL	Espacio en memoria	Tipo de C	Sufijo
Glbyte	Entero de 8 bits	signed char	b
Gushort	Entero de 16 bits	short	s
GLint, Glsizei	Entero de 32 bits	long	l
Gfloat, Glclampf	Flotante de 32 bits	float	f
Gldouble, Glclampd	Flotante de 64 bits	double	d
Glubyte, Glboolean	Entero sin signo de 8 bits	Unsigned char	ub
Glushort	Entero sin signo de 16 bits	Unsigned short	us
Gluint, Glenum, Glbitfield	Entero sin signo de 32 bits	Unsigned long	ui

- Finalmente, la aparición de una letra "v" indica un vector del tipo que indica la siguiente letra del resto del nombre. Así, la primitiva "glVertex3fv" requiere como parámetro un *array* de números *float*.

Del mismo modo que tenemos GL_TRIANGLES, OpenGL proporciona las siguientes primitivas de pintado. Nótese la importancia del orden de la declaración de vértices denotada en la ilustración mediante los subíndices del nombre de los vértices.



Las primitivas de dibujado siempre deben aparecer entre un `glBegin` y un `glEnd`.

Tira de triángulos

Por lo general, el tipo de objeto más utilizado es la tira de triángulos (`GL_TRIANGLE_STRIP`). Para dar una justificación a este hecho, efectuaremos el siguiente ejercicio práctico.

Mediante una representación basada en triángulos (`GL_TRIANGLES`), queremos modelar un objeto compuesto por un millar de triángulos. El número de vértices que deberemos enviar a nuestra tarjeta gráfica será exactamente 3000, pues necesitaremos tres vértices para cada triángulo. Sin embargo, si pudiéramos describir esta malla de triángulos mediante la primitiva `GL_TRIANGLE_STRIP`, el número de vértices sería 1002. De esta manera, hacemos que el número de vértices necesarios para representar un triángulo disminuya a la unidad, con la consecuente ganancia en reducción de vértices a tratar.

Mediante la primitiva `GL_TRIANGLE_STRIP` podemos describir una malla de n triángulos utilizando tan sólo $n+2$ vértices. Esto supone una gran ventaja para cualquier técnica que vayamos a utilizar después tales como selección de objetos, sombras, iluminación, *frustum culling* o *occlusion queries*, pues el número de datos con el que trabajaremos será muy inferior.

Para videojuegos en dos dimensiones, en cambio, el volumen de información con el que trabajamos no suele ser significativo, por lo que, en general, GL_QUADS suele ser la primitiva más empleada.

5.3.3. La máquina de estados

A continuación, veremos las llamadas de control de estado.

Cada vez que alteramos el estado, el cambio persiste hasta que volvemos a alterarlo. Por ejemplo, la llamada glColor modifica el valor de todos los vértices definidos tras ella, hasta que no tenga lugar una nueva llamada a esta función.

Tratemos el siguiente caso en el cual tiene lugar la definición de un cuadrado.

```
glColor3f(1.0,1.0,1.0);
glBegin(GL_QUADS)
    glVertex2f(0.0,0.0);
    glVertex2f(10.0,0.0);
    glVertex2f(10.0,10.0);
    glColor3f(0.0,0.0,1.0);
    glVertex2f(0.0,10.0);
glEnd();
```

En este ejemplo, el primer glColor altera el color de los tres primeros vértices a (1.0,1.0,1.0), que es el color blanco, mientras que el cuarto vértice queda afectado por la última definición del estado color que pasa a ser azul.

A medida que las primitivas geométricas son rasterizadas y pintadas, OpenGL actúa según lo que le indican sus variables de estado. Éstas contienen información sobre el modo de renderizado (*render*), el ancho de línea, colores, texturas a aplicar, etc. Podemos clasificar estas variables de estado en tres grupos, según el dominio de posibilidades que permiten: binarias, de modo y valor.

Variabes binarias

Las variables binarias tienen dos estados: activo (*on*) y desactivado (*off*). En este caso, la forma de controlarlas es sencilla, y consiste en utilizar las dos llamadas siguientes:

```
glEnable( variable_de_estado );
glDisable( variable_de_estado );
```

Ejemplos de variables binarias de estado

- GL_CULL_FACE: si activado, aplica *culling*
- GL_DEPTH_TEST: si activado, usa *Z-buffer*
- GL_FOG: si activado, aplica niebla
- GL_LIGHTING: si activado, aplica iluminación
- GL_NORMALIZE: si activado, auto-normaliza (escala a módulo uno) las normales

- `GL_TEXTURE_2D`: si activado, permite texturas 2D

Variables de modo

Una variante de las variables binarias son las de modo, que permiten N estados posibles. En este caso se emplean comandos específicos para cada variable.

Ejemplo de variable de modo

He aquí un ejemplo en el cual especificamos que el tipo de pintado sea plano (sin interpolar los colores de los vértices) o interpolado.

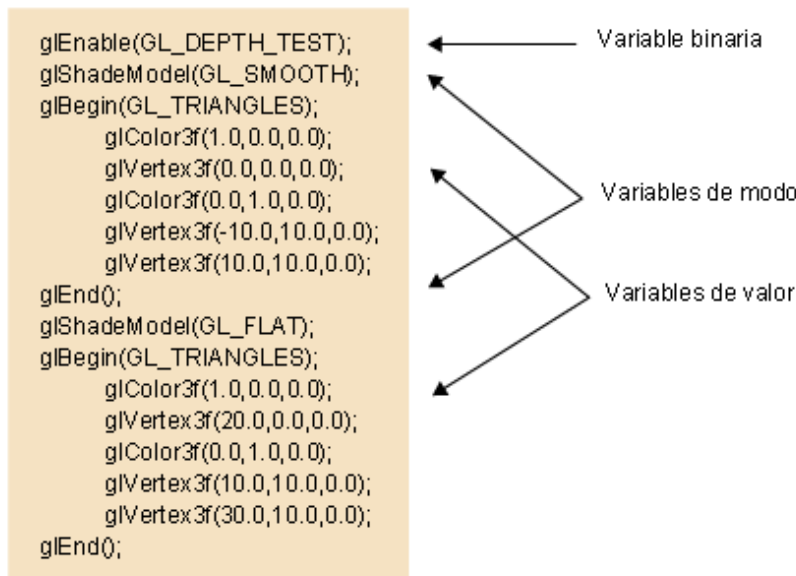
```
glShadeModel(GL_FLAT o GL_SMOOTH)
```

Variables de valor

Las variables de valor son aquellas que guardan alguna información que deseamos que OpenGL emplee directamente, como por ejemplo colores. La llamada más común de esta familia es precisamente `glColor`, si bien tenemos otras aplicaciones, por ejemplo, para controlar el tamaño de los puntos o líneas.

Ejemplo de variables de control de estado

Veamos un ejemplo final donde encontramos un uso de los diferentes tipos de variables de estado que acabamos de describir.



Este código visualiza dos triángulos de tamaño idéntico, uno con una interpolación de colores y el otro sin ella. El segundo aparecerá de un color plano, mientras que el primero lo hará con una gradación de los colores de los vértices. Este cambio de modo se debe a las variables de estado que hemos definido entre ambos.

5.3.4. Transformaciones geométricas

OpenGL emplea matrices para decidir cómo transformar nuestra geometría. Las operaciones que podemos realizar son translaciones, rotaciones y escalados. Con este propósito, OpenGL proporciona tres simples llamadas que harán el trabajo por nosotros. Sus cabeceras:

Traslación

```
glTranslate{f/d}( GLfloat x, GLfloat y, GLfloat z)
```

La geometría definida tras esta llamada será trasladada según los parámetros indicados.

Rotación

```
glRotate{f,d}(GLfloat angulo, GLfloat x, GLfloat y, GLfloat z)
```

Ésta es la llamada de rotación libre respecto a un eje. Las tres coordenadas indican el eje de giro, el ángulo y la magnitud en grados. Si queremos hacer una rotación de diferente magnitud en los tres ejes, deberemos llamar tres veces a esta función introduciendo en cada caso el ángulo que se desee y los ángulos de giro (1, 0, 0), (0, 1, 0) y (0, 0, 1).

Escalado

```
glScale{f,d} (GLfloat x, GLfloat y, GLfloat z)
```

Esta tercera llamada multiplicará respectivamente cada coordenada de vértice por los valores introducidos. Si introducimos los parámetros (1, 1, 1), el resultado dejará la misma geometría intacta.

Las transformaciones en OpenGL se aplican en el sentido inverso que se declaran. De este modo, en el ejemplo del código que tenemos a continuación, la geometría quedará primero traslada a lo largo del eje z, después se aplicará una rotación de cinco grados, y finalmente un escalado uniforme.

```
glScalef(3.0, 3.0, 3.0);  
glRotatef(5.0, 1.0, 0.0, 0.0);  
glTranslatef(0.0, 0.0, 5.0);
```

A la hora de trabajar con matrices, debemos tener en cuenta dos conceptos: la matriz identidad y la creación de ámbitos de contexto.

La siguiente función sirve para resetear la pila, introduciendo la matriz identidad (diagonal a unos y resto a ceros). Esta matriz, al ser aplicada a la geometría entrante, se queda intacta.

```
glLoadIdentity()
```

Otro aspecto muy importante a la hora de trabajar con matrices de transformación consiste en decidir qué elementos y cómo deben ser transformados de manera independiente. Para ello, resulta imprescindible la creación de ámbitos de contexto. Podemos hacerlo con las siguientes dos funciones:

```
glPushMatrix()
```

Esta llamada duplica la matriz que esté en la cima de la pila, para tener así una copia con la que trabajar sin perder la original.

```
glPopMatrix()
```

Esta llamada elimina la matriz de la cima de la pila de matrices.

Ejemplo de transformación

Veamos un ejemplo práctico de cómo una transformación puede ser aislada o aplicada sobre toda la escena.



5.3.5. Matrices de proyección y de visión del modelo

OpenGL trabaja con diferentes tipos de matrices, cada una para un propósito diferente:

- `GL_PROJECTION`: Matriz de proyección. Permite definir la cámara escogiendo entre una proyección perspectiva o paralela así como sus atributos.

- `GL_MODELVIEW`: Matriz de visión del modelo. Sirve para especificar cómo queremos ver el modelo geométrico. Habitualmente se emplea para colocar la cámara.
- `GL_TEXTURE`: Matriz de texturizado. Se emplea para definir cómo se aplicarán las texturas a nuestra geometría.

El matriz texturizado no será abordado en este módulo y lo dejaremos para cuando tratemos la programación gráfica 3D.

La idea es que nosotros seleccionamos la matriz sobre la que deseamos trabajar, y a continuación aplicamos transformaciones para definirla o alterarla. A partir de entonces nuestra geometría se verá modificada tal y como se haya establecido.

La función que permite seleccionar la matriz que deseamos manipular es:

```
glMatrixMode (GL_PROJECTION o GL_MODELVIEW);
```

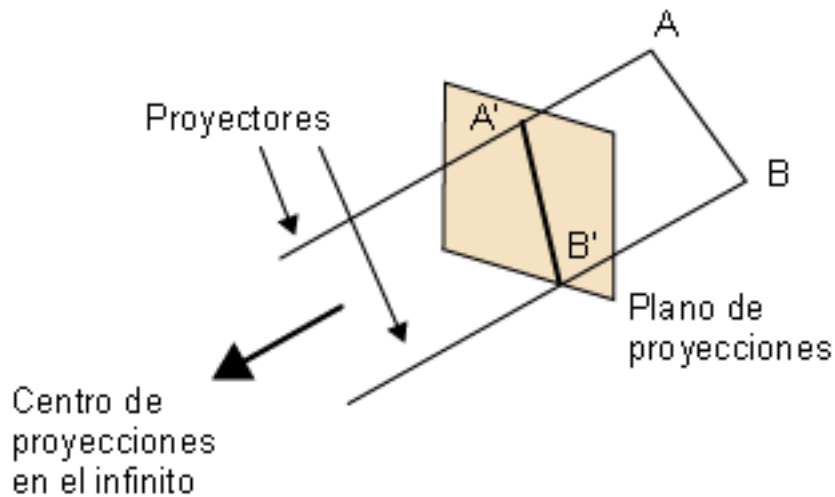
A la hora de definir la matriz de proyección, se nos presenta la posibilidad de escoger entre una vista con cámara paralela o perspectiva:

Cámara paralela

Viene determinada por la dirección de proyección y los proyectores son paralelos, ya que el centro de proyecciones está en el infinito. Se utiliza en diseños de ingeniería, planos de arquitectura, catálogos, diseño de muebles, etc. Éstas son sus principales características:

- Ventajas: se puede utilizar para mediciones exactas escalando en los ejes, las líneas paralelas permanecen como tales.
- Inconvenientes: no es realista.

Veamos a continuación un esquema y una representación 3D de un cilindro haciendo uso de esta cámara.



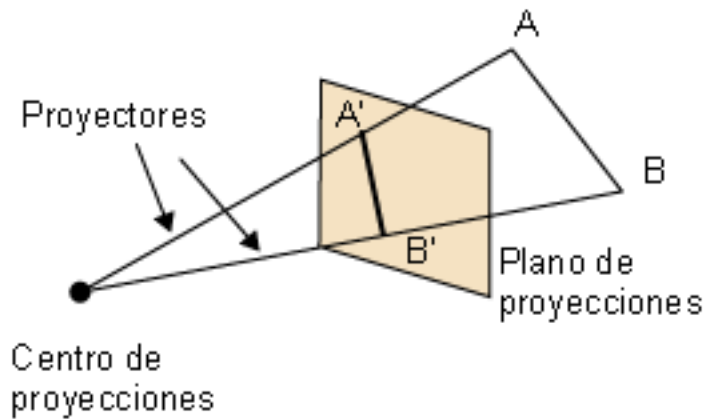
Cámara perspectiva

Viene determinada por el centro de proyecciones. Se utiliza en publicidad, presentación de diseños de arquitectura, diseño industrial, ingeniería, arte, animación, etc. Sus principales características son:

- Ventajas: proporciona realismo visual y sensación tridimensional.
- Inconvenientes: no mantiene la forma del objeto ni la escala (excepto en los planos paralelos al plano de proyecciones).

A grandes rasgos, difiere de la proyección paralela donde el tamaño de los objetos disminuye con la distancia puesto que la representación es realista y mantiene la profundidad de los objetos.

Veamos a continuación un esquema y una representación 3D de un cilindro haciendo uso de esta cámara:



5.3.6. Definición de cámara

En principio, es posible definir una cámara a mano mediante las funciones de transformación vistas con anterioridad de rotación y traslación. Sin embargo, OpenGL provee una serie de llamadas para poder llevar a cabo este trabajo.

Cámara perspectiva

```
glFrustum (GLfloat left, GLfloat right, GLfloat bottom,
          GLfloat top, GLfloat near, GLfloat far);
```

Función básica para crear matrices de perspectiva. Los parámetros son las distancias a los planos izquierdo, derecho, superior, inferior, frontal y trasero de la pirámide de proyección. Esta función no suele utilizarse en detrimento de `gluPerspective`, cuyos parámetros de definición resultan más interesantes.

Cámara paralela

```
glOrtho (GLfloat left, GLfloat right, GLfloat bottom, GLfloat top, GLfloat near, GLfloat far);
```

Similar a la cámara perspectiva, pero para el caso perspectiva ortoédrica (paralela).

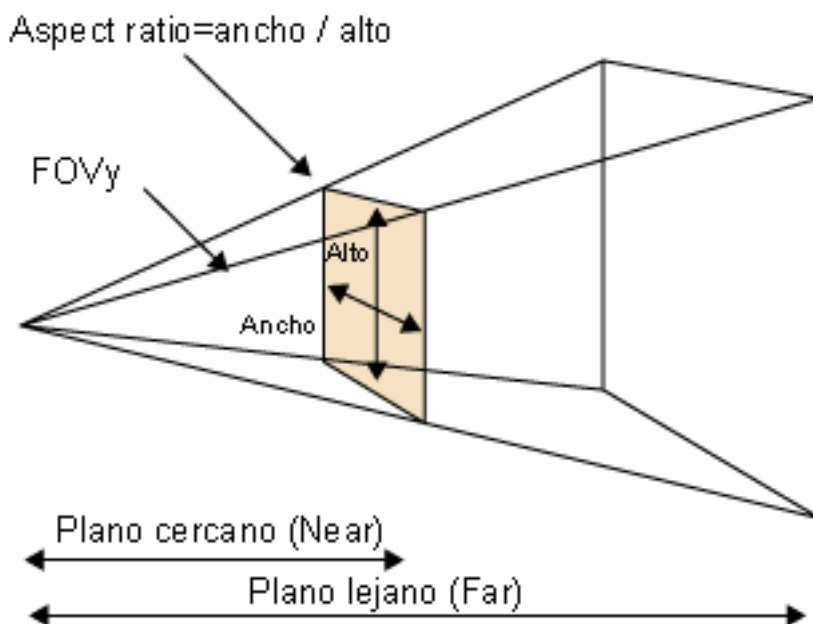
Cámara perspectiva con GLU

Las dos llamadas por excelencia para trabajar con matrices bajo OpenGL no están en la librería GL propiamente, sino en la de utilidades GLU. Son las llamadas siguientes:

```
gluPerspective (GLfloat fovy, GLfloat aspect, GLfloat near, GLfloat far);
```

Esta llamada genera una matriz de perspectiva cónica adecuada a los parámetros que le pasemos. El primero, *fovy*, es el *field of view* del eje y, es decir, la abertura de la cámara verticalmente. En segundo lugar, el *aspect* nos indica la relación entre el ancho y el alto de la pantalla. Este parámetro debe contener el valor resultante de dividir el ancho por el alto del *frame-buffer*. Los dos últimos parámetros indican los planos de recorte. OpenGL usa dos planos de recorte: frontal y posterior. La geometría que quede fuera de la pirámide de visión definida por estos dos planos (y los ángulos de abertura) será descartada y no se rasterizará.

La siguiente ilustración permite ubicar fácilmente cada uno de estos parámetros:



```
gluLookAt (eyex, eyey, eyez, centerx, centery, centerz, upx, upy, upz);
```

Esta segunda llamada genera una matriz de transformación de cámara. La cámara o punto de vista quedará situada en (eyex, eyey, eyez) y mirará hacia el punto (centerx, centery, centerz). El parámetro (upx, upy, upz) define el *view*

up vector que sirve para definir la orientación vertical. Este parámetro es útil para poder definir *rolls* de cámara como por ejemplo la inclinación de un avión cuando gira.

5.3.7. Definición de vistas (*viewports*)

Cada vez que definimos un área de *render* es necesario definir el *viewport*. Esto puede suceder cuando se ejecuta el videojuego (se crea una ventana) o cuando el tamaño de la ventana cambia, por ejemplo. La siguiente llamada permite realizar la definición del *viewport*.

```
glViewport (int x, int y, int wide, int tall)
```

Los parámetros se corresponden al área de la ventana. Si queremos que el espacio de renderizado sea toda la ventana, los valores a introducir serán el punto (0, 0) y el tamaño y el alto de la ventana.

Es común en un videojuego tener una pantalla dividida cuando nos encontramos en una visualización con varias cámaras a la vez, o cuando estamos en modo multijugador, donde tenemos dedicada para cada jugador un área de la pantalla. Precisamente, ésta será la función que se encargará de hacer esta división. En cada pasada de visualización de nuestro ciclo de juego definiremos un *viewport*, haremos un renderizado y repetiremos el proceso para cada nueva área que se quiera obtener.

Veamos algunas muestras de juegos donde se utiliza esta funcionalidad:

4 Viewports



Mario Kart: Double Dash!!

2 Viewports



Lotus Esprit Turbo Challenge

5.3.8. Colorear el fondo

OpenGL trabaja con varios *buffers*, entre ellos el de color, que es donde se dibuja la imagen resultante. Para inicializar o borrar este *buffer*, es necesario escribir en él utilizando un único color.

Con este propósito existen dos funciones. La primera de ellas nos permite declarar cuál es el color con el cual queremos inicializar el *buffer* de color y tiene el siguiente aspecto:

```
void glColor (GLclampf red, GLclampf green, GLclampf blue, GLclampf alfa)
```

La segunda función da la orden de limpieza. El comando que se utiliza para limpiar un *buffer* en OpenGL es genérico para cualquiera de ellos y es mediante el parámetro o *flag* como se indica el *buffer* sobre el cual se desea actuar. Nos estamos refiriendo al campo bits de la siguiente función:

```
void glClear (GLuint bits)
```

En el caso del *buffer* de color, debemos proceder de la siguiente manera:

```
glClear (GL_COLOR_BUFFER_BIT);
```

Si deseáramos inicializar más de un *buffer* a la vez, podríamos hacerlo utilizando el operador binario OR entre cada flag.

5.3.9. Modos de visualización

Los polígonos pueden visualizarse de tres modos:

- Sólido: rellenando con el color actual.
- Alámbrico o *wireframe*: sólo se visualizan las aristas.
- Puntos: sólo se visualizan los vértices.

Además, podemos visualizar la parte exterior y la parte interior de los polígonos de dos modos diferentes. Una vez cambiado el modo de visualización, todos los polígonos que definamos a continuación se visualizarán en ese modo, hasta que se cambie de nuevo. La función que se debe utilizar para ello es la siguiente:

```
void glPolygonMode (GLenum cara, GLenum modo)
```

El parámetro "cara" indica la cara a la que aplicamos el modo de visualización. Puede valer GL_FRONT (cara exterior), GL_BACK (cara interior) o GL_FRONT_AND_BACK (ambas caras).

El segundo parámetro "modo" indica el modo de visualización que se aplica. Puede valer GL_FILL (modo sólido; es el modo por defecto), GL_LINE (modo alámbrico) o GL_POINT (modo puntos).

5.3.10. Superficies ocultas: *Z-buffer* y *Culling*

Cuando visualizamos un objeto, aquellos polígonos que se encuentran detrás de otros, desde el punto de vista del observador, no deben dibujarse. El algoritmo de eliminación de caras ocultas más conocido es el del *Z-buffer*. OpenGL mantiene un *Z-buffer* (o *buffer* de profundidad) para realizar la ocultación de caras, cuyo manejo es muy sencillo. Consiste en una matriz en la que se almacena la profundidad (el valor de la componente z) para cada píxel. De esta manera es posible conocer qué elementos se encuentran delante de otros.

El algoritmo del *Z-buffer* es bastante costoso. Para incrementar la velocidad de proceso, OpenGL permite eliminar previamente las caras interiores de los objetos (*culling*). Este proceso es válido para objetos cerrados, en los que las caras interiores no son visibles. En el caso de objetos abiertos, utilizar *culling* puede producir efectos no deseados. El uso de la eliminación de caras interiores debe hacerse con precaución, puesto que la interioridad o exterioridad de las caras depende de su orientación.

Para activar el *Z-buffer* y el proceso de *culling* se utilizan las funciones glEnable y glDisable. Para el *Z-buffer* escribiremos el siguiente código:

```
glEnable(GL_DEPTH_TEST);  
glDisable(GL_DEPTH_TEST);
```

Para tratar el *culling* procederemos de igual manera:

```
glEnable(GL_CULL_FACE);
```

```
glDisable(GL_CULL_FACE);
```

En cuanto al *Z-buffer*, debe ser inicializado cada vez que se visualiza la escena, mediante la función `glClear`, igual que hemos hecho con el *buffer* de color. Como hemos comentado con anterioridad, para realizar ambas operaciones a la vez podemos hacerlo de la siguiente manera:

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

5.3.11. Modelos de sombreado

La función `glColor` que ya hemos visto sirve para seleccionar el color en el que se dibujan todos los elementos que se diseñen a continuación. Cuando definimos una primitiva de diseño, el color se asigna a cada uno de sus vértices y no a la primitiva completa. Esto puede ocasionar situaciones en las cuales tengamos vértices de diferente color en una misma primitiva. La pregunta que nos puede surgir es obvia: ¿cómo debe colorearse el interior de la primitiva?

El coloreado del interior depende del modelo de sombreado o *shading model*. Éste puede ser plano o suavizado. El modelo de sombreado se escoge con la función siguiente y afecta a todas las primitivas que se definan posteriormente, hasta que sea alterado de nuevo.

```
void glShadeModel (GLenum modo)
```

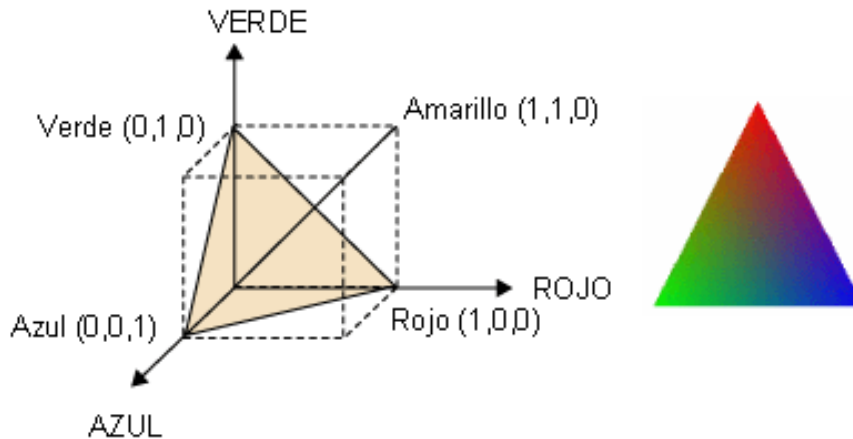
El modelo de sombreado elegido puede ser `GL_FLAT`, para el modelo plano o `GL_SMOOTH`, que es el modelo suavizado llamado modelo de Gouraud, que es el valor por defecto.

En el modelo plano (`GL_FLAT`) el interior del polígono se colorea completamente a un único color. Este color es el correspondiente al primer vértice de la primitiva.

En el modelo suavizado (`GL_SMOOTH`) el interior de las primitivas se colorea realizando una interpolación o suavizado entre los colores asociados a los vértices.

Evidentemente, el modelo de sombreado elegido sólo afecta cuando los vértices de las primitivas tienen colores distintos. Si los colores fueran iguales, la representación equivale al modelo plano.

Veamos un ejemplo práctico donde se aprecia el modelo suavizado bajo un triángulo con tres vértices de diferente color:



5.3.12. *Display lists*

Hemos comentado que OpenGL es una API de modo inmediato, en el sentido que los comandos OpenGL se ejecutan al instante, y afectan inmediatamente a los contenidos del *frame-buffer*. Sin embargo, esto no es cierto para el caso de las *display lists*.

Una *display list* o lista de visualización es una secuencia de comandos OpenGL, que almacenamos en la tarjeta gráfica para que se ejecuten uno detrás de otro en el momento en que nosotros así lo indicamos. Para ello, vamos almacenando nuestras diferentes *display lists* y OpenGL nos asigna un identificador para cada una de ellas. De este modo, indicando el identificador en cuestión hacemos referencia a la lista de comandos que queremos efectuar.

Esta técnica permite obtener un beneficio del rendimiento muy considerable puesto que tan sólo es necesario utilizar una vez el bus de la tarjeta gráfica, uno de los principales cuellos de botella en las aplicaciones gráficas, para indicar el identificador.

El tratamiento de *display lists* con OpenGL radica en las siguientes llamadas. Para crearla, utilizamos el siguiente método:

```
glNewList(GLuint name, GLenum mode)
```

El primer parámetro es un número que actuará de identificador de la lista, y el segundo es el modo en que queremos que OpenGL la cree. Hay dos modos posibles:

- **GL_COMPILE:** OpenGL compila (pero no ejecuta de momento) los comandos de la lista; éstos se ejecutarán sólo cuando se llame a la lista.

- `GL_COMPILE_AND_EXECUTE`: OpenGL almacena los comandos, pero además los ejecuta de momento, produciendo resultados instantáneos.

Para finalizar la descripción de la *display list*, es necesario llamar la siguiente acción:

```
glEndList();
```

Un ejemplo sencillo compuesto por un único triángulo podría ser el siguiente:

```
glNewList(1, GL_COMPILE);
glBegin(GL_TRIANGLES);
    glColor3f(1.0, 1.0, 1.0);
    glVertex3f(0.0, 0.0, 0.0);
    glColor3f(1.0, 0.0, 0.0);
    glVertex3f(3.0, 5.0, 0.0);
    glColor3f(0.0, 1.0, 0.0);
    glVertex3f(-3.0, 5.0, 0.0);
glEnd();
glEndList();
```

Al utilizar el parámetro `GL_COMPILE`, no se produce salida gráfica alguna de momento.

Para activar la *display list*, y ejecutar sus comandos, es necesario realizar la llamada a la siguiente función con el identificador de ésta como parámetro:

```
glCallList(GLuint name)
```

Definir una secuencia de comandos con un identificador ya utilizado significa actualizar la versión anterior. Para evitar posibles errores, OpenGL puede proporcionarnos identificadores de *display lists* libres. Es necesario llamar a la siguiente función:

```
GLuint glGenLists( GLsizei range )
```

Para un único identificador indicaremos como parámetro el valor 1. Un valor superior devolverá un rango de identificadores consecutivos.

5.3.13. Creación de objetos jerárquicos

El uso de estas funciones que acabamos de comentar se hace más patente cuando queremos crear objetos con jerarquías. En este sentido, las listas de visualización pueden ser descritas en función de otras. Veamos el ejemplo práctico de un objeto bicicleta.

```
#define MANILLAR 1
```

```
#define CUADRO      2
#define RUEDA      3
#define BICICLETA  4

glNewList (MANILLAR, GL_COMPILE);
...
glEndList ();

glNewList (CUADRO, GL_COMPILE);
...
glEndList ();

glNewList (RUEDA, GL_COMPILE);
...
glEndList ();

glNewList (BICICLETA, GL_COMPILE);
    glCallList (MANILLAR);
    glCallList (CUADRO);
    glTranslatef (-1.0, 0.0, 0.0);
    glCallList (RUEDA);
    glTranslatef (2.0, 0.0, 0.0);
    glCallList (RUEDA);
glEndList ();
```

5.3.14. Texturas

A continuación, daremos a conocer las nociones y funcionalidades que OpenGL brinda sobre el apartado de texturas para su uso en videojuegos 2D, de manera que se acotará el volumen de información para dar respuesta tan sólo a este contexto.

Una textura es una imagen que se usa como superficie para un objeto tridimensional. Hasta el momento tan sólo podíamos dar un aspecto mediante colores; a partir de ahora podremos aplicar imágenes.

Para un programador, la utilización de texturas conlleva cuatro pasos a realizar:

- 1) Carga de imágenes como recurso físico que son a partir de un fichero y diferente según el formato del mismo: BMP, PNG, JPG, TGA, etc.
- 2) Cargar la textura con sus atributos diversos así como su propio contenido.
- 3) Definición de filtros de aplicación de texturas.
- 4) Utilización.

El primer paso de todos es independiente de OpenGL. Si bien éste nos permite la carga de ficheros formato BMP, resulta insuficiente, por lo que es necesario hacer uso de otras librerías para esta tarea. En nuestro caso utilizaremos y detallaremos más adelante la librería Corona.

Una vez obtenida la información o contenido de la textura, debemos proceder a su carga. Para ello será necesario llamar a estas dos funciones en el orden en que se presentan. La primera de ellas se usa tanto en la creación como en el uso de las texturas y es la siguiente:

```
void glBindTexture(GLenum target, GLuint texturesName);
```

Este método realiza tres tareas:

- Cuando se le da un nombre de textura distinto de 0 por primera vez, crea una textura nueva y le asigna este nombre.
- Cuando se llama después de la primera vez (que ya hemos creado la textura) se selecciona la textura con ese nombre como activa.
- Si el nombre de la textura es 0, OpenGL deja de utilizar objetos de textura y devuelve como activa la textura por defecto.

Al igual que vimos con las *display lists*, es posible obtener un identificador de textura válido con la siguiente función:

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

Una vez hemos indicado el identificador de la textura a definir, procedemos con su carga:

```
void glTexImage2D ( GLenum valor, GLint nivel,
                  GLint componentes,
                  GLsizei ancho, GLsizei alto,
                  GLint borde, GLenum formato,
                  GLenum tipo, const GLvoid *pixels)
```

Los posibles parámetros con los que trabajaremos son:

- Valor: `GL_TEXTURE_2D`.
- Nivel: Nivel de detalle. Se utiliza para texturas múltiples. Para texturas simples debe valer 0.
- Componentes: Número de componentes del color. Puede valer entre 1 y 4.

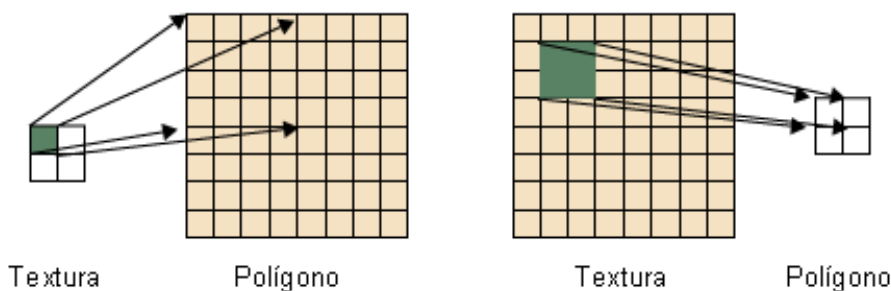
Tamaño de las texturas

Aunque las tarjetas gráficas más recientes no están sometidas a esta limitación, para que podamos realizar la carga de texturas en cualquier ordenador, éstas deberán ser de dimensión potencia de dos (128x128, 256x512, etc.).

- Ancho, alto: Dimensiones de la imagen. Para que cualquier tarjeta gráfica pueda operar con ésta, los dos valores deberán ser potencia de dos.
- Borde: Grosor del borde. Puede valer 0, 1 o 2 (suele ser nulo).
- Formato: Indica en qué formato se encuentra la imagen: GL_COLOR_INDEX, GL_LUMINANCE, GL_LUMINANCE_ALPHA, GL_RGB, GL_RGBA, GL_RED, GL_GREEN, GL_BLUE o GL_ALPHA.
- Tipo: Tipo de datos del *array* de pixels: GL_BYTE, GL_UNSIGNED_BYTE, GL_BITMAP, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT, GL_UNSIGNED_INT o GL_FLOAT.
- Píxeles: *Array* de píxeles que forman la imagen.

Las imágenes utilizadas para las texturas son cuadradas o rectangulares. En cambio, las primitivas sobre las que se aplican no tienen por qué serlo. Además, en general, el tamaño de éstas no suele coincidir. Por tanto, rara vez un píxel de la textura (llamado texel) se corresponde con un píxel de la escena. En estos casos hay que recurrir a algún filtro que obtenga el valor adecuado para cada píxel de la escena a partir de los texels de la textura.

Cuando no existe una correspondencia uno a uno entre píxeles de la escena y texels, pueden darse dos casos: que un píxel se corresponda con varios texels (minificación) o que se corresponda con una parte de un texel (magnificación). La siguiente imagen ilustra este hecho:



A la izquierda es necesario magnificar y a la derecha, minificar.

Es posible fijar filtros diferentes para cada una de estas dos situaciones utilizando el siguiente método:

```
void glTexParameteri ( GLenum tipo, GLenum valor,
                      GLint parametro)
```

Los parámetros son:

- Tipo: GL_TEXTURE_2D.

- Valor: Indica el parámetro que se está fijando. Puede valer `GL_TEXTURE_MIN_FILTER` o `GL_TEXTURE_MAG_FILTER`.
- Parámetro: Es el tipo de filtro. Veremos los casos `GL_NEAREST` y `GL_LINEAR`.

La opción `GL_NEAREST` elige como valor para un píxel el texel con coordenadas más cercanas al centro del píxel. Puede producir efectos de *aliasing*.

La opción `GL_LINEAR` elige como valor para un píxel el valor medio de la muestra de la matriz 2x2 de texels más cercano al centro del píxel. Produce efectos más suaves que `GL_NEAREST`.

Veamos dos imágenes resultantes de aplicar estos dos diferentes filtros. La diferencia de calidad es notoria.



Lara Croft del videojuego Tomb Raider. A la izquierda, con filtro de textura `GL_NEAREST`; a la derecha, con filtro de textura `GL_LINEAR`

Veamos ahora otra ilustración donde podemos observar el *aliasing* que se produce tanto a la hora de magnificar como de reducir si el filtro utilizado es `GL_NEAREST`, el más simple de todos:



Primera versión del popular juego Doom

Otro aspecto fundamental a la hora de manejar las texturas es conocer cómo se aplican las texturas a las primitivas y si sufren alguna operación adicional. Existen tres formas de aplicar las texturas y lo podemos indicar mediante la siguiente función:

```
glTexEnvf (GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GLint valor)
```

El parámetro valor puede ser:

- **GL_DECAL:** Cuyo efecto es colocar la textura sobre la primitiva, el valor del texel se copia directamente al píxel (textura opaca).
- **GL_MODULATE:** Permite modular o matizar el color de la primitiva con los colores de la textura.
- **GL_BLEND:** El valor del texel se usa para interpolar entre el color del objeto y un color constante definido para la textura mediante la función siguiente:

```
glTexEnvfv(GL_TEXTURE_ENV, GL_TEXTURE_ENV_COLOR, GLfloat color[4])
```

La última cuestión que afecta a los modos de aplicación de las texturas es la posibilidad de expandirlas o repetirlas. Cuando la textura es menor que la primitiva sobre la que debe colocarse, existen dos posibilidades: repetirla en forma de mosaico o expandirla más allá del final de la textura utilizando su propio borde. La textura puede repetirse en un sentido y expandirse en otro, o realizar la misma operación en ambas direcciones. Utilizaremos la función siguiente:

```
void glTexParameterf (GLenum tipo, GLenum valor, GLfloat parametro)
```

Los parámetros son:

- Tipo: GL_TEXTURE_2D.
- Valor: Indica el sentido al que afecta el modo de repetición. Puede valer GL_TEXTURE_WRAP_S (coordenada S) o GL_TEXTURE_WRAP_T (coordenada T). Las coordenadas de textura las trataremos en el siguiente apartado.
- Parametro: Es el tipo de repetición. Puede valer GL_REPEAT (repetición en modo mosaico) o GL_CLAMP (repetición en modo expandir).

La repetición GL_REPEAT se realiza colocando la textura en modo de mosaico. Los texels del borde se calculan como la media con los del mosaico adyacente. En el modo GL_CLAMP, los píxeles del borde se expanden en toda la primitiva. Veamos un ejemplo ilustrativo de estos modos:



Una vez que las texturas han sido creadas y se ha determinado el modo de aplicación de las mismas, deben ser activadas para que se apliquen a las primitivas:

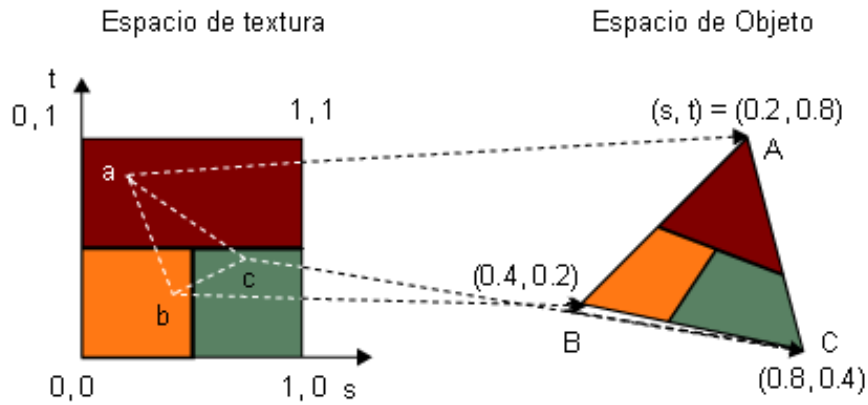
```
glEnable (GL_TEXTURE_2D)
```

Para poder usar la textura, utilizamos el procedimiento anteriormente nombrado glBindTexture() y seleccionamos la textura actual. Además debemos especificar las coordenadas de texturas, y lo hacemos por medio de:

```
void glTexCoord2f(TYPE coords);
void glTexCoord2fv(TYPE *coords);
```

Establece las coordenadas actuales de la textura (s, t). Debe preceder a glVertex*.

Veamos un ejemplo de definición de coordenadas de textura:



Los valores de coordenadas de textura situadas en el intervalo $(0.0, 1.0)$ denotan el mapeo parcial de una textura sobre un polígono. Esta opción resultará especialmente útil cuando queremos obtener un *frame* de una animación como ocurre en el caso siguiente:



"Marine del Doom"

En cambio, si utilizamos el intervalo (i, j) , haremos que este mapeo se efectúe i veces repetido sobre el eje x , y j veces sobre el eje y .

Veamos un ejemplo práctico en el cual podemos apreciar claramente el uso de texturas mapeadas con un factor de repetición:



"Quake III Arena"

5.4. DirectX

Como hemos comentado con anterioridad, DirectX es dependiente de la plataforma Microsoft Windows. Por esta razón, si queremos programar con esta API, primero debemos tratar el paradigma de la programación en Windows.

5.4.1. Programación en Windows

La programación en Windows es un tema complicado del cual hay muchos libros escritos. Dado que no es nuestro principal objetivo, mostraremos su funcionamiento directamente con un ejemplo que podemos considerar plantilla.

Toda aplicación Windows ha de tener dos funciones: WinMain y WinProc (la segunda realmente podemos llamarla como queramos).

- WinMain: Es la que se ejecuta al arrancar la aplicación. En nuestro ejemplo se encarga de crear una ventana sobre la cual se ejecutará la aplicación y el *event loop*, que simplemente recoge los eventos de Windows y se los pasa a WinProc, para que éste los gestione.
- WinProc: Es la encargada de gestionar los diferentes eventos, por ejemplo, de cerrar la aplicación si recibimos un mensaje WM_DESTROY. Al crear la ventana en la función WinMain, le hemos indicado que la función que ha de llamar cuando tenga un mensaje sea ésta.

A continuación, mostramos el código necesario para tener una aplicación bien sencilla para Windows. Ésta consistirá en una única ventana que tan sólo esperará la presión de la tecla ESC para salir de ella. Fijémonos con atención dónde colocamos nuestra función GameMain.

```
HWND      g_hMainWnd;
HINSTANCE g_hInst;

HWND      InitWindow(int iCmdShow);
LRESULT   CALLBACK WinProc(HWND hWnd, UINT message, WPARAM wParam,LPARAM lParam);
void GameMain();
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine,int nCmdShow)
{
    g_hInst = hInstance;
    g_hMainWnd = InitWindow(nCmdShow);

    if(!g_hMainWnd) return -1;

    while( TRUE )
    {
        MSG msg;

        if( PeekMessage( &msg, NULL, 0, 0, PM_REMOVE ) )
        {
            // Check for a quit message
            if( msg.message == WM_QUIT ) break;

            TranslateMessage( &msg );
            DispatchMessage( &msg );
        }
        else
        {
            GameMain();
        }
    }
    return 0;
}

void GameMain()
{
    //El main de nuestro juego
}

HWND InitWindow(int iCmdShow)
{
    HWND      hWnd;
    WNDCLASS  wc;
```

```
    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = WinProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = g_hInst;
    wc.hIcon = LoadIcon(g_hInst, IDI_APPLICATION);
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = (HBRUSH) GetStockObject (BLACK_BRUSH);
    wc.lpszMenuName = NULL;
    wc.lpszClassName = "MyGame";
    RegisterClass(&wc);

    hWnd = CreateWindowEx(NULL, "MyGame", "MyGame",
                          WS_POPUP, 0, 0, 640, 480,
                          NULL, NULL, g_hInst, NULL);

    ShowWindow(hWnd, iCmdShow);
    UpdateWindow(hWnd);
    SetFocus(hWnd);

    return hWnd;
}

LRESULT CALLBACK WinProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
        {
            case WM_KEYDOWN: if(wParam == VK_ESCAPE)
                {
                    PostQuitMessage(0);
                    return 0;
                }
            break;

            case WM_DESTROY: PostQuitMessage(0);
                return 0;
        }
    }
    return DefWindowProc(hWnd, message, wParam, lParam);
}
```

Es importante observar en este código los siguientes puntos:

- El nombre de la función que tratará los mensajes de Windows lo hemos declarado en uno de los atributos de la clase WNDCLASS, al hacer:

```
wc.lpfWndProc = WinProc;
```

- En la función que crea la ventana de la aplicación (InitWindow), al hacer la llamada CreateWindowEx, son importantes los siguientes atributos:
 - Para crear una ventana sin marcos:
WS_POPUP
 - Para crear una ventana con marcos típica de Windows:
WS_OVERLAPPEDWINDOW | WS_VISIBLE
 - Tamaño de la ventana que nosotros queramos:
640, 480
 - Tamaño de la ventana que sea la pantalla completa:
GetSystemMetrics(SM_CXSCREEN)
GetSystemMetrics(SM_CYSCREEN)

5.4.2. DirectX Graphics

DirectX Graphics es el componente principal de DirectX, tanto para la realización de videojuegos 2D como 3D.

Antiguamente, la programación de videojuegos en dos dimensiones implicaba la utilización del paquete DirectDraw. Hoy en día, aunque se puede seguir programando con él, se encuentra desfasada y se utiliza el modo 3D para la realización de juegos en 2D, por muy sencillos que sean. De hecho, DirectX Graphics no es otra cosa que la combinación de los paquetes DirectDraw y Direct3D.

Para programar un videojuego en 2D utilizando la API DirectX, existen dos técnicas básicas:

- La primera, consiste en definir una cámara paralela y ubicar todos los elementos que participan en el videojuego, en el plano $z = 0$.
- La segunda alternativa es más sencilla (es la que utilizaremos) y se basa en la utilización de la interfaz ID3DXSprite.

Veamos, a continuación, las variables y estructuras necesarias así como los diferentes pasos a seguir para lograr el propósito marcado.

Estructuras necesarias

```
// Objeto principal Direct3D  
LPDIRECT3D9 g_pD3D;  
  
// Dispositivo
```

```

LPDIRECT3DDEVICE9 g_pD3DDevice;
// Objeto principal Sprite
LPD3DXSPRITE g_pSprite;
// Textura para un sprite
LPDIRECT3DTEXTURE9 texture;

```

Inicializar Direct3D

```

// Crea un objeto IDirect3D9 y devuelve una interfaz
g_pD3D = Direct3DCreate9( D3D_SDK_VERSION );
if(g_pD3D==NULL) return -1;

// Describe los parámetros de presentación
D3DPRESENT_PARAMETERS d3dpp;
ZeroMemory( &d3dpp, sizeof( d3dpp ) );

d3dpp.Windowed          = FALSE;
d3dpp.SwapEffect        = D3DSWAPEFFECT_DISCARD;
d3dpp.BackBufferWidth  = SCREEN_RES_X;
d3dpp.BackBufferHeight = SCREEN_RES_Y;
d3dpp.BackBufferFormat = D3DFMT_X8R8G8B8;

HRESULT hr;
hr = g_pD3D->CreateDevice( D3DADAPTER_DEFAULT,
                          D3DDEVTYPE_HAL, hWnd,
                          D3DCREATE_SOFTWARE_VERTEXPROCESSING,
                          d3dpp, &g_pD3DDevice );
if(FAILED(hr)) return -1;

```

Inicializar Sprites

```

// Crea un objeto sprite, asociado a un dispositivo
D3DXCreateSprite( g_pD3DDevice, &g_pSprite );

// Crea una textura desde un archivo.
D3DXCreateTextureFromFileEx( g_pD3DDevice, "main.png",
                             0, 0, 1, 0,
                             D3DFMT_UNKNOWN,
                             D3DPOOL_DEFAULT,
                             D3DX_FILTER_NONE,
                             D3DX_FILTER_NONE,
                             NULL, NULL, NULL, &texture );

```

Este último método es fundamental.

Web

Para un estudio más exhaustivo, se recomienda una visita al enlace siguiente y realizar una búsqueda por el nombre del método: <http://msdn.microsoft.com>

Veamos con detalle la especificación de los parámetros más importantes; cada uno de éstos acepta diferentes posibilidades:

```
HRESULT WINAPI D3DXCreateTextureFromFileEx(  
  
    LPDIRECT3DDEVICE9 pDevice, // Dispositivo asociado  
    LPCTSTR pSrcFile,        // Nombre del archivo  
    UINT Width,              // Anchura y altura de la imagen  
    UINT Height,             // (0: las coge él solo)  
    UINT MipLevels,          // 1  
    DWORD Usage,             // 0  
    D3DFORMAT Format,        // D3DFMT_UNKNOWN  
    D3DPOOL Pool,            // D3DPOOL_DEFAULT  
    DWORD Filter,            // D3DX_FILTER_NONE  
    DWORD MipFilter,         // D3DX_FILTER_NONE  
    D3DCOLOR ColorKey,       // Color keying: 0x00ff00ff (rosa)  
    D3DXIMAGE_INFO *pSrcInfo, // NULL  
    PALETTEENTRY *pPalette,   // NULL  
    LPDIRECT3DTEXTURE9 *ppTexture // Objeto textura a crear  
);
```

Comentaremos los parámetros importantes acotando por relevancia, teniendo en cuenta que queremos realizar un videojuego 2D y prefijando aquellos que no tengan mayor importancia:

- pDevice: Dispositivo asociado a la textura
- pSrcFile: Nombre del archivo
- Width, Height: Anchura y altura de la imagen (con el valor cero las coge él solo)
- MipLevels: 1
- Usage: 0
- Format: D3DFMT_UNKNOWN
- Pool: D3DPOOL_DEFAULT
- Filter: D3DX_FILTER_NONE
- MipFilter: D3DX_FILTER_NONE
- ColorKey: Color keying, por ejemplo para indicar el rosa introduciríamos el valor 0x00ff00ff
- pSrcInfo: NULL
- pPalette: NULL
- ppTexture: Objeto textura a crear

Renderizado y visualización

```
// Limpiamos la pantalla con el color negro
```

```

g_pD3DDevice->Clear( 0, NULL, D3DCLEAR_TARGET,
                    0xFF000000, 0, 0 );
// Iniciamos el renderizado de la escena
g_pD3DDevice->BeginScene();

// Iniciamos el renderizado de sprites
g_pSprite->Begin( D3DXSPRITE_ALPHABLEND );

    g_pSprite->Draw( texMain, NULL, NULL, &D3DXVECTOR3(0.0f,0.0f,0.0f), 0xFFFFFFFF);

// Finalizamos el renderizado de sprites
g_pSprite->End();

// Finalizamos el renderizado de la escena
g_pD3DDevice->EndScene();

// Visualización
g_pD3DDevice->Present( NULL, NULL, NULL, NULL );

```

Nuevamente, comentaremos las líneas de código más importantes, que son las del pintado de *sprites*.

```

HRESULT Draw(

    LPDIRECT3DTEXTURE9 pTexture,
    CONST RECT *pSrcRect,
    CONST D3DXVECTOR3 *pCenter,
    CONST D3DXVECTOR3 *pPosition,
    D3DCOLOR Color

);

```

Los parámetros son:

- pTexture: Textura del *sprite*
- pSrcRect: Rectángulo deseado (NULL para completo dibujado)
- pCenter: Centro del *sprite* (NULL para valor (0,0,0))
- pPosition: Posición donde pintar (NULL para valor (0,0,0))
- Color: Color y canal alfa para modular (0xFFFFFFFF para no alterar)

Finalización

Al finalizar la aplicación, es necesario liberar todos los recursos solicitados. Por lo que respecta a los objetos de la API DirectX Graphics tratados, el código para nuestro ejemplo sería el siguiente:

```

if(texture)
{

```

```

    texture->Release();
    texture = NULL;
}
if(g_pSprite)
{
    g_pSprite->Release();
    g_pSprite = NULL;
}
if(g_pD3DDevice)
{
    g_pD3DDevice->Release();
    g_pD3DDevice = NULL;
}
if(g_pD3D)
{
    g_pD3D->Release();
    g_pD3D = NULL;
}

```

5.4.3. DirectInput

DirectInput es la parte de las DirectX encargada de la captura de eventos de entrada, ya sea desde el teclado, el ratón o un *joystick*. Este proceso a diferencia de Direct3D, es bastante más sencillo. Veámoslo en tres partes: estructuras y variables necesarias, inicialización y lectura.

Estructuras necesarias

```

LPDIRECTINPUT8 m_pDI;           // Objeto que inicia Direct Input

LPDIRECTINPUTDEVICE8 Raton;     // Este será el ratón
DIMOUSESTATE EstadoRaton;      // Estado del ratón
int PosXRaton, PosYRaton;       // Posición X,Y en pantalla del ratón

LPDIRECTINPUTDEVICE8 Teclado;   // Este es el teclado
UCHAR EstadoTeclado[256];       // Estado de las teclas

```

Inicialización

Para inicializar DirectInput, primero es necesario crear el propio objeto y luego, sucesivamente para cada dispositivo de entrada, efectuar las siguientes operaciones:

- Crear el dispositivo
- Iniciar el nivel de cooperación
- Iniciar el modelo de datos

- Adquirir el dispositivo para la aplicación

Veamos la implementación de la inicialización de DirectInput para una aplicación que fuera interaccionable mediante teclado y ratón:

```
HRESULT hr;
// Crear DirectInput
hr = DirectInput8Create( hInst, DIRECTINPUT_VERSION, IID_IDirectInput8, (void **)&m_pDI, NULL);
if(FAILED(hr)) return -1;
// 1. Crea el teclado
hr = lpDI->CreateDevice( GUID_SysKeyboard, &Teclado, NULL);
if(FAILED(hr)) return -1;
// 2. Inicia el nivel de cooperacion
hr = Teclado->SetCooperativeLevel(hWnd, DISCL_FOREGROUND | DISCL_NONEXCLUSIVE);
if(FAILED(hr)) return -1;
// 3. Inicia el modelo de datos
hr = Teclado->SetDataFormat(&c_dfDIKeyboard);
if(FAILED(hr)) return -1;
// 4. Adquiere el teclado para la aplicacion
hr = Teclado->Acquire();
if(FAILED(hr)) return -1;
// 1. Crear el ratón
hr = lpDI->CreateDevice(GUID_SysMouse, &Raton, NULL);
if(FAILED(hr)) return -1;
// 2. Inicia el nivel de cooperacion
hr = Raton->SetCooperativeLevel(hWnd, DISCL_FOREGROUND | DISCL_EXCLUSIVE);
if(FAILED(hr)) return -1;
// 3. Inicia el modelo de datos
hr = Raton->SetDataFormat(&c_dfDIMouse);
if(FAILED(hr)) return -1;
// 4. Adquiere el teclado para la aplicacion
hr = Raton->Acquire();
if(FAILED(hr)) return -1;
```

Lectura

Para realizar la lectura de un dispositivo y comprobar si ha tenido lugar un evento de entrada a partir de éste, se utiliza el método `GetDeviceState`. Siguiendo con el ejemplo anterior procederíamos de la manera siguiente:

```
// Leer teclado
Teclado->GetDeviceState(256, EstadoTeclado);
```

```
// Leer raton
Raton->GetDeviceState(sizeof(DIMOUSESTATE), &EstadoRaton);
```

1) Lectura del estado del teclado

Estas dos funciones recogen el estado y lo guardan en sus respectivas variables.

Para comprobar si se ha pulsado una tecla podemos usar:

```
if (EstadoTeclado[Codigo_Tecla] > 0) ...
```

Donde los códigos de tecla con qué trabaja DirectX están definidos según la sintaxis: DIK_F1, DIK_F2, DIK_UP, DIK_LEFT, DIK_RETURN, DIK_SPACE, DIK_A, DIK_B, DIK_0, DIK_1, DIK_NUMPAD0, DIK_NUMPAD1, etc. Es decir, "DIK" (*direct input keyboard*) y la tecla correspondiente.

2) Lectura del estado del ratón

Para comprobar el estado del ratón, podemos usar:

```
EstadoRaton.lX // Desplazamiento de las X
EstadoRaton.lY // Desplazamiento de las Y
EstadoRaton.lZ // Desplazamiento de las Z
EstadoRaton.rgbButtons[4] // Indica el estado de los botones
```

Debemos tener en cuenta que las variables lX, lY, lZ sólo indican el desplazamiento desde el estado anterior, y no la posición absoluta en pantalla. Por eso, para actualizar las variables del ratón, debemos hacer algo como:

```
Mouse.x = Mouse.x + EstadoRaton.lX;
Mouse.y = Mouse.y + EstadoRaton.lY;
```

El campo lZ se refiere al desplazamiento efectuado con la rueda del ratón.

Finalización

Al finalizar la aplicación, es necesario liberar todos los recursos solicitados. Por lo que respecta a los objetos de la API DirectInput tratados, el código para nuestro ejemplo sería el siguiente:

```
if (Teclado)
{
    Teclado->Unacquire();
    Teclado->Release();
    Teclado = NULL;
}
if (Raton)
{
    Raton->Unacquire();
    Raton->Release();
    Raton = NULL;
}
```

```
}  
if (m_pDI)  
{  
    m_pDI->Release();  
    m_pDI = NULL;  
}
```