Alternativas libres para creación de videojuegos

Francisco Javier Tapia Merino
D.N.I.: 45886356-E
E-mail: i22tamef@uco.es
Titulación: Grado en Ingeniería Informática
Escuela Politécnica Superior
Universidad de Córdoba

Índice de contenido

Introducción:

Numerosos motores son actualmente conocidos en la industria del videojuego que permiten el desarrollo de videojuegos de forma sencilla y estructurada con poco esfuerzo. Unity3D, Unreal Engine, CryEngine, GameMaker, etc. Todos ellos, aunque muy completos y sencillos de usar, son privativos, y no tienen el código abierto, lo cual los hace poco recomendables para que estudiantes que desean introducirse en este mundo aprendan las bases sobre las que se estructuran estos. Además algunos de ellos imponen condiciones al usuario respecto a la distribución de los productos resultantes.

El presente documento pretende introducir algunas herramientas de desarrollo orientadas a la industria del videojuego y de cualquier forma de contenido gráfico que permitan al usuario completo acceso y control sobre sus funciones y la distribución de los productos resultantes.

El uso o no del uso de las herramientas libres para producir contenido profesional destinado al mercado es altamente dependiente del usuario y de los recursos disponibles. Aunque completas y capaces de desarrollar contenido profesional y funcional estas herramientas suelen ser de un mas bajo nivel que las privativas comúnmente usadas y por tanto, para ciertos usuarios la inversión extra de tiempo y esfuerzo que requieren puede no ser viable. Sin embargo las creemos muy recomendables desde el punto de vista académico.

Contenidos:

El presente documento pretende suponer una breve introducción a las tecnologías libres mas comunes, presentar estas y dar un breve tutorial de su uso mas básico. En ningún caso se pretende que el lector salga dominando ninguna de estas, si no que se pretende darlas a conoces lo suficiente como para que su posterior estudio sea lo mas cómodo posible.

Nos centraremos en librerías orientadas a gráficos 2D como son SDL y pygame, por ser mas sencillas y accesibles para un publico con poca o ninguna experiencia en gráficos por computador. Se mencionarán brevemente algunos recursos para el desarrollo de programas 3D, por último se mencionará una librería enfocada únicamente a simular comportamientos físicos, sin considerar los gráficos.

Aparte de las librerías tratadas en este documento existen otras muchas igualmente válidas y para multitud de plataformas y lenguajes. Algunas son:

- Allegro (C)
- Cocos2D, Cocos2D-x, Cocos2D-html (C++, python, javascript)
- HaxeFlixel (Haxe)
- LibGDX (java)
- Ogre3D
- SFML (C++)

OpenGL:

OpenGL representa el estándar en la industria para el desarrollo de gráficos por computador tanto para 2D como 3D. Originariamente en C, aunque portada actualmente a numerosos lenguajes y plataformas, OpenGL permite la generación de gráficos de forma altamente eficiente, y controlada por el usuario, con capacidad de interacción hardware a bajo nivel para sacar el máximo partido de este.

OpenGL se distribuye para desarrolladores bajo una licencia de software libre B, muy similar a la BSD, X y Mozilla. Sin embargo, las empresas que desean desarrollar hardware compatible con OpenGL deben comprar un licencia privativa.

Se ha optado por no incluir un tutorial de OpenGL en este documento. Debido a la gran profundidad y complejidad de OpenGL nos arriesgamos a introducir malas prácticas intentado hacer una explicación resumida que pueda estar con tenida en este documento. En su lugar se recomienda comenzar usando las otras librerías que mencionamos en este documento. Ambas librerías, SDL y pygame, se basan en OpenGL limitando el acceso a bajo nivel y simplificando su uso de cara al usuario novel.

SDL:

Introducción:

Simple DirectMedia Layer es una librería de desarrollo multiplataforma diseñada para proveer de acceso de bajo nivel a audio, teclado, ratón joystick y hardware gráfico por medio de OpenGL y Direct3D.

SDL está escrito en C y permite su uso en C++, además existen numerosas adaptaciones para otros lenguajes.

SDL 2.0 se distribuye bajo licencia zlib, la cual permite su libre redistribución tanto de forma libre como comercial.

Tutorial:

Inicializar ventana:

Lo primero que necesitamos hacer antes de poder usar SDL es crear una ventana en la que mostrar nuestro trabajo. Para crear una ventana se van a usar las siguientes funciones y estructuras:

SDL_Window : Tipo de dato que almacena la ventana donde se mostrarán las imágenes. Se declara un puntero a esta, ya que las funciones de SDL requerirán lo usarán de esta manera.

int SDL_Init(Uint32 flags) : Función que inicializa los sistemas de SDL. Es necesario invocar a esta función para usar cualquier elemento de SDL. La función devuelve 0 si se ha inicializado correctamente o -1 si hay algún error.

La función admite los siguientes parámetros:

`SDL_INIT_TIMER`	Subsistema de gestión de tiempo
`SDL_INIT_AUDIO`	Subsistema de audio
`SDL_INIT_VIDEO`	Subsistema de vídeo
`SDL_INIT_CDROM`	Subsistema de CDROM
`SDL_INIT_JOYSTICK`	Subsistema de Joystick
`SDL_INIT_EVERYTHING`	Todos los subsistemas
`SDL_INIT_NOPARACHUTE`	Evita que SDL capture señales de terminación
`SDL_INIT_EVENTTHREAD`	Arranca el sistema de eventos en un hilo separado.

SDL_Window* SDL_CreateWindow(const char* title, int x, int y, int w, int h, Uint32 flags): Crea la ventana para mostrar las imagenes que usaremos posteriormente. Recibe:

title	Cadena con el título de la ventana en UTF-8.
x	Posición x de la ventana, SDL_WINDOWPOS_CENTERED, o SDL_WINDOWPOS_UNDEFINED.
y	Posición y de la ventana, SDL_WINDOWPOS_CENTERED, o SDL_WINDOWPOS_UNDEFINED.
w	Ancho de la ventana.
h	Alto de la ventana.
flags	0 o más flags.

Flags disponibles:

DL_WINDOW_FULLSCREEN	Pantalla completa.
SDL_WINDOW_FULLSCREEN_DESKTOP	Pantalla completa a resolución actual del escritorio.
SDL_WINDOW_OPENGL	Ventana compatible con Open_GL.
SDL_WINDOW_HIDDEN	Ventana no visible.
SDL_WINDOW_BORDERLESS	Ventana sin bordes.
SDL_WINDOW_RESIZABLE	Ventana que permite reescalado.
SDL_WINDOW_MINIMIZED	Ventana minimizada.
SDL_WINDOW_MAXIMIZED	Ventana maximizada.
SDL_WINDOW_INPUT_GRABBED	Ventana con el focus.
SDL_WINDOW_ALLOW_HIGHDPI	Ventana creada en high-DPI si se soporta. (>= SDL 2.0.1)

Una vez que conocemos las funciones podemos proceder a crear nuestro primer programa con SDL.

```
int main(int argc, char** argv){
   SDL_Window* window = NULL;
```

```
    if( SDL_Init( SDL_INIT_VIDEO ) < 0 )
    {
        printf( "SDL could not initialize!”);
        exit(-1);
    }
    window = SDL_CreateWindow( "SDL Tutorial", SDL_WINDOWPOS_UNDEFINED,
SDL_WINDOWPOS_UNDEFINED, SCREEN_WIDTH, SCREEN_HEIGHT,
SDL_WINDOW_SHOWN );
    if( window == NULL )
    {
        printf( "Window could not be created!");
        exit(-1);
    }
    return 0;
}
```

El código mostrado genera una ventana y una superficie asociada. Sin embargo no somos capaces de verlo ya que la ejecución termina demasiado rápido.
Normalmente generaremos la ventana y la mostraremos y actualizaremos dentro de un bucle infinito hasta recibir la orden de cerrado. Ahora mismo haremos uso de la función SDL_Delay() para darnos tiempo a ver la ventana. Además añadiremos las funciones de destrucción para asegurarnos que terminamos la ejecución correctamente.

```
int main(int argc, char** argv){
    SDL_Window* window = NULL;
    if( SDL_Init( SDL_INIT_VIDEO ) < 0 )
    {
        printf( "SDL could not initialize! ");
        exit(-1);
    }

    window = SDL_CreateWindow( "SDL Tutorial", SDL_WINDOWPOS_UNDEFINED,
SDL_WINDOWPOS_UNDEFINED, SCREEN_WIDTH, SCREEN_HEIGHT,
SDL_WINDOW_SHOWN );
    if( window == NULL )
    {
        printf( "Window could not be created! " );
        exit(-1);
    }

    SDL_Delay( 2000 );// Espera 2 segundos (2000 milisegundos)
    SDL_DestroyWindow( window ); // Libera la ventana
    SDL_Quit(); // Cierra SDL
    return 0;
}
```

Con esto tendremos un programa que crea una ventana durante 2 segundos. Sin embargo aun no estamos dibujando nada en la superficie.
Pasemos entonces al dibujado en pantalla.

Dibujar en pantalla:

Una vez que tenemos una ventana podemos proceder a dibujar sobre ella. La funciones que usaremos para el dibujado son las siguientes:

SDL_Renderer: Estructura que gestiona el renderizado sobre una ventana.

SDL_Renderer* SDL_CreateRenderer(SDL_Window* window, int index, Uint32 flags): Inicializa el renderer a la ventana pasada como argumento. Index representa el índice del driver a inicializar. Con -1 se inicializa el primero compatible con los flags. Los flags disponibles son los siguientes:

SDL_RENDERER_SOFTWARE	El renderer es un software fallback
SDL_RENDERER_ACCELERATED	El renderer usa aceleración hardware.
SDL_RENDERER_PRESENTVSYNC	Sincronizado con la velocidad de refresco.
SDL_RENDERER_TARGETTEXTURE	Soporte para renderizado de texturas.

SDL_Rect: Estructura de datos que representa un área rectangular.

int SDL_SetRenderDrawColor(SDL_Renderer* renderer, Uint8 r, Uint8 g, Uint8 b, Uint8 a): Establece el color de borrado para la pantalla asociada al renderer en rgb y alfa.

int SDL_RenderClear(SDL_Renderer* renderer): Borra la pantalla con el color especificado.

int SDL_SetRenderDrawColor(SDL_Renderer* renderer, Uint8 r, Uint8 g, Uint8 b, Uint8 a): Establece el color de dibujado de las primitivas de SDL.

int SDL_RenderFillRect(SDL_Renderer* renderer, const SDL_Rect* rect): Dibuja el rectángulo pasado.

void SDL_RenderPresent(SDL_Renderer* renderer): Actualiza la ventana con los cambios efectuados por el renderer.

Con estas funciones y estructuras básicas podemos comenzar a dibujar elementos en pantalla de la siguiente forma:

```
int main(int argc, char** argv){
   SDL_Window* window = NULL;
   SDL_Rect rect;
   if( SDL_Init( SDL_INIT_VIDEO ) < 0 )
   {
      printf( "SDL could not initialize!" );
      exit(-1);
   }

   window = SDL_CreateWindow( "SDL Tutorial", SDL_WINDOWPOS_UNDEFINED,
SDL_WINDOWPOS_UNDEFINED, SCREEN_WIDTH, SCREEN_HEIGHT,
```

```
SDL_WINDOW_SHOWN );
   if( window == NULL )
   {
      printf( "Window could not be created! ");
      exit(-1);
   }

   rect.x = 0;
   rect.y = 0;
   rect.w = 32;
   rect.h = 32;

   SDL_Renderer* renderer = SDL_CreateRenderer(window, -1, 0);
   SDL_SetRenderDrawColor(renderer, 255, 255, 255, 255);
   SDL_RenderClear(renderer);

   SDL_SetRenderDrawColor(renderer, 255, 0, 0, 255);
   SDL_RenderFillRect(renderer, &rect);
   SDL_RenderPresent(renderer);
   SDL_Delay( 2000 );
   SDL_DestroyWindow( window );
   SDL_Quit();
   return 0;
}
```

Con esta breve introducción ya seríamos capaces de comenzar a hacer uso de SDL, sin embargo a la hora de realizar proyectos serios es muy improbable que queramos usar las primitivas de SDL para el dibujado. Por tanto introduciremos algunas funciones y estructuras mas para poder cargar imágenes y mostrarlas en nuestro programa.

SDL_Surface: Estructura que gestiona imágenes. Eficiente para la carga.

SDL_Textrure: Estructura que gestiona imágenes incorporada en la versión 2.0. Compatible con renderer pero mas lenta de cargar.

SDL_Surface* SDL_LoadBMP(const char* file): Lee un fichero de imagen y genera una surface con los datos de esta.

SDL_Texture* SDL_CreateTextureFromSurface(SDL_Renderer* renderer, SDL_Surface* surface): Convierte una surface en un texture compatible con el renderer.

int SDL_RenderCopy(SDL_Renderer* renderer, SDL_Texture* texture, const SDL_Rect* srcrect, const SDL_Rect* dstrect): Dibuja en la ventana la textura pasada. Srcrect representa el rect a tomar de la imagen origen, Null para toda la imagen. Dstrect representa el rect de la textura destino, null para toda la textura (se estirará la imagen para ajustar).

De esta forma podemos cargar una imagen que hemos creado con algún programa de dibujo externo:

```
int main(int argc, char** argv){
   SDL_Window* window = NULL;
   SDL_Rect rect;
```

```
    SDL_Surface* surf;
    SDL_Texture* text;

   SDL_Init( SDL_INIT_VIDEO )
   window = SDL_CreateWindow( "SDL Tutorial", SDL_WINDOWPOS_UNDEFINED,
SDL_WINDOWPOS_UNDEFINED, SCREEN_WIDTH, SCREEN_HEIGHT,
SDL_WINDOW_SHOWN );

    SDL_Renderer* renderer = SDL_CreateRenderer(window, -1, 0);

    surf=SDL_LoadBMP("helloworld.bmp");
    text=SDL_CreateTextureFromSurface(renderer,surf);
    SDL_FreeSurface(surf);
    surf = NULL;

   rect.x = 100;
   rect.y = 100;
   rect.w = 64;
   rect.h = 64;

   SDL_SetRenderDrawColor(renderer, 255, 255, 255, 255);
   SDL_RenderClear(renderer);
   SDL_SetRenderDrawColor(renderer, 255, 0, 0, 255);
   SDL_RenderCopy(renderer,text,NULL,&rect);
   SDL_RenderPresent(renderer);
   SDL_Delay( 2000 );
   SDL_DestroyWindow( window );
   SDL_Quit();
   return 0;
}
```

Para la carga de imágenes en otros formatos existe una librería extra de SDL que hay que descargar, incluir y enlazar contra ella, llamada SDL_Image.

Con lo visto se cubren todos las funciones de uso básico de SDL para poder comenzar a realizar programas. Sin embargo SDL es una librería compleja y sería imposible cubrirla completamente, por tanto se recomienda al usuario a comenzar por lo que se ha mostrado aquí y consultar la documentación oficial.

Pygame:

Introducción:

Pygame es un conjunto de módulos diseñados para la creación de juegos. Pygame añade funcionalidad sobre SDL. Esto permite la creación de juegos y contenido multimedia en lenguaje Python. Pygame es altamente portable y puede ejecutarse en casi cualquier plataforma y sistema operativo.

Pygame se distribuye bajo licencia LGPL 2.1, por tanto permite la distribución, tanto libre como comercial, del contenido creado con ella.

Tutorial:

Inicializar ventana:

pygame: Módulo mas alto de la jerarquía de Pygame.

pygame.init(): Inicializa todos los sistemas de pygame importados.

pygame.quit(): Cierra todos los sistemas de pygame.

pygame.display: Modulo que gestiona la ventana de visualización.

pygame.display.set_mode(resolution=(0,0), flags=0, depth=0): Función que crea una ventana de visualización y devuelve su superficie asociada. Debemos pasarle la resolución en píxeles, las banderas seleccionadas y la profundidad del color. Generalmente es mejor no pasar la profundidad ya que pygame selecciona la mas adecuada según los modos de inicio seleccionados. Las banderas disponibles son las siguientes:

pygame.FULLSCREEN	Ventana en pantalla completa
pygame.DOUBLEBUF	Configuración recomendada si se usa HWSURFACE u OPENGL
pygame.HWSURFACE	Aceleración hardware, solo para pantalla completa.
pygame.OPENGL	Ventana compatible con OpenGL
pygame.RESIZABLE	Permite redimensionar la ventana
pygame.NOFRAME	Ventana sin bordes o controles.

pygame.display.set_caption(title, icontitle=None): Establece el título de la ventana creada al especificado y permite asignarle un icono.

pygame.time: Módulo que gestiona el control del tiempo.

pygame.time.delay(): Para la ejecución el tiempo indicado (en milisegundos)

Con estas funciones ya podemos crear nuestro primer programa.

```
pygame.init();
screen=pygame.display.set_mode((640,460))
```

pygame.display.set_caption("Pygame tutorial")

pygame.time.delay(2000)

pygame.quit()

Con esto hemos creado nuestra primera ventana con pygame. Pasemos a dibujar cosas sobre ella.

Dibujar en pantalla:

pygame.draw: Módulo de pygame para dibujar figuras simples.

pygame.rect: Objeto para guardar coordenadas de un rectángulo.

pygame.draw.rect(Surface, color, Rect, width=0): Función apra dibujar rectángulos. Surface representa la superficie sobre la que dibujamos el rectángulo. Color es el color en una tupla en forma (r,g,b). Rect es el rectángulo a dibujar. Width es el ancho de linea, si no se especifica se rellena el cuadrado.

*Otras funciones de dibujado:

- **pygame.draw.polygon**: Dibuja forma con cualquier número de lados.
- **pygame.draw.circle**: Dibuja un circulo.
- **pygame.draw.ellipse**: Dibuja una elipse.
- **pygame.draw.line**: Dibuja una linea

pygame.surface.fill(color, rect=None, special_flags=0): Rellena la superficie especificada, o el fragmento de esta indicado por el rect con el color dado.

Por tanto nuestro programa que dibuja un rectángulo quedaría de la siguiente manera:

pygame.init();

screen=pygame.display.set_mode((640,460))

pygame.display.set_caption("Pygame tutorial")

rect=(0,0,32,32)

color=(255,0,0)

screen.fill((255,255,255))

pygame.draw.rect(screen,color,rect)

pygame.display.update()

pygame.time.delay(2000)

pygame.quit()

De nuevo nos encontramos con que lo habitual será querer usar imágenes creadas con programas de dibujo externos. Para esto al igual que SDL, pygame usa las siguiente funciones y estructuras:

pygame.image: Módulo para la lectura y escritura de imágenes

pygame.image.load(filename): Carga la imagen especificada por filename. Devuelve una surface que representa la imagen.

pygame.surface.blit(source, dest, area=None, special_flags = 0): Dibuja la imagen source sobre la que invoca a la función en el área especificada por el rect dest. Puede pasársele un rect area que especifica qué parte de la imagen se desea copiar.

Una vez que conocemos estas funciones básicas podemos reproducir lo que previamente hicimos con SDL y mostrar nuestra imagen de HelloWorld.

```
pygame.init();

screen=pygame.display.set_mode((640,460))

pygame.display.set_caption("Pygame tutorial")

image=pygame.image.load('helloworld.bmp')

screen.fill((255,255,255))

screen.blit(image,(100,100,image.get_width(),image.get_height()))

pygame.display.update()

pygame.time.delay(2000)

pygame.quit()
```

Es importante destacar que la función load de pygame es compatible no solo con imágenes bmp, si no que incorpora todos los formatos que añadía SDL_Image.

Box2D:

Introducción:

Box2D es un motor en C++ para la simulación de solidos rígidos en 2D. Se distribuye bajo licencia zlib.

La librería incluye numerosos módulos que permiten simular multitud de comportamientos como: detección de colisiones continuas, sistemas de solidos con diversos tipos de ligaduras, apilamientos de diferentes formas geométricas, sistemas en árbol para la gestión de cuerpos, etc.

Toda la funcionalidad de la librería esta orientada a la simulación física y numérica y, por tanto, es necesario el uso de otras herramientas para visualizar los elementos en pantalla.