Universidad Politécnica de Valencia

Memoria PSVITA

Autores:
Francisco Elías Serrano
Martínez-Santos
Jose Alemany Bordera

15 de mayo de 2014

Índice

Resumen

Desarrollar con herramientas oficiales siempre ha requerido de una empresa con juegos publicados y ser viable en el futuro, aparte de pagar por el kit de desarrollo. Sony, durante el año 2013, lanzo una herramienta para desarrollar aplicaciones y videojuegos en dispositivos con el certificado Playstation Mobile. Con ella nos brindan un año gratis para ser publicadores.

1. Introducción

En esta memoria se abarcarán las herramientas usadas para empezar a desarrollar aplicaciones y videojuegos usando PSM SDK, los pasos para darse de alta para ser publicador y un ejemplo de videojuego que acompañará a esta memoria explicado a modo de tutorial de como se ha ido creando el juego SpaceShips[1].

2. Herramientas

2.1. Desarrollo

La herramienta utilizada para desarrollar ha sido el PSM SDK 1.21.02[2] .La aplicación puede dar problemas al iniciarse, ya que al ejecutarla no hace nada. Esto es un problema según que GTK Sharp[3] tengáis, por lo que recomiendo la versión 2.12.9-2[4].

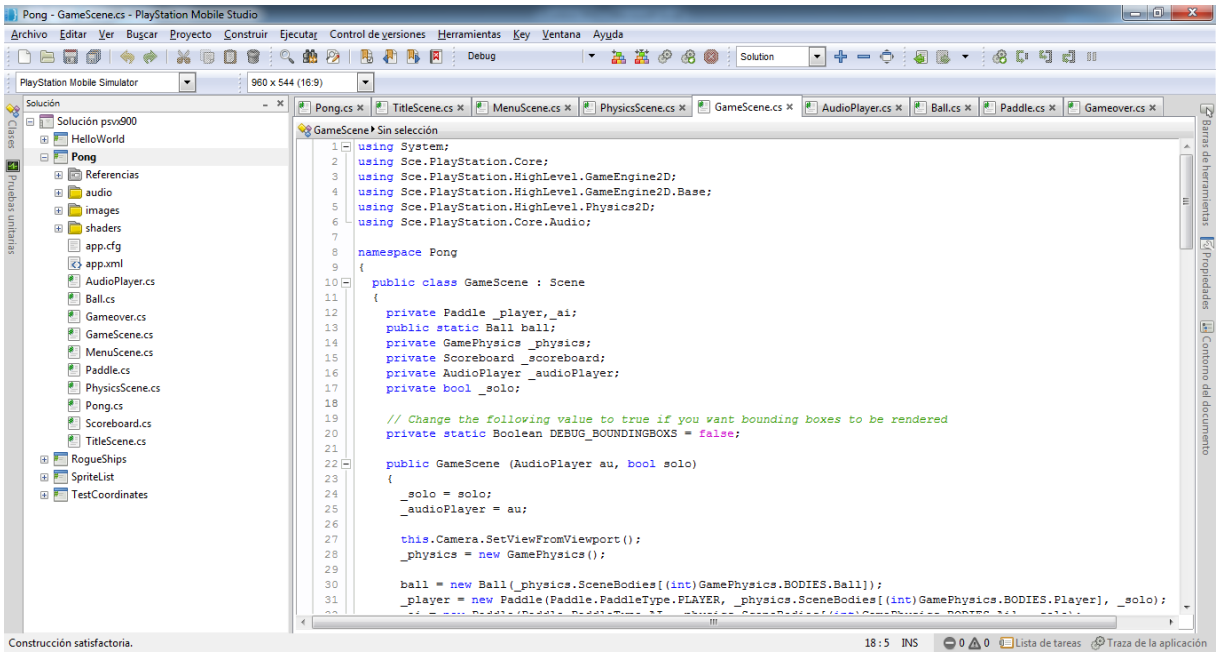


Figura 1: Instantanea desarrollo PSVita

[1]https://github.com/Reisor/SpaceShips
[2]http://psm-resource.np.dl.playstation.net/psm_resource/general/all/PSM_SDK_1.21.02.exe
[3]http://www.go-mono.com/mono-downloads/download.html
[4]http://download.mono-project.com/gtk-sharp/gtk-sharp-2.12.9-2.win32.msi

Si usáis windows 8 o 8.1, el programa no instalara el .NET Framework 3.5 ni el mono, por lo tanto al ejecutar no hará nada. Tendréis que instalarlo por vuestra cuenta bajándolo de las paginas oficiales.

Para que nuestras aplicaciones funcionen en una PS Vita necesitaremos tenerla actualizada a la última versión y tener la misma versión de PSM que el usado para desarrollar. Para actualizar el PSM podemos hacerlo de dos formas distintas: dirigirnos a Ajustes ->Actualizar, o descargarnos una aplicación oficial del Store de PlayStation y ejecutarla para que así se actualice.

2.2. Gráficos

Para generar las imágenes que han sido utilizadas en el juego se ha empleado la aplicación Pyxel Edit[5] ya que es una herramienta orientada a la creación de pixelart, que permite generar tileset con las que podemos hacer tilemaps y obtener un xml, para ser procesado por nuestros programas.

Aun así, para generar los xml con el contenido de los niveles se ha usado Tiled[6], ya que no solo permite indicar dónde van las texturas, sino que también podemos generar objetos que pueden contener enemigos, jefes de fase, etc.

Para crear imágenes también se puede usar el Aseprite[7].

[5]http://pyxeledit.com/index.php
[6]http://www.mapeditor.org/
[7]http://www.aseprite.org/

3. Primeros pasos con el PSM SDK

3.1. Como obtener el PSM SDK

Para obtener el software lo que hay que hacer es dirigirnos a la página web oficial PSM DevPortal[8] donde nos registraremos para obtener el software de desarrollo, o también podemos usar una cuenta de PlayStation Network si lo deseáis.

Una vez hecho esto ya podremos descargarnos el programa.

3.2. Como ser Desarrollador

En la página de PSM DevPortal tendréis que entrar con vuestro usuario y daros de alta como publisher, esto requerirá unos pocos pasos donde os pedirán una cuenta bancaria para domiciliar el pago una vez se acabe el año gratis, aunque es opcional.

Tras todo esto, solo tenéis que esperar a que os manden un correo de verificación, suele tardar unos días, así que no desesperéis.

3.3. Como firmar Aplicaciones

Con la aplicación PSM Publishing Utility, que hemos instalado previamente con el software descargado de la web oficial, podrémos firmar nuestras aplicaciones y videojuegos.

Para ello vamos al apartado "*Key Management*" y pulsamos el icono de la llave con un más. A continuación nos pedirá un nombre para la clave, ponéis el que queráis, y luego tendréis que poner la cuenta que os han verificado para ser desarrolladores.

Tras eso le damos al símbolo +, y agregamos el archivo "*app.xml*" que estará situado en la carpeta del proyecto, es recomendable que editéis dicho archivo y cambiéis el nombre del proyecto ya que por defecto pone un asterisco (*).

Una vez hecho todo esto, seleccionamos la aplicación y pulsamos el icono del juego de llaves con la semilla y un circulo de flechas, le damos a sí y nos volvemos a conectar con la cuenta de desarrollador. Si todo ha ido bien te dirá que la aplicación ya está firmada y ya puede ser usada por el PlayStation Mobile Suite.

3.4. Ejecutar usando el Simulador y un Dispositivo

Abrimos la aplicación PlayStation Mobile Suite, que hemos instalado previamente con el software descargado de la web oficial, con la que podremos desarrollar nuestras aplicaciones y videojuegos, y creamos un nuevo proyecto.

A continuación hacemos click derecho sobre la carpeta de Referencias y le damos a "*Editar Referencias*". Ahora añadimos la librería:

Sce.Playstation.HighLevel.GameEngine2D

Tras eso modificamos el fichero "*AppMain.cs*" y añadimos el siguiente código:

[8] https://psm.playstation.net/portal/en/index.html

Listing 1: Hello world

```
using System;
using System.Collections.Generic;
using Sce.Pss.Core;
using Sce.Pss.Core.Environment;
using Sce.Pss.Core.Graphics;
using Sce.Pss.Core.Input;

using Sce.Pss.HighLevel.GameEngine2D;
using Sce.Pss.HighLevel.GameEngine2D.Base;
using Sce.Pss.Core.Imaging;

namespace HelloWorld
{
  public class AppMain
  {
    public static void Main (string[] args)
    {
      Director.Initialize();

      Scene scene = new Scene();
      scene.Camera.SetViewFromViewport();

      var width = Director.Instance.GL.Context.GetViewport().Width;
      var height = Director.Instance.GL.Context.GetViewport().Height;

      Image img = new Image(ImageMode.Rgba, new ImageSize(width,height),
                            new ImageColor(255,0,0,0));
      img.DrawText("Hello World", new ImageColor(255,0,0,255),
                   new Font(FontAlias.System, 170, FontStyle.Regular),
                   new ImagePosition(0,150));

      Texture2D texture = new Texture2D(width, height, false, PixelFormat
          .Rgba);
      texture.SetPixels(0,img.ToBuffer());
      img.Dispose();

      TextureInfo ti = new TextureInfo();
      ti.Texture = texture;

      SpriteUV sprite = new SpriteUV();
      sprite.TextureInfo = ti;

      sprite.Quad.S = ti.TextureSizef;
      sprite.CenterSprite();
      sprite.Position = scene.Camera.CalcBounds().Center;

      scene.AddChild(sprite);

      Director.Instance.RunWithScene(scene);
    }
  }
}
```

Para ejecutar el programa es tan fácil como darle a la pestaña de Ejecutar y luego elegir Ejecutar como y seleccionamos si queremos utilizar el simulador o un dispositivo. El código anterior solamente genera en pantalla el texto “Hello world”, este es el ejemplo más básico de programa.

4. Proyecto SpaceShips

Este es un ejemplo de videojuego desarrollado mediante la utilización de las herramientas anteriormente mencionadas. En este documento se pretende que el lector pueda aprender a desarrollar su propia aplicación o videojuego.

Para ello se mencionarán en los siguientes apartados los aspectos más importantes sobre dicho juego que permitirán al lector comprender su funcionamiento. Las siguientes explicaciones están separadas en funcionalidades propias de la API y aspectos del videojuego en sí.

4.1. API del Videojuego

4.1.1. Carga e impresión de Imagenes

Si queremos mostrar una simple imagen en nuestra aplicación o juego lo único que tendremos que hacer será:

- Declarar una variable del tipo Texture2D.
- Cargar una imagen usando el constructor.
- Declarar una variable del tipo SimpleSprite de la API.
- Opcional: Indicar en qué posición del eje de coordenadas queremos la imagen.
- Llamar al método render del sprite.

Podemos verlo en forma de código en el siguiente ejemplo.

Listing 2: Carga de Imagenes

```
public Texture2D testTexture;

test = new Texture2D("/Application/image/test.png", false);

public SimpleSprite spriteTest;

spriteTest.Position.X = rectScreen.Width/2 - textureTitle.Width/2;
spriteTest.Position.Y = rectScreen.Height/2 - textureTitle.Height;

spriteTest.render();
```

Este es el método de cargar imágenes más sencillo, ya que la API se encarga de crear el sprite usando shaders, pero no es el más eficiente. Para cargar las imágenes de los actores suele hacerse de forma diferente, pero para imágenes que se muestras muy pocas veces, por ejemplo el titulo o gameover, es más cómodo de esta forma.

A continuación veremos un método mejor para cargar muchas imágenes.

4.1.2. Carga de Imágenes Eficiente

Ya hemos visto como cargar imágenes, pero si usáramos ese método para cargar todas las imágenes del juego (aun solo teniendo que cargar 6 tipos de imágenes para los actores) haríamos muchos sprites por pantalla que daría lugar a que los FPS bajaran a 15-20 dependiendo del sistema. Esto es debido a que el porcentaje gastado en renderizado superaría el 100 %.

De ese modo, para que podamos cargar 1000 sprites y que el render no pase del 60 % lo que haremos será:

- Crear un archivo .bat o ejecutar en el terminal la siguiente linea de comando:

  ”%SCE_PSM_SDK%/tools/UnifyTexture.exe” --output
  <Ruta destino>/<nombre a elegir>
  <Ruta imagen 1>/<imagen 1> ... <Ruta imagen n>/<imagen n>

  Esto generará un .png con todas las imágenes juntas y un .xml con la ruta que hemos dado de la imagen, su posición y su tamaño en el fichero de la imagen conjunta.

- Crear una variable del tipo PackedIndexedSprite e inicializarla usando PackedIndexedSprite (graphics).
- Crear una variable del tipo Dictionary<string, UnifiedTextureInfo>.
- Inicializar la variable usando el método UnifiedTexture.GetDictionaryTextureInfo(string path) pasándome como parámetro la ruta del xml.
- Declarar una variable del tipo Texture2D.
- Cargar una imagen conjunta utilizando el constructor.
- Actualizar y renderizar la variable del tipo PackedIndexedSprite.

La forma de representarlo en nuestro programa seria:

Listing 3: Carga de Imagenes Empaquetadas

```
PackedIndexedSprite piSprite = new PackedIndexedSprite(graphics);

Dictionary<string, UnifiedTextureInfo> dicTextureInfo dicTextureInfo =
    Texture2D UnifiedTexture.GetDictionaryTextureInfo("/Application/image
    /test.xml");

Texture2D textureUnified=new Texture2D("/Application/image/test.png",
    false);

SpriteB spriteB = new SpriteB(gs.dicTextureInfo["image/myship.png"]);

graphics.Disable(EnableMode.DepthTest);
```

```
graphics.SetTexture(0, this.textureUnified);

piSprite.Clear();

piSprite.Update();
piSprite.Render();
```

Aunque no veáis correlación entre las variables no es así, ya que cuando piSprite lanza los métodos Update y Render estos llaman a todas las SpriteB que hay.

4.1.3. Carga de Imagenes desde el API

En los apartados anteriores se ha explicado como cargar las imágenes pero no como funciona por dentro la API. Resulta muy fácil usar la API, pero siempre viene bien saber cómo se comporta por si tenemos comportamientos extraños, errores o por si queremos hacer algún cambio.

Para cargar una imágenes necesitamos una textura cargada usando Texture2D y el vertex shader (*.vcg y *.fcg). Con eso, lo que tenemos que hacer primero es inicializarlo.

Listing 4: Inicialización

```
public static void Initialize ()
{
    graphics = new GraphicsContext();
    ...
    texture = new Texture2D("/Application/resources/Player.png", false);
    shaderProgram = new ShaderProgram("/Application/shaders/sprite.cgx");
    ...
}
```

De este modo cargamos la textura y iniciamos el shaderProgram, aunque veáis que carga un .cgx que no existe no os preocupes, cuando construis el proyecto junta los dos archivos shader en uno, crea el .cgx, que es el que tenemos que cargar.

A continuación tenemos que crear 3 arrays, uno para los vértices, otro para las coordenadas de la textura y uno ultimo para los colores. Estos sirven a la hora de rellenar el vertex buffer, ya que es donde está la información que indica donde y como cargar la textura.

La forma escogida, una de tantas, es trabajar con dos polígonos triangulares y crear un cuadrilátero.

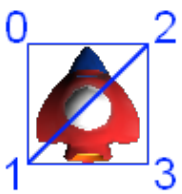


Figura 2: Ejemplo de un vertex buffer

Entonces, ahora indicaremos que coordenadas de la textura vamos a coger para rellenar el vertex buffer y que color queremos en cada punto.

Listing 5: Creación de Arrays

```
public class AppMain
{
    ...
    static float[] vertices=new float[12];

    static float[] texcoords = {
            0.0f, 0.0f, // 0 arriba izquierda
            0.0f, 1.0f, // 1 abajo izquierda
            1.0f, 0.0f, // 2 arriba derecha
            1.0f, 1.0f, // 3 abajo derecha
        };

    static float[] colors = {
            1.0f,   1.0f,   1.0f,   1.0f,   // 0 arriba izquierda
            1.0f,   1.0f,   1.0f,   1.0f,   // 1 abajo izquierda
            1.0f,   1.0f,   1.0f,   1.0f,   // 2 arriba derecha
            1.0f,   1.0f,   1.0f,   1.0f,   // 3 abajo derecha
        };

    const int indexSize = 4;
    static ushort[] indices;
    ...
}
```

Como podemos ver, aún queda por rellenar los vértices, estos se encargan del tamaño que tendrá el cuadrilátero donde sale la imagen. La forma normal de rellenar es utilizando los anchos y altos de la textura, aunque pueden haber casos que se deseen otros tipos de tamaños. Para el caso normal el código seria:

Listing 6: Inicialización de los vértices

```
public static void Initialize ()
{
    graphics = new GraphicsContext();
    ImageRect rectScreen = graphics.Screen.Rectangle;

    texture = new Texture2D("/Application/resources/Player.png", false);
    shaderProgram = new ShaderProgram("/Application/shaders/Sprite.cgx");
    shaderProgram.SetUniformBinding(0, "u_ScreenMatrix");

    vertices[0]=0.0f;    // x0
    vertices[1]=0.0f;    // y0
    vertices[2]=0.0f;    // z0

    vertices[3]=0.0f;    // x1
    vertices[4]=texture.Height; // y1
    vertices[5]=0.0f;    // z1

    vertices[6]=texture.Width;  // x2
    vertices[7]=0.0f;    // y2
    vertices[8]=0.0f;    // z2

    vertices[9]=texture.Width;  // x3
```

```
        vertices[10]=texture.Height;      // y3
        vertices[11]=0.0f;   // z3
        ...
    }
```

Llegado a este punto ya tenemos todo lo que necesitamos, ahora solo queda volcar toda la información al Vertex Buffer.

Listing 7: Añadir la información al Vertex Buffer

```
public static void Initialize ()
{
    ...
    vertexBuffer = new VertexBuffer(4, indexSize, VertexFormat.Float3,
        VertexFormat.Float2, VertexFormat.Float4);
    ...
    vertexBuffer.SetVertices(0, vertices);
    vertexBuffer.SetVertices(1, texcoords);
    vertexBuffer.SetVertices(2, colors);

    vertexBuffer.SetIndices(indices);
    graphics.SetVertexBuffer(0, vertexBuffer);
}
```

A continuación solo queda indicar el sistema de conversión de los pixeles hacia la pantalla y cargar la imagen. Por defecto, el punto $(0, 0)$ está en la esquina superior izquierda, por lo tanto, si queremos trabajar con otro sistema hay que hacerlo nosotros mismo, por lo que yo he escogido el siguiente sistema de coordenadas.

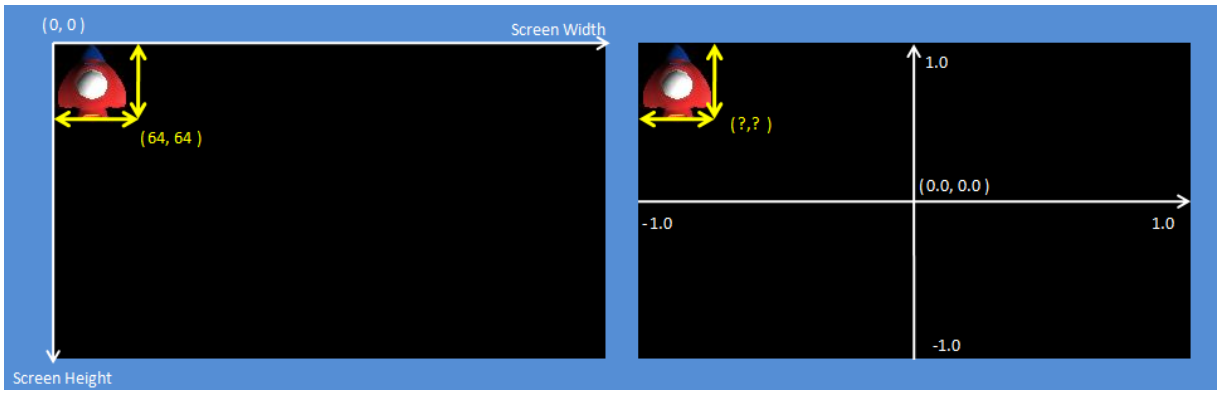


Figura 3: Coordenadas 2D del juego (izquierda) y sistema de coordenadas de la pantalla (derecha)

Por lo tanto, en el código escribiremos:

Listing 8: Sistema de coordenadas y carga de imagen

```
public static void Initialize ()
{
    ...
    ImageRect rectScreen = graphics.Screen.Rectangle;
    ...
```

```
        screenMatrix = new Matrix4(
                    2.0f/rectScreen.Width, 0.0f,          0.0f, 0.0f,
                              0.0f,    -2.0f/rectScreen.Height,    0.0f, 0.0f,
                                        0.0f,    0.0f, 1.0f, 0.0f,
                                                  -1.0f,   1.0f, 0.0f, 1.0f
                                                      );
    }

    public static void Render ()
    {
        graphics.Clear();

        graphics.SetShaderProgram(shaderProgram);
        graphics.SetTexture(0, texture);
        shaderProgram.SetUniformValue(0, ref screenMatrix);

        graphics.DrawArrays(DrawMode.TriangleStrip, 0, indexSize);

        graphics.SwapBuffers();
    }
```

Y con esto, si queremos que se muestre la imagen por pantalla, solo tenemos que llamar a la función Render.

4.1.4. La clase GameFramework

Esta clase es la encargada de hacer que el juego se ejecute, ya que en ella se encuentra el bucle principal como ya hemos explicado.

A la hora de hacer nuestro juego, tenemos que crear una clase que herede de esta clase, y para iniciar el programa, desde la clase principal llamaremos al método Run de nuestras clase hija de GameFramework. Con ello ejecutara el bucle principal, que se encarga, antes de empezar el bucle, iniciar el reloj y el entorno gráfico.

El entorno gráfico, la variable graphics, es una de las variables más importantes, dado que en ella tenemos toda la información referente a los gráficos y es la encargada de dibujar las texturas por pantalla. El reloj sirve a la hora de hacer el cálculo de los FPS, porcentaje de actualización y renderización.

El bucle principal se encarga de lanzar el método Update, el que se encarga de guardarse en la variable gamePadData los botones que han sido apretados y luego llama a la función CalculateProcessTime el cual calcula los FPS, el porcentaje de uso del método Update y Render de todo el juego.

Cuando hablamos de a los métodos Update y Render nos referimos a todas las clases del juego, ya que cuando en nuestra clase hijo y en todos los Actores llamaremos a esos dos métodos para actualizar la información y, en algunos casos, mostrarlos por pantalla. Por lo tanto, calcular estos porcentajes viene muy bien a la hora depurar.

4.1.5. La clase UnifiedTexture

Cuando generamos una imagen unificada como ya hemos explicado, el programa nos proporciona un .xml, este nos sirve a la hora de crear el diccionario que guarda la posición y tamaño de cada elemento en la imagen conjunta.

Lo único que tenemos que hacer es crear un diccionario que la clave sea un string y la descripción un UnifiedTextureInfo. Tras eso llamamos al método GetDictionaryTextureInfo que se encarga de devolvernos el diccionario con la información.

Por ejemplo podríamos hacerlo de la siguiente manera:

Listing 9: Crear un diccionario

```
public Dictionary<string, UnifiedTextureInfo> dicTextureInfo;
dicTextureInfo = UnifiedTexture.GetDictionaryTextureInfo("/Application/
    assets/image/game.xml");
textureUnified=new Texture2D("/Application/assets/image/game.png", false)
    ;
```

4.1.6. La clase PackedIndexedSprite

Esta clase se encarga de controlar la imagen unificada. Hay que tener en cuenta que solo podemos tener un objeto de ella, ya que usa la variable graphics del GameFramework y solo puede haber una sola, por lo tanto, al usar la textura unificada solo puede ser una.

Lo primero que debemos hacer es inicializar la clase en nuestra clase hijo de GameFramework

Listing 10: Inicializar la clase

```
piSprite = new PackedIndexedSprite(graphics);
```

En este momento, tenemos que llamar al método Clean, para que libere todos los elementos que había renderizado en el frame anterior. Por lo tanto, cada vez que se renderiza hay que limpiar todos los elementos antes de volver a renderizarlos.

A continuación, lo que haremos será ir añadiendo los SpritesBuffer, esto puedo hacerse, por ejemplo, en cada Actor. Una vez tenemos esto llamamos a los métodos Update y Render que se encargar de preparar todos los elementos y luego los muestra por pantalla.

Por lo tanto, en el método Render de nuestra clase hija de GameFramework deberíamos tener:

Listing 11: Inicializar la clase

```
piSprite.Clear();

Root.Render();

piSprite.Update();
piSprite.Render();
```

Antes de tener confusiones, Root.Render() lo que hace es ejecutar el método Render de todos los hijos de Root, en el caso del juego, Root es la clase Actor principal, lo que haría que se ejecutara el método de todas las clases que hereden de esta.

4.1.7. Las clases que muestran imágenes o texto

Pese a que hay varias clases para mostrar imágenes o texto (DebugString, SimpleSprite, SpriteBuffer y TextSprite), el funcionamiento interno de todos ellos es muy parecido. Como

hemos explicado en el apartado de cómo se cargan las imágenes en la API, todos estos métodos tienen esos mismos pasos implementados. Aun así son idénticos, ya que cada uno tiene sus peculiaridades.

- SpriteBuffer: Hay que usar esta clase si queremos usar la imagen unificada, ya que genera el sprite pasándole la información del diccionario.
- SimpleSpite: Crea una imagen simple de un archivo especificado. Se puede cargar una imagen con más de una textura y luego especificar las coordenadas de la textura que queremos de esta imagen.
- DebugString: Crea una imagen a partir de un String, usado por el GameFramework para mostrar la información del juego.
- TextSprite: Se usa junto con Text y sirve para crear una imagen a partir de una String pero usando una fuente especificada.

Todas estas clases tienen variables para poder cambiar el color de la textura, darle un ángulo para que la imagen rote, cambiar su posición, cambiar las coordenadas de la textura para mostrar solo una parte. Para hacer eso de todo esto solo tenemos que, una vez hemos creado un objeto a partir de una, llamar al método o a la variable, en el caso de que sea público, y cambiar su valor.

4.2. Funcionamiento del Videojuego

4.2.1. Jerarquia del Videojuego

En el proyecto SpaceShips hay dos clases principales que actúan también como interfaces.

- **GameFramework**, encargada de declarar todas las funciones principales de inicialización, actualización y renderizado. De esta clase hereda Game y se encarg de implementar el motor del juego a nuestra manera.
- **Actor**, que define los métodos y funciones de cada entidad del juego (personaje, enemigos, balas, etc.). La mayoría de clases de nuestro proyecto hereda de esta y forman una estructura en forma de árbol.

  El actor *Root* corresponde al juego en sí, este contiene una lista a actores gestores de las diferentes partes del juego como *Player*, *EnemyManager*, *enemyBulletManager*, etc. Y estos a su vez contienen a los actores a los que gestionan.

  En la siguiente imagen podemos observar la jerarquía de la clase Player. En esta no representamos el árbol completo del videojuego solo la descendencia del Player.

El uso de estas jerarquías es debido a que C# no tiene herencia múltiple, y una manera de solucionarlo es utilizando una interfaz donde las demás hereden. No solo se hace por eso, sino también porque luego es muy cómodo trabajar con nuevos actores que redefinan los métodos del padre.

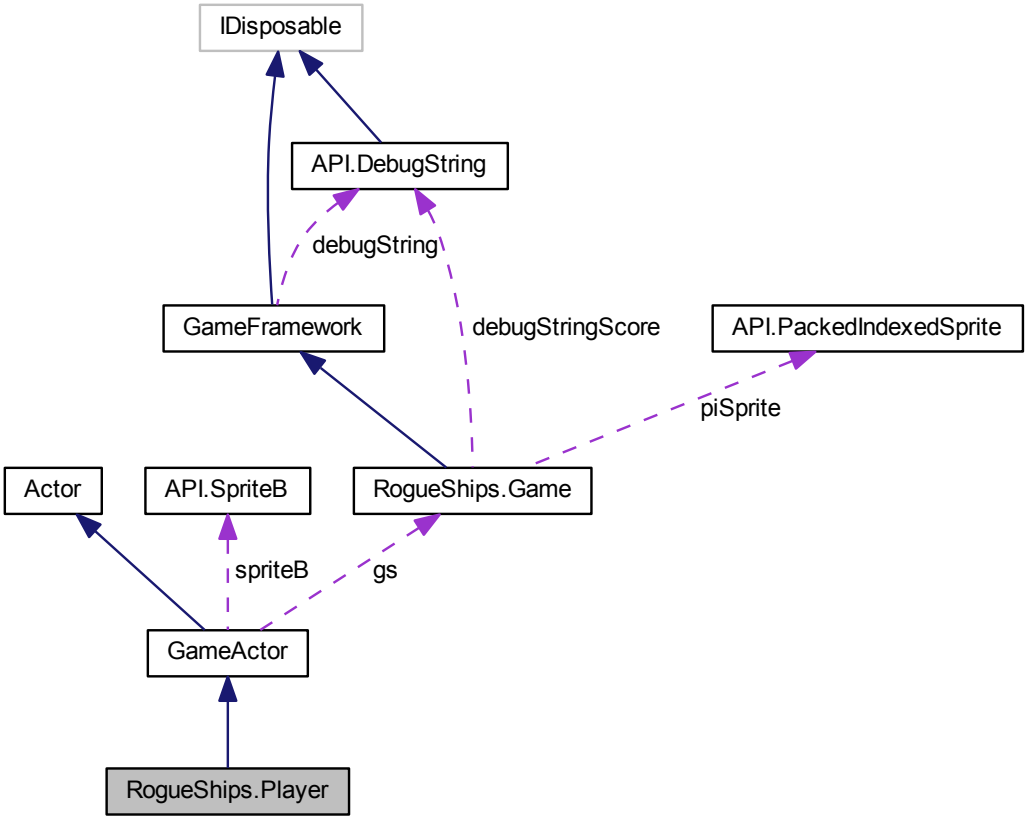


Figura 4: Jerarquía de la clase Player

4.2.2. Bucle Principal

Todos los juegos, tanto de pequeñas como grandes empresas, siguen el siguiente esquema:

Listing 12: Bucle según Jason Gregory

```
while (true)
{
  PollJoypad();
  float dt = g_gameClock.CalculateDeltaTime();

  for (each gameObject) gameObject.Update(dt);

  g_animationEngine.Update(dt);
  g_physicsEngine.Simulate(dt);
  g_collisionEngine.DetectAndResolveCollisions(dt);
  g_audioEngine.Update(dt);
  g_renderingEngine.RenderFrameAndSwapBuffers();
}
```

Este esquema está ideado por Jason Gregory, encargado de dirigir el equipo de diseño de jugabilidad en Naughty Dogs, donde podemos ver una organización ideal.

Para el juego SpaceShips se ha usado el siguiente:

Listing 13: Bucle principal de SpaceShips

```
public void Run(string[] args)
{
  Initialize();

  while (loop)
  {
    time[0] = (Initializent)stopwatch.ElapsedTicks;
    SystemEvents.CheckEvents();
    Update();
    time[1] = (int)stopwatch.ElapsedTicks;
    Render();
  }

  Terminate();
}
```

Este método se llama nada más iniciar el juego y se puede ver que difiere un poco del ideado por Jason Gregory pero, aunque no se vea, sigue el mismo esquema.

Como he comentado en la sección del apartado anterior, gracias a la herencia, cuando se llama a Initialize, Update y Render esto llama a todos los actores, de este modo, comprobará si se ha pulsado alguna tecla, detectará las colisiones, reproducirá los sonidos y renderizará.

4.2.3. Detección de Colisiones

En el proyecto SpaceShips existe una clase que hereda de actor y que se encuentra en el primer nivel del árbol que se encarga de la detección de colisiones.

En cada Update ejecutado del bucle principal se realiza una comprobación de colisiones mediante una adaptación de la técnica de detección de colisiones AABB, ya que si la nave es la siguiente

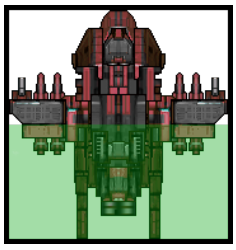


Figura 5: Ejemplo de Detección de Colisiones

y el cuadrado exterior representa su caja de inclusión, solo comprobamos la colisión de los objetos con el rectángulo verde interno. Esto se hace ya que la colisión es este juego siempre se produce en el mismo sentido.

Ademas solo existen tres tipos de colisiones en el SpaceShips:

- Las colisiones producidas por las balas del Player con los Enemigos.
- La colisión del Player con una nave Enemiga.
- Las colisiones producidas por las balas enemigas con el Player.

4.2.4. Inteligencia Artificial

Los enemigos son los agentes del videojuego a los cuales dotamos de inteligencia, es decir, de un comportamiento con el objetivo de destruir al jugador.

La técnica empleada para la implementación de esta es mediante un lenguaje imperativo, sucesiones de sentencias case. La clase enemigo tiene unos estados para el movimiento y un rango de importancia que nos ayuda a decidir los comportamientos que tendrá cada enemigo en cada momento.

Un ejemplo que ilustra este comportamiento es el siguiente

Listing 14: IA Enemy para el movimiento a realizar

```
private void UpdateMovement()
{
  switch(enemyMovement)
  {
    case EnemyMovement.Nothing:
      defaultMovement();
      break;

    case EnemyMovement.Positioning:
      if(Vector3.DistanceSquared(positionI, spriteB.Position) > 2*speed)
      {
        if((spriteB.Position.X >= positionI.X + speed) || (spriteB.
            Position.X <= positionI.X - speed))
          spriteB.Position.X += speed*(spriteB.Position.X > positionI.X ?
              -1 : 1);
        if((spriteB.Position.Y >= positionI.Y + speed) || (spriteB.
            Position.Y <= positionI.Y - speed))
          spriteB.Position.Y += speed*(spriteB.Position.Y > positionI.Y ?
              -1 : 1);
      }
      else enemyMovement = EnemyMovement.Nothing;
      break;

    case EnemyMovement.Horizontal:
      UpdatePositionX();
      if(cnt > 500) enemyMovement = EnemyMovement.Vertical;
      break;

    case EnemyMovement.Vertical:
      UpdatePositionY();
      if(cnt > 500 && enemyLevel < 9) enemyMovement = EnemyMovement.
          Horizontal;
      break;

    case EnemyMovement.Bounce:
      UpdatePositionX();
```

```
            UpdatePositionY();
            break;

        case EnemyMovement.Exit:
            spriteB.Position.X += speed;
            break;

        case EnemyMovement.Stop:
            break;

        default:
            break;
    }

    if(enemyLevel <= 2 && cnt > enemyLevel*1000)
    {
        cnt = 0;
        speed = 2*speed;
        enemyMovement = EnemyMovement.Exit;
    }
}
```

en el cual podemos apreciar que dadas unas determinadas condiciones pasamos de un estado a otro del movimiento. Esta característica también se encuentra presente a la hora de elegir qué tipo de disparo se efectuará y en otros.

4.2.5. Reproducir Música y Sonidos

Para hacer eso no hace falta crear nada especial, el SDK ya nos da los métodos necesarios para hacerlo de manera sencilla. Por lo tanto si queremos cargar un sonido y una canción y reproducirlo haremos lo siguiente:

1. Crear una variable del tipo Sound.
2. Cargar el sonido con el constructor.
3. Crear una variable del tipo SoundPlayer.
4. Usar el método CreatePlayer() del sonido.
5. Llamar al método play() para reproducir el sonido del reproductor
6. Crear una variable del tipo Bgm.
7. Cargar la canción con el constructor.
8. Crear una variable del tipo BgmPlayer.
9. Usar el método CreatePlayer() de la bgm.
10. Llamar al método Play() del reproductor de bgm.

En forma de código seria:

Listing 15: Reproducción de Música o Sonido

```
Sound soundTest = new Sound("/Application/sound/test.wav");
SoundPlayer soundPlayerBullet = soundTest.CreatePlayer();
SoundPlayer.Play();

Bgm bgm = new Bgm("/Application/sound/test.mp3");
BgmPlayer bgmPlayer = bgm.CreatePlayer();
bgmPlayer.Play();
```

Por otra parte, las clases SoundPlayer y BgmPlayer tienen más funciones, no solo la de reproducir.

Función	BgmPlayer	SoundPlayer	Explicación
Play	✓	✓	Reproduce el archivo
Stop	✓	✓	Detiene el audio
Pause	✓		Pausa la música
Resume	✓		Continua la reproducción

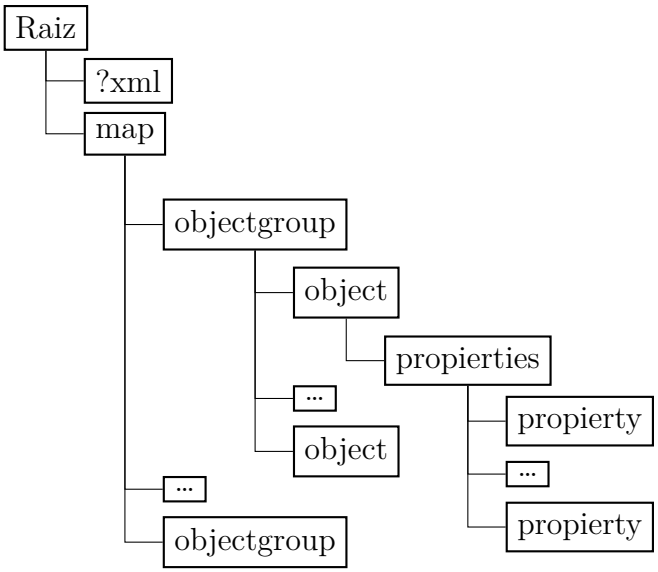
Y también tenemos otro conjunto de propiedades:

Propiedad	BgmPlayer	SoundPlayer	Explicación
Status	✓	✓	Devuelve el estado en que se encuentra la ejecucion
Volume	✓	✓	Especifica el nivel de sonido (0.0∼1.0)
Loop	✓	✓	Repite el audio una vez finaliza
Pan		✓	Indica por que altavoz emite el sonido (-1.0∼1.0)
PlaybackRate	✓	✓	Tasa de velocidad y tono (0.25∼4.0)
Time	✓	✓	Tiempo de reproducción
Duration	✓	✓	Duración de la reproducción
LoopStart	✓		Especifica cuando empieza el bucle
LoopEnd	✓		Especifica cuando acaba el bucle

4.2.6. Leer niveles de Tiled

Cuando creamos un mapa con Tiled podemos guardarlo con diferentes codificaciones, xml, base64, base64 con compresión gzip o zlib o csv. Para el juego se ha elegido xml.

Los archivos generados por Tiled siguen el siguiente esquema:



Podemos ver que todos los archivos contienen dos elementos padres, uno de ellos está presente en todos los xml que el que contienen la versión y codificación del xml, y el siguiente contiene la información acerca de nuestro mapa, en el podemos extraer su versión, orientación, ancho, alto, ancho y alto del tile.

El map contiene ya todo lo referido a lo que hemos puesto en el, grupo de objetos o patrones. Como usamos los fondos los cargamos por otros medios, en el solo encontramos el grupo de objetos, donde cada uno contendrá ciertos objetos que serán enemigos, potenciadores, etc...y cada objeto tendrá unas ciertas propiedades.

Si por ejemplo queremos cargar un archivos, que contiene:

Listing 16: Nivel de ejemplo

```
<?xml version="1.0" encoding="UTF-8"?>
<map version="1.0" orientation="orthogonal" width="56" height="17"
    tilewidth="32" tileheight="32">
 <objectgroup name="Enemy" width="56" height="17">
  <object name="Enemy" type="enemy" x="416" y="256" width="32" height
      ="29">
   <properties>
    <property name="File" value="assets/image/enemy/flanker.png"/>
    <property name="Level" value="2"/>
    <property name="Life" value="10"/>
    <property name="Puntuation" value="1000"/>
    <property name="Speed" value="1.0"/>
   </properties>
  </object>
  <object name="Enemy" type="enemy" x="0" y="192" width="32" height="29">
   <properties>
    <property name="File" value="assets/image/enemy/spectreb.png"/>
    <property name="Level" value="1"/>
    <property name="Life" value="1"/>
    <property name="Puntuation" value="100"/>
    <property name="Speed" value="3.0"/>
   </properties>
  </object>
```

```
 </objectgroup>
</map>
```

Con el siguiente código podríamos controlarlo:

Listing 17: Ejemplo de parseador

```
XElement root = doc.Element("map");

mapWidth = int.Parse(root.Attribute("width").Value);
mapHeight = int.Parse(root.Attribute("height").Value);
tileWidth = int.Parse(root.Attribute("tilewidth").Value);
tileHeight = int.Parse(root.Attribute("tileheight").Value);

foreach( var objectGroup in root.Descendants("objectgroup"))
{
  int level = 1, life = 1, puntuation =1;
  float speed = 0.15f;
  Vector3 position;

  foreach( var element in objectGroup.Descendants("object"))
  {
    name = element.Attribute("name").Value;
    position.X = float.Parse(element.Attribute("x").Value);
    position.Y = float.Parse(element.Attribute("y").Value);
    position.Z = 0.2f;
  }

  gs.Root.Search("enemyManager").AddChild(new Enemy(gs, name, position,
                                                    puntuation, speed, life))
                                                        ;
}
```

Podemos ver como en root cargamos el elemento y sub-elementos de map y tras eso nos guardamos los 4 atributos del mismo y pasamos a recorrer tus sub-elementos.

4.2.7. Crear botones en la pantalla

Esta función solo esta disponible para dispositivos android. Estos dispositivos, cuando están ejecutando la aplicación si pulsan el botón de menú pueden activar los botones por pantalla. Al activarlos, si no se ha creado el .osc se le dará al jugador unos botones por defecto que podrá cambiar a su gusto, pero si se genera el .osc, podremos especificar que botones y posiciones tendrán por pantalla.

Hay que tener en cuenta que para que estos botones funcionen, en el archivo app.xml ha de estar activo los controles táctiles, que sería este campo:

Listing 18: Extracto de app.xml

```
<feature_list>
    <feature value="GamePad" />
  <feature value="Touch" />  <!-- Esto -->
</feature_list>
```

Para poder abrir el programa habrá que ejecutar

%SCE_PSM_SDK%/tools/OscCustomizeTool/OscCustomizeTool.exe

Con este programa podremos editar que botones queremos por pantalla y en que posiciones. La posición la podemos cambiar tocando en la ventana de visualización o bien dando valores en el editor.

Una vez hayamos terminado, al darle a salvar nos guardara un archivo .osc, que si lo colocamos en el directorio de la aplicación, cuando los dispositivos android activen estos botones, saldrán de la forma que se ha especificado.

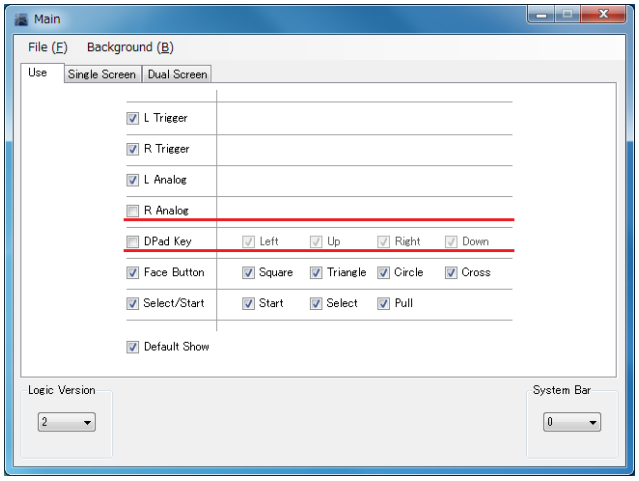


Figura 6: Menu de selección de botones

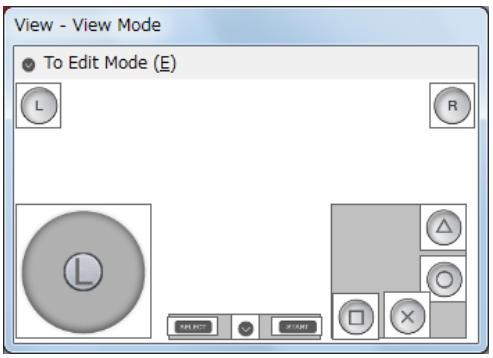


Figura 7: Salida por pantalla