ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
Grado en Ingeniería del Software

Programación de videojuegos de puzzles en 2D usando Unity: tutorial y ejemplo práctico del juego.

Programming of 2D puzzle game using Unity: tutorial and practical example of game.

Realizado por
José Antonio Herrera Peña
Tutorizado por
Antonio José Fernández Leiva
Departamento
Lenguages y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, Septiembre del 2015

Fecha defensa:
El Secretario del Tribunal

Resumen.

Unity3d es un motor gráfico para la implementación de videojuegos creado por *Unity3d Technologies* que tiene el honor de ser una de las principales herramientas que propició el auge de los estudios independientes. Pese a estar enfocado al desarrollo de videojuegos en 3D, los desarrolladores lograban crear contenido en 2D a base de manipulaciones como la cámara y de utilizar herramientas no nativas. Afortunadamente, a partir de la versión 4.3 *Unity3d* integra herramientas nativas que facilitan esta tarea.

Este proyecto consiste en la creación de un tutorial sobre la implementación de un videojuego del género puzles en 2D y enfocado a plataformas móviles, empleando para ello *Unity3d* y concretamente las nuevas herramientas nativas para 2D. La implementación de un videojuego es una tarea muy complicada y extensa, es por ello que en el tutorial se va a tratar la fase de diseño del videojuego y a nivel de implementación centrarse en dichas herramientas para el 2D, mostrando al final una versión jugable del videojuego implementado. Por otra parte, al ser numerosas las herramientas, es imposible tratar cada una de las herramientas aunque se intentará abarcar lo máximo posible y mencionar las que no se incluyan dentro de la implementación del videojuego. El resultado generado de este proyecto puede constituir una referencia para otras personas interesadas en aprender cómo usar *Unity3d* para programar videojuegos en 2D, lo cual les allanará seguramente el camino y les suavizará la curva de aprendizaje.

Palabras claves.

Unity3d, tutorial, videojuegos, 2D, programación.

Abstract.

Unity3d is an engine created by *Unity3d* Technologies for the implementation of video games which has the honor of being one of the main tools that led to the rise of independent companies. Although *Unity3d* is focused on the development of 3D video games, developers were able to create content in 2D through modifications such the camera and using not native tools. Fortunately, from version 4.3 *Unity3d* integrates native tools that facilitate this task.

This Project involves the creation of a tutorial about implementing a puzzle video game in 2D and focused on mobile platforms, using the new native tools for 2D of *Unity3d*. The implementation of a video game is a very complex and extensive task, for this reason the tutorial will focus on the design phase and in the implementation phase with these tools for 2D, showing finally a playable version of the video game implemented. On the other hand, *Unity3d* has many tools so it is imposible to focus on each of the tools but we try to cover as much as possible and mention the tools that were not included in the implementation of the video game. The result of this Project can be a reference for other people interested in learning how to use *Unity3d* to implement 2D video games. This Project will provide them the way and smooth them the learning curve.

Keywords.

Unity3d, tutorial, video games, 2D, programming.

Indice

1. Introducción.

El desarrollo de videojuegos ha estado ligado durante muchos años a grandes empresas con centenares de personas trabajando en un mismo desarrollo, ya que generalmente eran las únicas que podían permitirse pagar las costosas licencias de las herramientas para el desarrollo. Además, la mayoría de estas herramientas no son fáciles de aprender a utilizar ya que requieren de conocimientos muy avanzados, no sólo de programación, en la creación del videojuego intervienen muchos campos como la implementación de la jugabilidad ("*gameplay"*), inteligencia artificial, físicas, gráficos, arte, gamificación, etc. Por suerte, han ido surgiendo herramientas comerciales que facilitan a estudios más humildes la implementación de un videojuego.

Una de estas herramientas es *Unity3d* [Unity3d, 2015] [Kyaw y otros, 2013] que es un motor gráfico creado por *Unity3d Technologies*, y tiene el honor de ser una de las principales herramientas que propició el auge de los estudios "*indies*" o independientes, que son aquellos estudios compuestos por un grupo reducido de personas y en el que generalmente los desarrollos se financian de manera autónoma. *Unity3d* surgió como respuesta a las necesidades de dichos estudios que no podían crear su propio motor gráfico ni adquirir caras licencias, enfocándose al desarrollo de videojuegos en 3D y siendo multiplataforma. A día de hoy es posible exportar el videojuego implementado a prácticamente cualquier plataforma actual: PC (Windows, Linux y Mac), videoconsolas y smartphones. *Unity3d* tiene soporte para *DirectX11* [DirectX 11, 2015], *OpenGL* [OpenGL, 2015] e interfaces propietarias de las videoconsolas.

Unity3d nació a partir de *MonoDevelop* [Monodevelop, 2015], un entorno de desarrollo integrado libre y gratuito que proporciona el scripting tan característico de *Unity3d*. Desde la versión 3.0, *Unity3d* incluye una versión personalizada de *MonoDevelop* para la depuración de scripts y permite la programación en lenguajes tan dispares como *C#* [Jesse Liberty, 2002] y *Javascript* [Shelley Powers, 2008]. Aun así, es posible integrar *Microsoft Visual Studio* como IDE por defecto de *Unity3d*. Otra característica muy interesante es el *Asset Store* [Asset Store, 2015], la tienda oficial de *Unity3d* con recursos gratuitos y de pago. Muchos diseñadores y desarrolladores, en vez de utilizar *Unity3d* para implementar videojuegos, se dedican a publicar recursos o herramientas no nativas para otros desarrolladores. En cuanto al tema de licencias, *Unity3d* dispone de una licencia de pago ($1500 o $75 al mes) y de una gratuita cuyas limitaciones no son determinantes para implementar un videojuego, sino herramientas de soporte y ayuda.

Con el auge de estos estudios independientes y de plataformas con facilidad para publicar videojuegos para los smartphones, se ha vuelto a recuperar los videojuegos en dos dimensiones. En *Unity3d*, pese a estar enfocado al desarrollo de videojuegos en tres dimensiones, los desarrolladores lograban crear contenido en 2D a base de manipulaciones como la cámara y de utilizar herramientas no nativas. Afortunadamente, desde la versión 4.3 (finales del 2013), *Unity3d* integra herramientas nativas que facilitan a los desarrolladores la implementación en 2D.

Existen otras alternativas a *Unity3d* que también son muy utilizadas por los estudios "*indies*", como por ejemplo *Cocos2d-X* [Cocos2d-X, 2015] y *LibGDX* [LibGDX, 2015] que también son multiplataforma. Sin embargo no proporcionan tantas herramientas, además de estar más enfocados al desarrollo en 2D por lo que son más bien considerados "*frameworks*" en vez de motores gráficos. La alternativa que más se acerca a nivel técnico es el recién lanzado *Unreal Engine 4* [Unreal Engine 4, 2015] que permite un mayor despliegue visual, sin embargo *Unity3d* sigue siendo un referente gracias a su política de licencias y a las numerosas características que ha ido incluyendo a lo largo de sus versiones.

Existen muchos tutoriales de *Unity3d* pero la mayoría están enfocados a la implementación de videojuegos en 3D y hay pocos enfocados al 2D y menos aún como ejemplo práctico, es por ello que el proyecto que vamos a realizar consiste en la creación de un tutorial práctico haciendo uso de las herramientas nativas de *Unity3d* para 2D. El ejemplo práctico consistirá en la implementación de un videojuego del género puzles en dos dimensiones y enfocado a plataformas móviles. La implementación de un videojuego desde cero es una tarea muy complicada y extensa, es por ello que en el tutorial se va a tratar la fase de diseño de un videojuego enfocado a plataformas móviles y a nivel de implementación centrarse en dichas herramientas para el 2D, mostrando al final una versión jugable del videojuego implementado. Por otra parte, al ser numerosas las herramientas y al ser el videojuego a implementar de un único estilo, es imposible tratar cada una de las herramientas aunque se intentará abarcar lo máximo posible y mencionar las que no se incluyan dentro de la implementación del videojuego. En la fase de programación nos centraremos en la inteligencia artificial de los enemigos que determinará el comportamiento de éstos. Por último, se incluirá enlaces de interés con recursos gratuitos para utilizar en proyectos independientes. El resultado generado de este proyecto puede constituir una referencia para otras personas interesadas en aprender cómo usar *Unity3d* para programar videojuegos en 2D, lo cual les allanará seguramente el camino y les suavizará la curva de aprendizaje.

1.1. Introducción a Unity3d.

Para este proyecto utilizaremos la versión personal que es gratuita. Recientemente se ha lanzado una nueva versión de *Unity3d*, concretamente la versión 5 en marzo del 2015 y con la cual cambia toda la política de licencias. Hasta esta versión, las diferencias de la versión *Professional* con respecto a la *Personal* se limitaban a la pantalla de "*splash*" personalizable, algunos efectos de imagen y texturas como HDR y varias opciones más que solo eran amortizables en proyectos grandes, por lo que la versión Personal era perfectamente usable para el desarrollo de proyectos "*indies*". Gracias a la competencia, con la versión 5 la versión *Personal* incluye todas las características de la versión *Professional* con la única restricción de que la pantalla de "*splash*" sigue sin ser personalizable. Por otra parte la versión *Professional* incluye de forma adicional varias herramientas de soporte como por ejemplo backup en la nube, una herramienta para reportar los bugs u otra herramienta para gestionar el acceso a betas.

Aunque este proyecto trata de introducir las nuevas herramientas enfocadas para el desarrollo en 2D introducidas en la versión 4.3, utilizaremos la última versión disponible (5.0) pero el proyecto y tutorial puede ser exportado sin problemas a la versión 4.3 o superior.

1.2. Crear un nuevo proyecto.

Antes de profundizar con la herramienta *Unity3d*, lo primero es instalarla y configurarla junto con otras herramientas que nos facilitarán el desarrollo de un videojuego. Dichas tareas están explicadas con detalle en el Anexo técnico. Una vez que tenemos todas las herramientas instaladas y configuradas, ejecutamos *Unity3d*. Si anteriormente hemos estado trabajando en algún proyecto, nos abrirá automáticamente dicho proyecto, en caso contrario, nos aparecerá una pequeña ventana como se muestra en la Figura 1.2 para crear un nuevo proyecto o abrir alguno de los que tenemos en el disco duro.

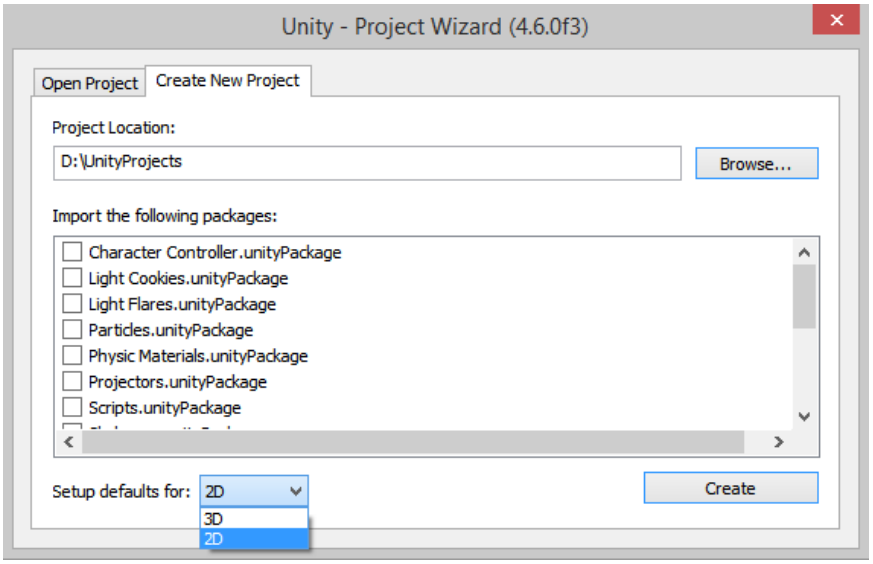


Figura 1.2. Ventana de nuevo proyecto o abrir un proyecto existente.

En esta ventana elegiremos el directorio donde se guardará el proyecto y los diferentes paquetes a importar, al igual que en anteriores versiones de *Unity3d*. La diferencia radica en un pequeño desplegable con las opciones "2D" y "3D", y que sirve para que *Unity3d* se configure por defecto con la elección escogida. En nuestro caso, dado que se trata de un videojuego en 2D, elegimos la opción "2D". En un principio no seleccionamos ningún paquete aunque más adelante necesitaremos algunos de ellos lo añadiremos a posteriori. Pulsamos el botón "Create" para crear un nuevo proyecto y nos configure la ventana principal de trabajo de *Unity3d*.

1. Introducción.

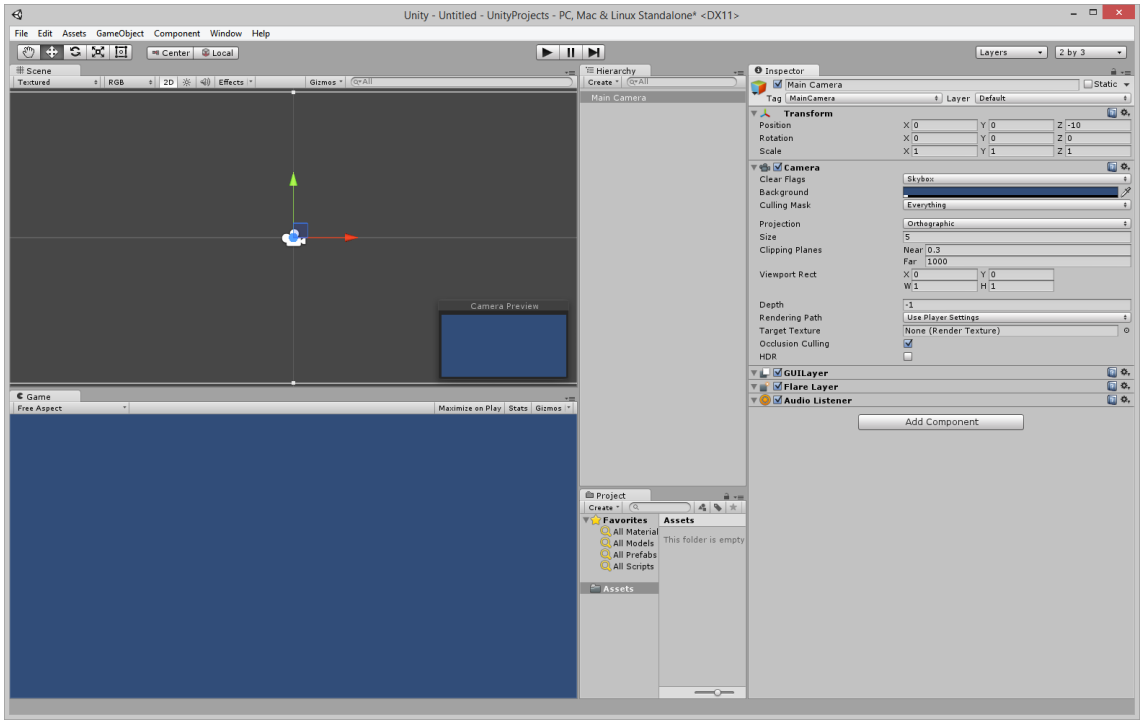


Figura 1.3. Ventana principal de trabajo de Unity3d.

Como se puede apreciar en la Figura 1.3, la ventana principal de trabajo del editor está compuesta por varias vistas que se pueden personalizar arrastrándolas a la ubicación deseada. En este tutorial se da por hecho que el lector tiene unos conocimientos mínimos de *Unity3d* y ha trabajado con dicha herramienta, por lo que nos centraremos en las peculiaridades del 2D.

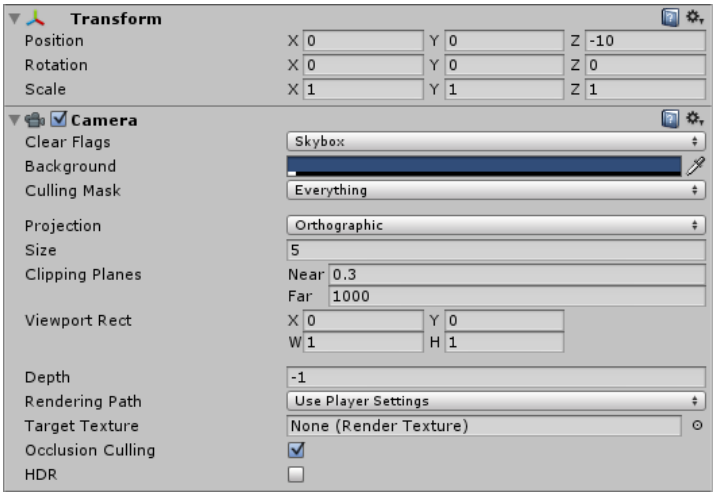


Figura 1.4. Configuración por defecto de la cámara para el 2D.

En las propiedades de la cámara, la propiedad "Projection" indica la capacidad de la cámara para renderizar objetos con o sin perspectiva. Con el 2D esta propiedad se auto configura en "Orthographic", es decir, la cámara renderizará los objetos de forma uniforme y sin perspectiva por lo que pierde completamente la profundidad. Aun así, el eje Z se sigue manteniendo aunque no lo apreciemos. La otra opción es "Perspective" que es la que añade la perspectiva a la cámara para el renderizado de objetos. Al tener configurada esta propiedad con "Orthographic", se habilita la propiedad "Size" que indica el tamaño de la vista gráfica de la cámara. A más tamaño, más lejos se situará la cámara, y viceversa.

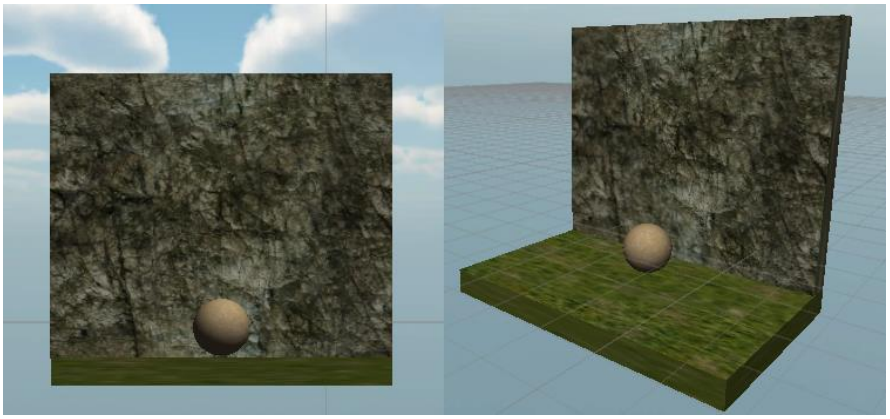


Figura 1.5. Diferencias entre la vista en 2D y 3D.

Al perder la profundidad, posicionar los objetos en un determinado orden puede ser una tarea complicada según vaya aumentando el número de objetos en la escena. Para facilitarnos esta tarea, tenemos la propiedad "Sorting Layer" para asignarle una determinada capa a un objeto y un orden en esa capa con respecto a los demás objetos de esa capa. También podemos crear nuevas capas y ordenarlas como se muestra en la Figura 1.6.

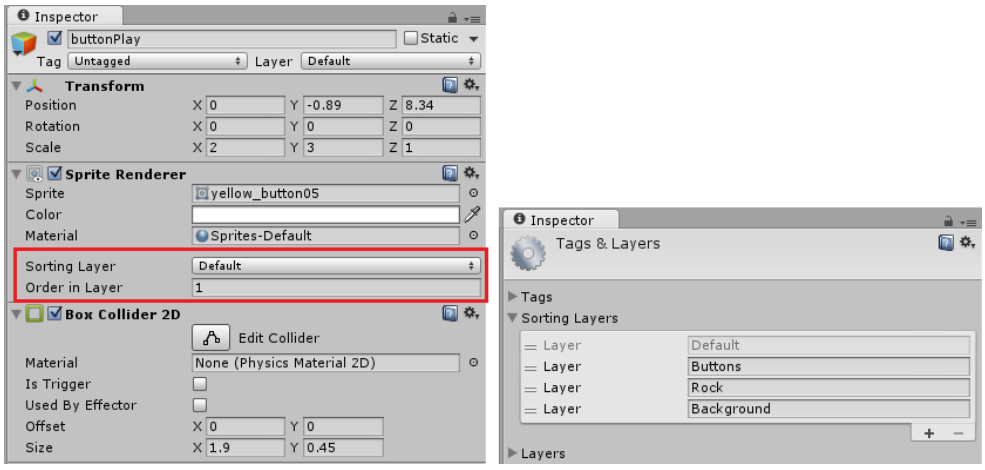


Figura 1.6. Propiedad Sorting Layer y creación de nuevas capas en el "Inspector".

Antes de seguir con el proyecto y profundizar más con *Unity3d*, en el siguiente capítulo vamos a definir un pequeño documento de diseño del videojuego a implementar y que servirá para presentar las nuevas herramientas para el 2D.

1.3 ¿Qué es el documento de diseño de un videojuego?

Un documento de diseño del videojuego, o también llamado "*Game design document*" (GDD) es una parte muy importante en cualquier desarrollo serio, al igual que ocurre en otros tipos de proyectos software en los que se define el documento de Especificación de Requisitos Software (ERS). En este caso es un documento más laxo y orientado al mundo de los videojuegos ya que algunos conceptos como las mecánicas de un videojuego son difíciles de definir en un documento ERS. En él se detallan todas las características del videojuego: género del videojuego, mecánicas, personajes, historia, estilo gráfico, etc. y será usado por todos los integrantes del equipo como guía de trabajo.

La extensión de este documento depende del tamaño del proyecto aunque tampoco hay que confundir tamaño con calidad. En este caso, al no ser un videojuego de gran tamaño y tampoco es el principal objetivo de este TFG, describiremos de forma resumida los diferentes apartados del documento de diseño de nuestro videojuego en el Anexo 4.

2. Primera escena e introducción a la herramienta *Animation*.

En el anterior capítulo vimos cómo crear un nuevo proyecto, el cual automáticamente nos crea una escena completamente vacía donde añadiremos los primeros objetos e introduciremos la herramienta *Animation*.

En la figura 2.1 se muestra la primera pantalla de presentación de nuestro juego. La pantalla está compuesta por el logo del juego que presentará una animación creada con la herramienta *Animation*, un botón para pasar a la siguiente pantalla.

Figura 2.1. Primera pantalla de presentación.

Los juegos de las primeras videoconsolas u ordenadores tenían menús estáticos dado la limitación de hardware de la época pero con el tiempo fue aumentando la potencia de estas máquinas y con ellas aparecieron animaciones en los menús que transmitían una mayor calidad en el videojuego. Es posible realizar estas animaciones a nivel de código pero resulta costoso a nivel de programación ya que dependiendo de la animación, puede incluir complejas fórmulas matemáticas. Con la herramienta *Animation* se simplifica esta tarea.

2. Primera escena e introducción a la herramienta *Animation*.

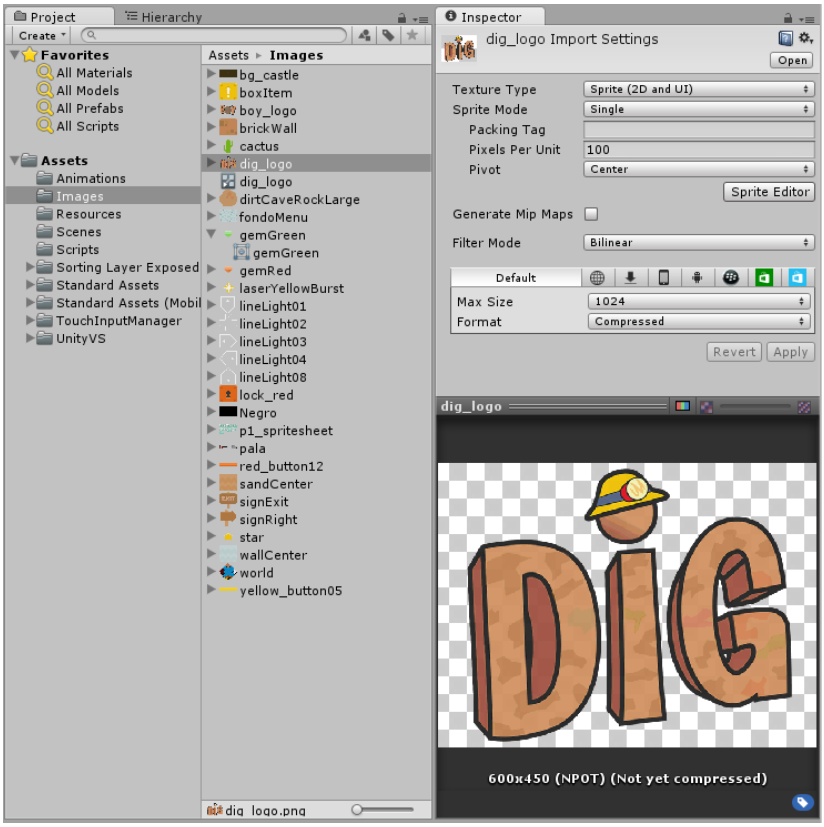


Figura 2.2. Pestaña "Proyect" y pestaña "Inspector" de un recurso en concreto.

Antes de añadir las imágenes del logo que animaremos, crearemos subcarpetas en el proyecto para que sea más fácil identificar cada recurso. Como se puede ver en la figura 2.2, tendremos los siguientes directorios:

- Animations: donde se guardarán las animaciones creadas con la herramienta *Animation* y con la herramienta *Animator* que en próximos capítulos veremos.

- Imágenes: donde se guardarán todas las imágenes que se utilizarán en el juego.

- Resources: donde se guardarán recursos que se acceden desde distintos scripts, como por ejemplo ficheros de lectura.

- Scripts: donde se guardarán los scripts que se utilizarán en el proyecto.

Además de estas carpetas, se crearán otras cuando importemos paquetes.

En la pestaña Inspector de la figura 2.2 aparece los ajustes de la imagen seleccionada. Con *Unity3d* 4.3 aparece el tipo de textura "*sprite*" que es un mapa de bits en 2D que se dibuja directamente en mediante coordenadas. En "Single Mode" seleccionamos "Single" puesto que es una imagen simple y el Pivot lo ponemos centrado para que esté centrado el punto donde se origina las coordenadas locales de la imagen. El "Filter Mode" añade un suavizado a la imagen. Pese a que un filtro trilineal aporta un mayor suavizado, también requiere de la memoria de vídeo del ordenador por lo que para este proyecto lo dejamos por defecto en bilineal. Por último el tamaño máximo y el formato determinan el tamaño máximo de la imagen que *Unity3d* reconoce y el tipo de compresión. La máxima compresión que permite *Unity3d* es 16 bits por lo que si queremos darle a nuestro juego una estética de 8 bits, en el modo de filtro seleccionamos "Point" que pixela la imagen. Dejamos el tamaño máximo por defecto y la compresión en "true color" para mostrar la calidad completa de la imagen.

Figura 2.3. Sprite izquierdo con filtro "Point" e izquierdo con filtro "Bilinear".

Una vez configurada la imagen como "*sprite*", la arrastramos a la ventana de la escena y abrimos la herramienta *Animation* desde el menú "Windows" y nos aparecerá una ventana como se muestra en la Figura 2.4.

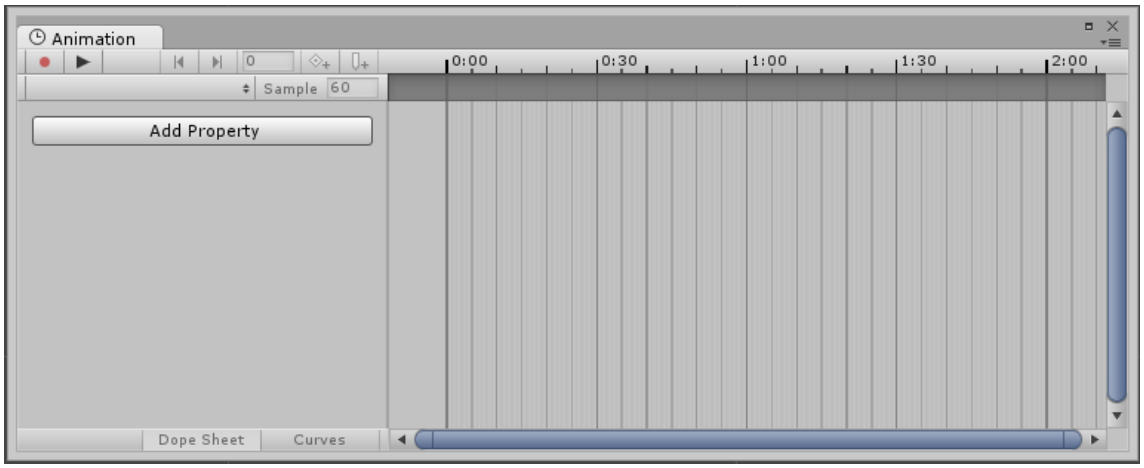


Figura 2.4. Ventana de la herramienta Animation.

2. Primera escena e introducción a la herramienta *Animation*.

La ventana de *Animation* está compuesta por una franja temporal donde se modifica las propiedades de los objetos a lo largo del tiempo. Le damos a "Add Property" para guardar la animación y añadir los objetos. Otra forma de añadir los objetos es modificándolos desde la propia escena mientras tenemos abierta la herramienta *Animation*.

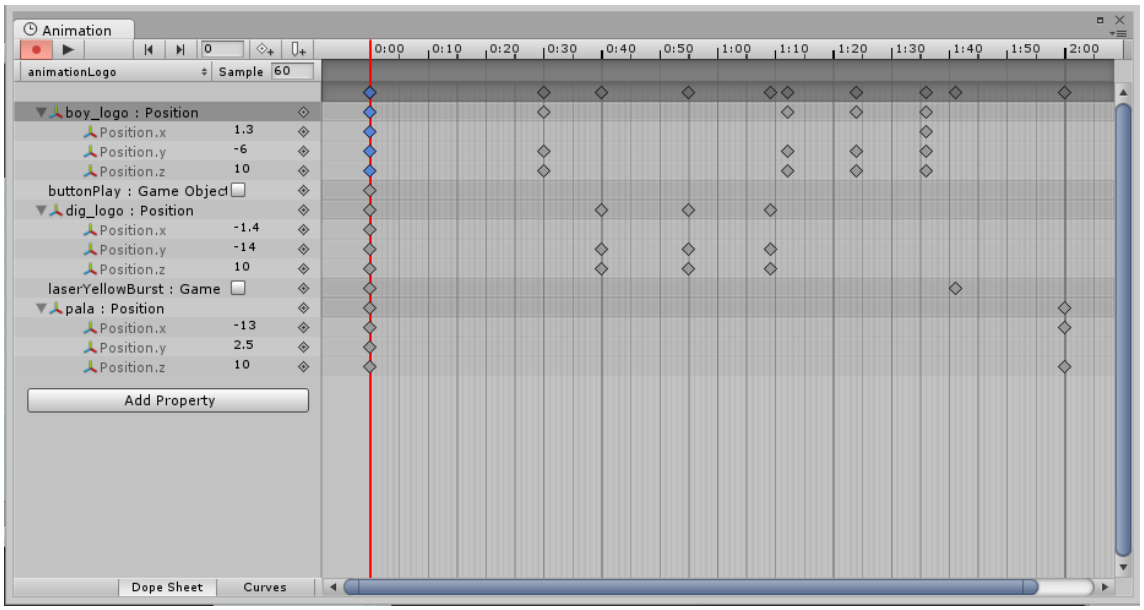


Figura 2.5. Ventana de la herramienta "Animation" con objetos añadidos.

Los rombos en la línea temporal indican los cambios en las propiedades en ese momento. Los cambios no tienen por qué ser solo de posición: también es posible cambiar el color, la opacidad, activarlo o desactivarlo, etc. Pulsando el botón "play" podremos ver la animación en tiempo real. En la figura 2.5 se muestra la animación completa de la primera pantalla de presentación.

2. Primera escena e introducción a la herramienta *Animation*.

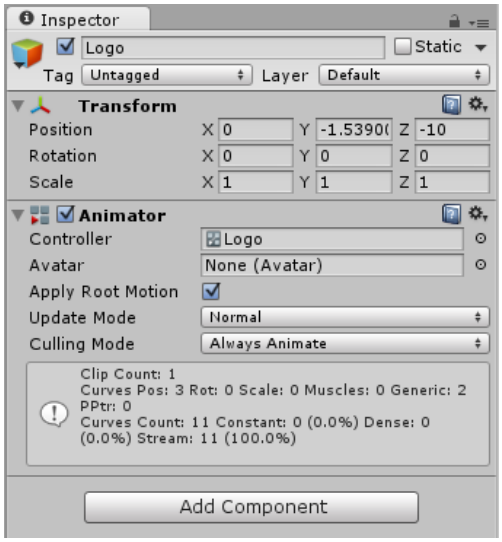


Figura 2.6. Ventana "Inspector" de un objeto con animación.

Una vez que hemos terminado, la animación se añadirá automáticamente al objeto y será visible en la ventana "Inspector" de la Figura 2.6, donde tenemos las opciones de "Apply Root Motion" para activar la animación como "*loop*", la opción "Update Mode" para la animación se sincronice con el engine de físicas del juego o sea independiente, y la opción "Culling Mode" para desactivar la animación cuando el objeto no esté visible.

2. Primera escena e introducción a la herramienta *Animation*.

3. Uso de la clase Input en dispositivos móviles.

La clase Input soporta multitud de dispositivos de entrada convencionales como teclado, ratón, joypad, etc, pero hasta la versión 4.3 no soportaba de forma nativa los dispositivos de entrada táctiles, salvo la clase GUI para construir menús básicos. Para ello se utilizaba al igual que en un juego en 3D, la función "Raycast" de la clase "Physics" que consiste en lanzar un "rayo" desde un punto y con una dirección y comprobar todos los objetos con los que colisiona el rayo.

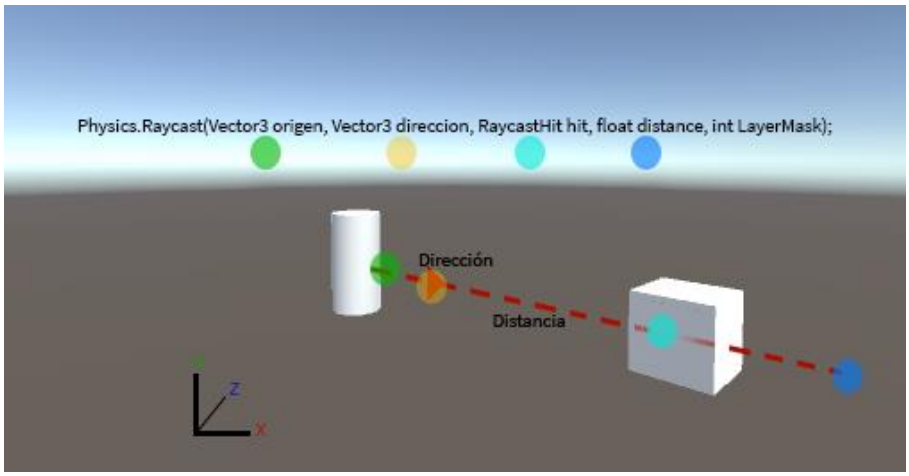


Figura 3.1. Funcionamiento del Raycast.

3.1. Touch con la función OnMouseDown.

En vez de utilizar la función "Raycast", usaremos la función "OnMouseDown" dentro de un script asociado a un objeto que además tenga asociado un componente "Box Collider 2D". Con esta función, cuando hagamos clic o pulsemos sobre dicho *GameObject*, la función "OnMouseDown" será llamada.

```
void OnMouseDown(){
      Application.LoadLevel("CreatePlayer");
}
```

3. Uso de la clase Input en dispositivos móviles.

```
void Update(){
	if(Input.GetMouseButtonDown(0)){
		Ray = Camera.main.ScreenPointToRay(Input.mousePosition);
		RaycastHit hit;
		if(Physics.Raycast(ray, out hit)){
			GameObject gmHit = hit.collider.gameObject;
			if(gmHit.tag == "buttonCreatePlayer"){
				Application.LoadLevel("CreatePlayer");
			}
		}
	}
}
```

Figura 3.2. Diferencias de código entre la función OnMouseDown y la función Raycast.

Como vemos en la Figura 3.2, con "Raycast" es necesario comprobar si se ha pulsado el botón correspondiente y lanzar el rayo en cada frame, es decir, dentro de la función "Update", por lo que es menos optimizado.

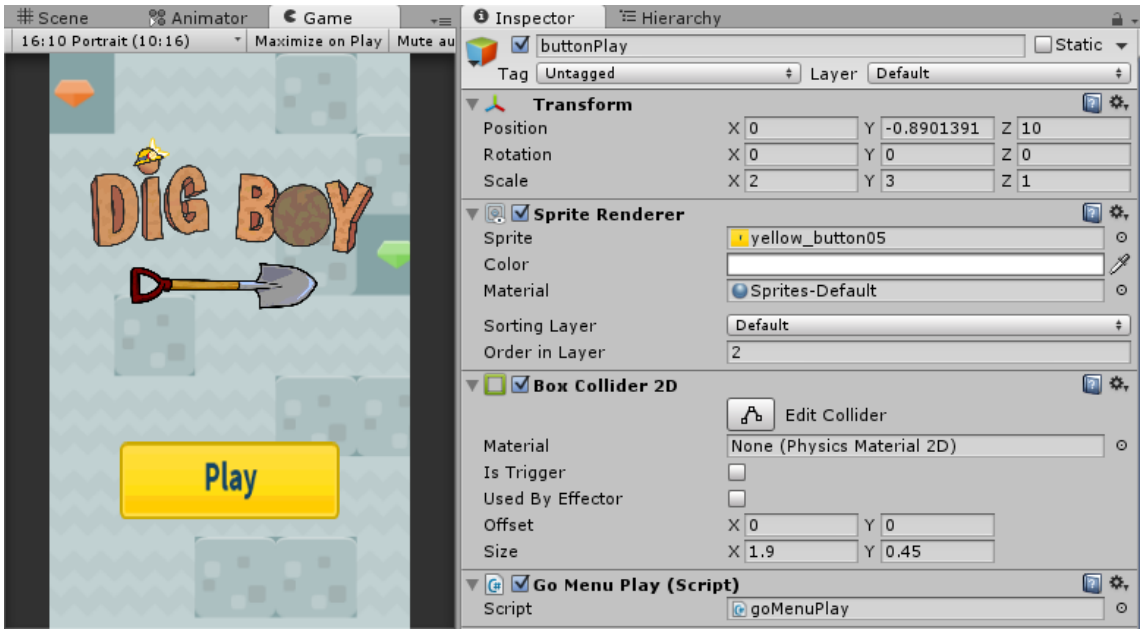


Figura 3.3. GameObject botón con el script asociado que contiene la función OnMouseDown.

Tras asignarle el script al objeto botón y además asociarle un "Box Collider 2D", cuando pulsemos sobre él, detectará el Input y se llamará a la función "OnMouseDown" que en este caso cargará la siguiente escena con la instrucción "Application.LoadLevel ("CreatePlayer")".

3.2. Teclado virtual con TouchScreenKeyboard.

Ya tenemos nuestra pantalla de presentación con un botón para adentrarnos en el juego, pero si es la primera vez que lo ejecutamos, lo lógico es pedir que el usuario escriba un "*nick*" con el que guardar las partidas. Para mostrar un teclado virtual, existe la clase "TouchScreenKeyboard" que estaba ya presente en anteriores versiones pero ha sido mejorada soportando un mayor número de dispositivos táctiles.

Para mostrar un teclado virtual, añadimos una *GameObject* vacío a la escena y le asociamos el siguiente script de la Figura 3.4.

```
private TouchScreenKeyboard keyboard;
private string text;

void Start () {
       keyboard = TouchScreenKeyboard.Open(text,
              TouchScreenKeyboardType.Default,  false, false, false);
}
void Update () {
       if (keyboard.done){
              text = keyboard.text;
              if (text == null || text.Equals(""))
                     keyboard = TouchScreenKeyboard.Open(text,
                            TouchScreenKeyboardType.Default, false, false, false);
             else{
                 PlayerPrefs.SetString("nick", text);
                 Application.LoadLevel("MenuWorld");
             }
         }
}
```

Figura 3.4. Script que añade un teclado virtual.

Para inicializar el teclado virtual, llamamos al método "Open" que se le puede pasar los siguientes parámetros:

- keyboardType: tipo de teclado (cualquier texto, sólo numérico, etc).
- autocorrection: si se aplica autocorrección.
- multiline: si el texto entra en más de una línea.
- secure: si el texto va enmascarado.
- alert: si el teclado virtual está abierto en modo alerta.
- textPlaceholder: texto a usar si el texto actual no contiene ningún texto.

3. Uso de la clase Input en dispositivos móviles.

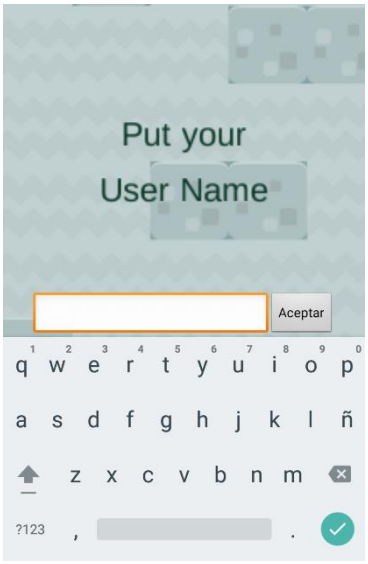


Figura 3.5. Teclado virtual.

3.2. Multi-Touch.

Para explicar el "multi-touch" en *Unity3d* nos saltaremos momentáneamente la pantalla del menú principal para enfocarnos en la pantalla de niveles. En esta pantalla nos deslizaremos con el dedo para navegar por los diferentes niveles. Para que la pantalla se deslice, es necesario implementar un sistema de "multi-touch" que compruebe hacía que dirección movemos el dedo.

Los dispositivos con *Android* o *iOS* son capaces de rastrear hasta cinco dedos tocando la pantalla al mismo tiempo, sin embargo, en el caso de Android al tener más variedad de dispositivos, puede ir de los dos dedos en los dispositivos más viejos hasta en los cinco en los más modernos. Para recuperar el estado de cada dedo tocando la pantalla, se utiliza la matriz de "input.touches". En esta matriz, cada toque de dedo está representado por una estructura "input.touch" que entre sus múltiples propiedades contiene un ID único y la posición para cada toque.

Para implementar el "multi-touch", añadimos un *GameObject* con el fondo de la pantalla y dentro de él añadimos como hijos los *GameObject* que representan cada nivel, tal y como se muestra en la Figura 3.6. Al *GameObject* padre le asignamos el script que controlará el deslizamiento de la pantalla a través del "multi-touch".

3. Uso de la clase Input en dispositivos móviles.

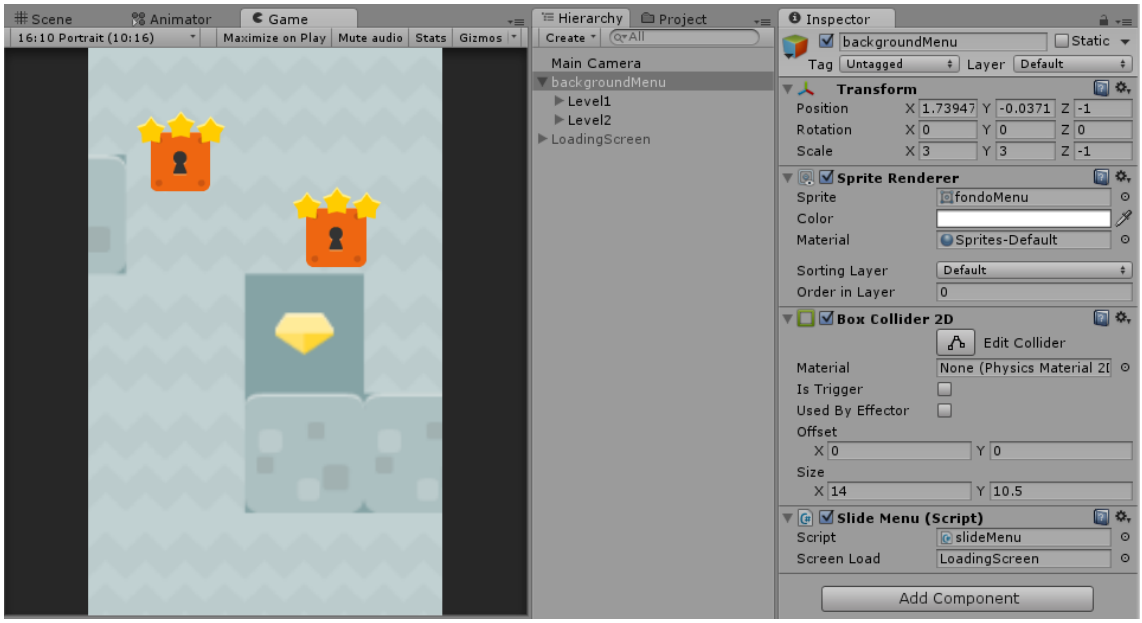


Figura 3.6. GameObject padre con los niveles añadidos como hijos.

```
if(Input.touches.Lenght > 0){
	if(Input.touches[0].phase == TouchPhase.Moved){

		float touchY = Input.touches[0].deltaPosition.y*speed *Time.deltaTime;

		if(transform.position.y + touchY > 0
			&& transform.position.y + touchY < longMax){
			transform.Translate(0, touchY, 0);
		}
	}
}
```

Figura 3.7. Script para controlar el deslizamiento de la pantalla.

En la figura 3.7 se muestra el script para controlar el deslizamiento de la pantalla y que estaría incluido dentro del "Update" para comprobar en cada frame los toques en la pantalla. Comprobamos sólo la primera fila de la matriz "touches" porque en este caso sólo vamos a tener un dedo a la vez tocando la pantalla. Capturamos la posición en el eje "Y" del dedo según la velocidad a la que se mueva la pantalla ya que ésta puede estar en movimiento cuando la toquemos, y si está en los límites, realizamos un "Translate" a la pantalla.

Por último sólo faltaría añadir terminar de implementar la pantalla y añadir la lógica a cada nivel.

3. Uso de la clase Input en dispositivos móviles.

4. Uso de sprites e introducción a la herramienta *Animator*.

En el capítulo 2 definimos lo que es un "*sprite*" y ahora toca definir un "*sprite sheet*" u hoja de sprite, que no es más que una serie de imágenes en un mismo fichero. Este conjunto de imágenes mostradas frame a frame dan lugar a animaciones en 2D. Existen otras técnicas para realizar animaciones en 2D pero el uso de un "sprite sheet" es la técnica más utilizada dado su sencillez.

4.1. Introducción al Sprite Sheet.

En la figura 4.1 se muestra el "sprite sheet" de nuestro personaje, el cual contiene una serie de imágenes, cada una del mismo tamaño (70x70 píxeles). Si reproducimos frame a frame las imágenes enumeradas desde el uno al seis, se visualizaría a nuestro personaje corriendo.

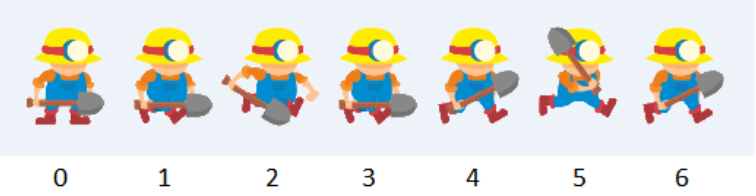


Figura 4.1. Sprite sheet de nuestro personaje.

Es posible realizar esta animación a partir de código, dibujando en cada frame sólo el trozo de imagen que nos interesa, sin embargo, *Unity3d* incorpora una herramienta que nos facilita la tarea de separar cada imagen de la hoja y construir las animaciones. Para ello vamos al inspector del "sprite sheet", configuramos la propiedad "Sprite Mode" como "Multiple" y pulsamos el botón "Sprite Editor".

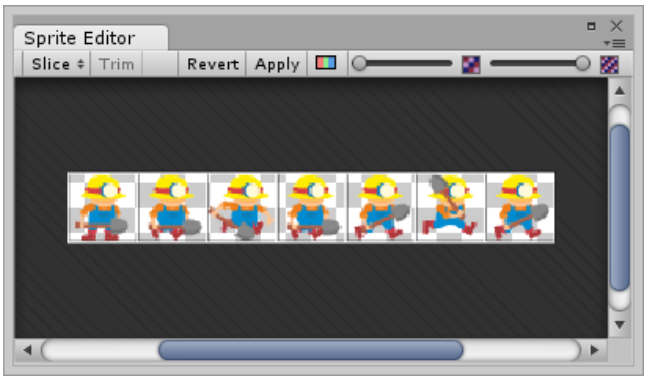


Figura 4.2. Ventana del Sprite Editor.

En la ventana del Sprite Editor (Figura 4.2) podemos seleccionar cada imagen individual de dos formas. La primera dejando que *Unity3d* se encargue de hacerlo automáticamente según la transparencia de la imagen, y la segunda forma indicándole manualmente el tamaño de cada imagen. Como en este caso las imágenes tienen el mismo tamaño (70x70 píxeles), elegimos la segunda opción seleccionando la opción "Grid" dentro del menú "Slice" e indicándole las proporciones. Tras ello, nos aparecerá cada imagen dentro de un recuadro y cuando apliquemos los cambios podremos desplegar el "sprite sheet" en imágenes individuales como se puede apreciar en la Figura 4.3.

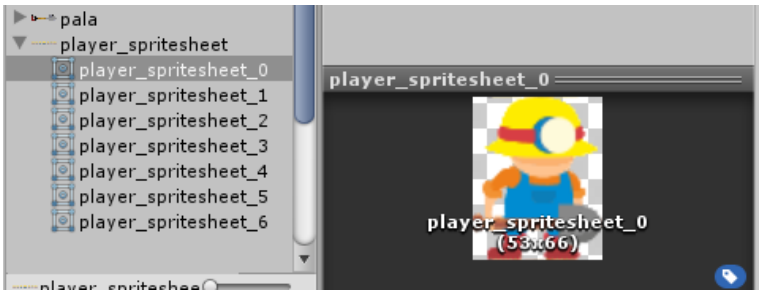


Figura 4.3. Sprite sheet dividido en imágenes individuales.

Para crear la animación, la forma más fácil es seleccionar las imágenes que queramos de la hoja y arrastrarlas a la escena, y así se creará la animación y nos pedirá guardarla.

4.2. Uso de la herramienta *Animator* para controlar las animaciones.

Para hacer uso de la herramienta *Animator*, nos centraremos en la pantalla del menú principal que es la siguiente tras la pantalla de presentación y la pantalla de escribir el "*nick*" del jugador.

Figura 4.4. Pantalla del menú principal.

En esta pantalla seleccionaremos uno de los cuatro mundos y el personaje mostrará una animación de correr mientras el planeta gira hacia nuestro personaje. Con este *GameObject* seleccionado, vamos a "*Windows/Animator"* para que nos muestre la ventana de *Animator*. Con esta herramienta, podremos controlar las animaciones de un *GameObject* en base a un diagrama de estados.

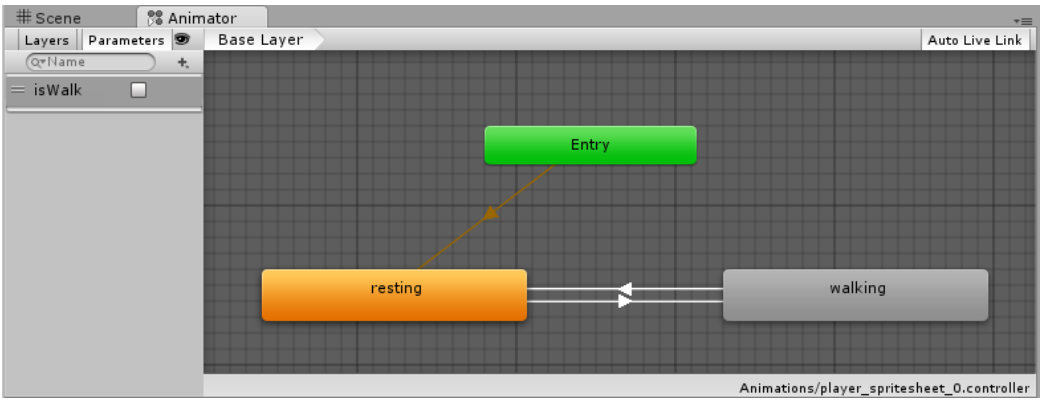


Figura 4.5. Diagrama de estados de nuetro personaje en la pantalla del menú principal.

El diagrama de estados aparecerá con un único estado inicial "Entry". Para crear un nuevo estado, hacemos clic derecho y la opción "Create State" y automáticamente se creará una transición entre el estado inicial y éste. Le ponemos de nombre "resting" pues será el estado cuando el personaje no esté en movimiento. Lo siguiente, arrastramos al diagrama la animación que hemos creado anteriormente ("walking") y creamos dos transiciones en ambos sentidos entre los dos estados como se muestra en la Figura 4.5. Tal y como está ahora, si ejecutásemos la escena, el personaje pasaría indefinidamente por los estados "resting" y "walking" al no existir ninguna condiciones entre dichas transiciones. Para ello, vamos a "parameters" creamos una variable de tipo Boolean ("*isWalk*") y se la asociamos como condición en ambas transiciones pero en una de ellas como *true* y en otra como *false*.

4. Uso de sprites e introducción a la herramienta *Animator*.

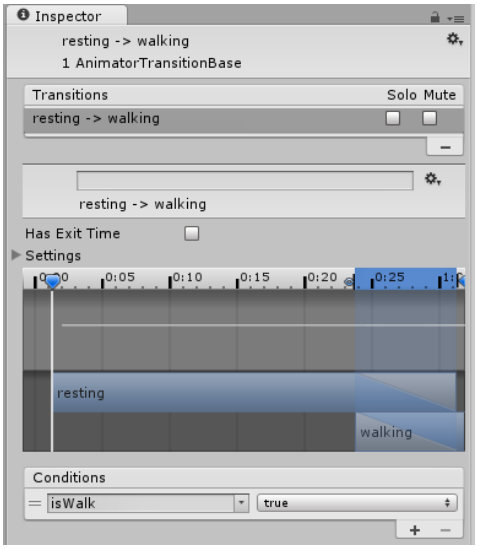


Figura 4.6. Propiedades de una transición.

Como podemos ver en la Figura 4.6, además de asignar condiciones a las transiciones, también podemos configurar el tiempo de éstas. En nuestro caso queremos que la transición sea instantánea por lo que desmarcamos la casilla "Has Exit Time".

Una vez que tenemos nuestro diagrama de estados, para pasar de estados a nivel de código es tan fácil como recoger el "Animator" asociado del personaje y llamar a la variable "isWalk" como se muestra en la Figura 4.7.

```
Animator animatorPlayer = player.GetComponent<Animator>();
animatorPlayer.SetBool("isWalk", true);
```

Figura 4.7. Script que cambia una condición del Animator asociado al GameObject.

Ya solo queda añadir la lógica de la pantalla, la cual está completamente implementada en el Anexo 5.

5. Montando un nivel: controles y lógica.

Ya hemos implementado todas las pantallas excepto la última, la pantalla para jugar a un nivel. En este capítulo nos centraremos en esta pantalla a falta de los enemigos y su inteligencia artificial que lo reservaremos para capítulo 6.

Antes de describir todo el proceso, describiremos brevemente la jugabilidad y la física del movimiento de los objetos. Con *Unity3d* sería muy fácil implementar el movimiento libre del personaje y otros objetos en un escenario mediante el uso de sus colisiones. Sin embargo, en el juego original "*Boulder Dash*" el movimiento de éstos se realizar por cuadrículas a consecuencia de las limitaciones técnicas de su época.

Figura 5.1. Pantalla de nuestro juego (izquierda) y pantalla del juego original Boulder Dash para C64 (derecha).

Para conseguir ese efecto de movimiento, se ha utilizado la siguiente estructura de la Figura 5.2.

```
public static GameObject[,] arraySquare;
public static StateRock[,] arrayStateRocks;
public static float[,] arrayTimeRocks;
public static bool[] arrayBusyColumns;
```

Figura 5.2. Estructura de datos de la lógica de un nivel.

Usaremos una matriz ("arraySquare") para guardar la posición de cada objeto. En el caso de las rocas que pueden caerse y ser movidas, se guardará en otra matriz ("arrayStateRocks") sus estados (*stop, left, right, down*) para tener controlado sus movimientos mediante estados y además existirá otra matriz ("arrayTimeRocks") que controlará el movimiento de una cuadrícula a otra. Para simplificar la física del juego, un solo objeto (roca o diamante) podrá moverse a la vez en una columna y para ello se utiliza

el array “arrayBusyColumns” que bloqueará una columna para que solo una roca o diamante caía por ella a la vez.

Dado la rapidez del movimiento, esta limitación no resultará molesta a la hora de jugar y nos dará facilidad para controlar el movimiento de los objetos mediante código.

5.1. Carga del escenario desde fichero.

Podríamos crear un nivel directamente en la escena como hemos hecho con las demás pantallas, sin embargo, esto implicaría crear una nueva escena para cada uno de los niveles. Además de lo trabajoso que sería crear una escena por cada nivel, estaríamos limitados cuando quisiésemos actualizar nuestro juego con nuevos niveles. Es por ello, que en estos tipos de juegos se suelen cargar niveles predefinidos desde ficheros y así reutilizar la misma escena y código.

Existen muchas formas de cargar un fichero pero nos hemos decantado por un fichero de texto plano en vez de otros ficheros más complejos como el XML que proporciona un mayor control de los datos. La razón de esta elección es porque resulta muy sencillo su lectura y además previsualizar el nivel de una forma más fácil que un XML.

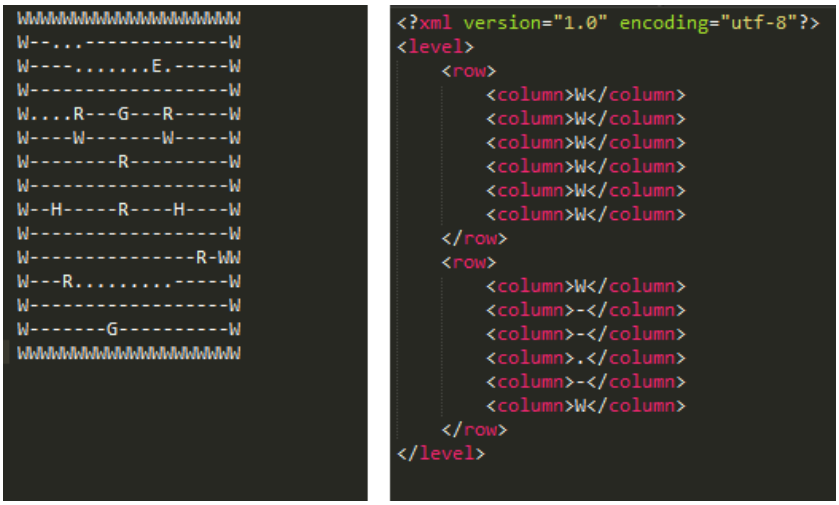


Figura 5.3. Visualización de un documento en texto plano (izquierda) y de un documento XML (derecha).

Lo primero es definir que representará cada uno de los caracteres: “W” pared, “R” roca, “E” enemigo, “G” y “H” diamantes. Una vez tenemos construido nuestro fichero como muestra la figura 5.3, lo guardamos en el directorio “Resources” que es el directorio donde se suelen guardar los recursos que se cargarán posteriormente desde scripts.

Para cargar un fichero u otro, en la pantalla previa de elegir un nivel, guardamos con *PlayerPrefs* el nombre del fichero del nivel a cargar y después en el script del nivel recogemos ese nombre y cargamos el fichero como se muestra en las Figuras 5.4 y 5.5.

```
TextAsset txt = (TextAsset)Resources.Load(filename, typeof(TextAsset));
string contentFile = txt.text;
string[] arrayFileSquare = contentFile.Split(new char[] { '\n' });

columns = arrayFileSquare[0].Length;
rows = arrayFileSquare.Length;

limitRows = rows;
limitColumns = columns;
arraySquare = new GameObject[rows, columns];
arrayStateRocks = new StateRock[rows, columns];
arrayTimeRocks = new float[rows, columns];
arrayBusyColumns = new bool[columns];
```

Figura 5.4. Carga del fichero e inicialización de las estructuras.

```
for (int i = 0; i < arrayFileSquare.Length; i++){
       string rowFile = arrayFileSquare[i];
       for (int j = 0; j < rowFile.Length; j++){
              Vector3 position = new Vector3(OFFSET_X + widthSquare * j, OFFSET_Y -
                     widthSquare * i, 0);
              GameObject squareaux = null;
              Object prefab = null;

              switch (rowFile[j]) {
                     case ('W'):
                     prefab = Resources.Load("wallSquare");
                     squareaux = Instantiate(prefab, position, Quaternion.identity)
                            as GameObject;
                     arraySquare[i, j] = squareaux;
                     arrayStateRocks[i, j] = StateRock.stop;
                     break;
                     case ('E'):
                     prefab = Resources.Load("EnemyWorld1");
                     squareaux = Instantiate(prefab, position, Quaternion.identity)
                            as GameObject;
                     int[] values = new int[] { i, j, (int)speedEnemy, rows,
                            columns };
                     squareaux.GetComponent<MoveSmartEnemy>().ResetEnemy(
                            values);
                     break;
…
…
```

Figura 5.5. Instancia de cada objeto en las estructuras.

5.2. Control del personaje.

Para controlar el personaje, lo primero es chequear los toques en la pantalla mediante un joystick virtual. No sería demasiado complicado utilizando las funciones vistas en el capítulo 3:

1 – Un script asociado a la cámara que incluya una función "OnMouseDown" que compruebe cuando se pulsa la pantalla.

2 – Al pulsar la pantalla, en la función "OnMouseDown" se instancia un objeto "Prefab" (joystick) en la posición del toque.

3 – En el "Update" del script asociado al joystick, se recoge el input.touches y se calcula la dirección de los toques.

Sin embargo, aunque la lógica del joystick es sencilla, depende de otras características como su estética para que resulte atractivo y útil al usuario. En este caso, es una buena oportunidad para hacer uso de la *Asset Store*, la tienda con recursos gratuitos y de pago de *Unity3d*. En ella tenemos multitud de joysticks gratuitos y de pago completamente personalizables por lo que nos ahorrará tiempo. Como se suele decir, no hay que reinventar la rueda. Para acceder a la tienda, vamos al menú "*Windows/Asset Store*".

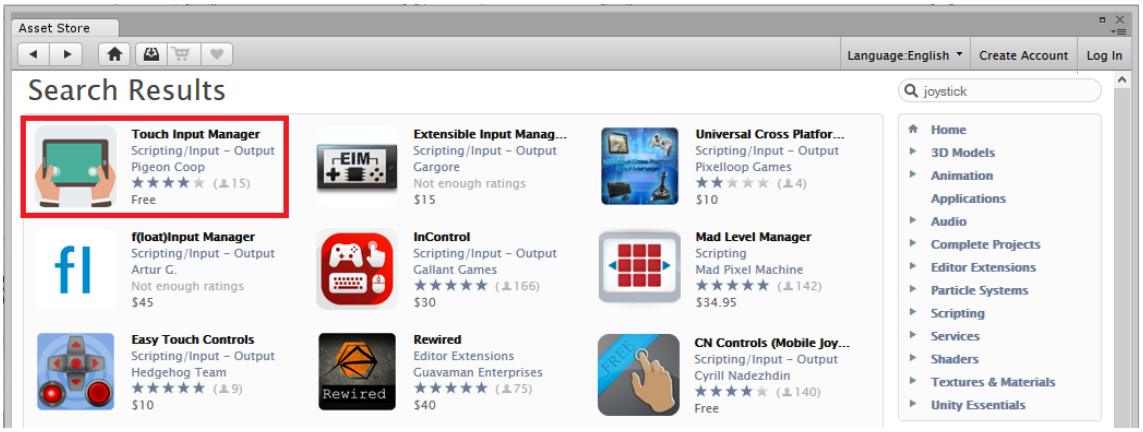


Figura 5.6. Asset Store de Unity3d.

La *Asset Store* está disponible desde las versiones iniciales y en ella podemos encontrar todo tipo de recursos para utilizar en nuestros proyectos. Escribimos "joystick" en la búsqueda y elegimos el recurso destacado en la Figura 5.6 y que en el momento de desarrollar este juego era gratuito. Aun así, hay muchas alternativas a elegir. Lo importamos y se creará la carpeta "TouchInputManager" dentro del directorio "Assets". Además, en el menú Windows nos aparecerá la opción "Touch Input Manager" donde podremos configurar todos los parámetros del joystick o añadir más de un joystick o botones.

Puesto que no es una funcionalidad nativa de *Unity3d* y que puede cambiar si hemos elegido otro recurso, nos centramos directamente en su utilización para recoger los valores que nos interesa como se muestra en la Figura 5.7.

```
TouchInputManager.RenderLayoutAll(true);
TouchInputManager.PassInputToLayoutAll(true);
…
…
Vector2 joystickValue = TouchInputManager.GetJoystick(InputID.DBJoystick,
        LayoutID.DBLayout);
double jx = joystickValue.x;
double jy = joystickValue.y;
```

Figura 5.7. Inicialización y uso del joystick de la Asset Store.

Una vez que tenemos el joystick virtual, podemos empezar a definir el comportamiento y control de nuestro personaje. Nuestro personaje también se moverá por estados (*down, up, left, right*) para controlar el movimiento de una cuadrícula a otra.

```
void Update()
       if (!blockPlayer) {
            if (state.Equals(State.stop))
                updateStatePlayer();
            else
                movePlayerByState();
            if (ControllerLevel.arraySquare[posRows, posColumns] != null){
                if (ControllerLevel.arraySquare[posRows,
                     posColumns].tag.Contains(TAG_GEM)) {
                    updatePoints(posRows, posColumns);
                    //Destruir objeto gema.
                }
               else if (ControllerLevel.arraySquare[posRows,
                           posColumns].tag.Contains(TAG_SAND)){
                    //Destruir objeto arena
                }
                else if (ControllerLevel.arraySquare[posRows,
                    posColumns].tag.Equals(TAG_EXIT)) {
                    //Fin partida (salida)
                }
            }
       }
      else if (!gameOver) {
             //Fin partida (game over)
      }
}
```

Figura 5.8. Método Update con la lógica del movimiento del personaje.

En el método “Update” lo primero que hacemos es comprobar que el usuario no esté bloqueado (ha muerto por una piedra o un enemigo) y realizamos los siguientes pasos:

1. Comprobamos que el personaje está en estado “stop”, es decir, ha llegado a una cuadrícula. En ese caso, actualizamos el estado según los valores que nos devuelva el joystick.

```
private void updateStatePlayer() {
       Vector2 joystickValue = TouchInputManager.GetJoystick(
              InputID.DBJoystick, LayoutID.DBLayout);
       double jx = joystickValue.x;
       double jy = joystickValue.y;

       if ((jx < -0.75f || Input.GetAxis("Horizontal") < 0)
       && posColumns > 0) {//Left
              if (!checkPosition(posRows, posColumns - 1, TAG_ROCK)
                     && !checkPosition(posRows, posColumns - 1, TAG_WALL)) {
                     animatorPlayer.SetBool("isWalk", true);
                     state = State.left;
                     distanceMov = OFFSET_X + widthSquare * posColumns;
                     posColumns--;
              }
              else if (checkPosition(posRows, posColumns - 1, TAG_ROCK)) {
                     ControllerLevel.pushDir = "left";
                     ControllerLevel.pushi = posRows;
                     ControllerLevel.pushj = posColumns - 1;
              }
              if (!isFlip)//Voltear la imagen horizontalmente
              {
                  Flip();
                  isFlip = true;
              }
       }
       …
       …
        //Implementación de Right, Up, Down
```

Figura 5.9. Método updateStatePlayer.

En la figura 5.9 se muestra la lógica para el estado “left”. Si la cuadrícula a la izquierda del personaje no es de tipo pared, cambia el estado a “left”. Además, si en dicha cuadricula existe una piedra que se puede empujar, cambiará el estado de la piedra a “left”. El método “Flip” lo que hace es voltear horizontalmente la imagen del personaje ya que inicialmente nuestro personaje mira hacia la derecha y “checkPosition” comprueba si es accesible esa posición.

2. Lo siguiente es actualizar la posición del personaje según el estado en el que nos encontremos.

3.

```
private void movePlayerByState()
       switch (state) {
              case State.right:
              if (System.Math.Abs(transform.position.x - distanceMov)
                     >= widthSquare) {
                     //Para no esperar al estado Stop y que el movimiento
                     //sea continuo, llamamos al updateStatePlayer();
                     updateStatePlayer();
              }
              else{
                     transform.Translate(speed * Time.deltaTime, 0, 0);
              }
              break;
…
…
```

Figura 5.10. Método movePlayerByState.

4. Por último en la Figura 5.8, comprobamos si en la cuadricula existe un objeto de tipo arena (“Sand”) o gema (“Gem”) y lo eliminamos. En el caso de ser una gema, actualizamos el marcador con la puntuación. En el capítulo 7 implementaremos la puntuación y las diferentes pantallas de “game over” y “victoria”.

5.3. Lógica de las rocas.

La lógica de las rocas se controlará en el método “Update” del script principal del nivel ya que si una roca no tiene suelo debajo o a los lados, ésta automáticamente caerá.

5. Montando un nivel: controles y lógica.

1. Comprobamos el estado de las rocas.

```
private void checkPhysicsRocksGems() {
	for (int i = 0; i < rows; i++){
		for (int j = 0; j < columns; j++){

			//Down (nada debajo):
			if (arraySquare[i, j] != null
				&& (arraySquare[i, j].tag.Equals(TAG_ROCK) ||
				arraySquare[i, j].tag.Contains(TAG_GEM)) && i + 1
				< rows && arraySquare[i + 1, j] == null) {
				if (arrayBusyColumns[j] == false) {
					arrayTimeRocks[i, j] += Time.deltaTime;
					while (arrayTimeRocks[i, j] > TICK_INICIAL
						+ TICK_ADICIONAL) {
						arrayTimeRocks[i, j] = 0;
						arrayStateRocks[i, j] = StateRock.down;
						arrayBusyColumns[j] = true;
					}
				}
			}
			…
			…
			//Implementación para down left y para down right
```

Figura 5.11. Método que actualiza el estado de las rocas.

En el script de la Figura 5.11 recorremos la matriz y para cada roca o gema, comprobamos si no existe ningún objeto a cada lado que les impida caerse (Figura 5.12), y en caso afirmativo, cambia su estado hacía el lado que se va a caer. Bloqueamos la columna en el array "arrayBusyColumns" para impedir que otra roca caiga a la vez por la misma columna.

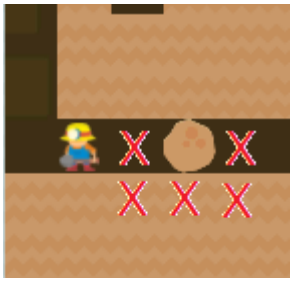


Figura 5.12: cuadriculas que se comprueban para cambiar el estado de una roca.

Para evitar que la roca comience a caer instantáneamente aplastando al personaje, se utiliza un temporizador antes de cambiar su estado.

2. Actualizamos la posición de las rocas o gemas según el estado en el que se encuentren.

```
private void updatePhysicsRocksGems() {
       for (int i = 0; i < rows; i++) {
              for (int j = 0; j < columns; j++){
                 GameObject ob;

                 //Caer abajo (down):
                 if (arrayStateRocks[i, j] == StateRock.down) {
                     ob = arraySquare[i, j];

                     if (arraySquare[i, j] != null &&
                            ob.transform.position.y > OFFSET_Y
                                  - this.widthRock * (i + 1)) {
                         ob.transform.Translate(0, -speedRock *
                                  Time.deltaTime, 0);
                     }
                     else{
                         arrayStateRocks[i, j] = StateRock.stop;
                         Vector3 pos = ob.transform.position;
                         pos.y = OFFSET_Y - this.widthRock * (i + 1);
                         ob.transform.position = pos;
                         arraySquare[i + 1, j] = arraySquare[i, j];
                         arraySquare[i, j] = null;
                         if (i + 2 < rows && arraySquare[i + 2, j] == null){
                             arrayStateRocks[i + 1, j] = StateRock.down;
                         }
                         else{
                             arrayStateRocks[i + 1, j] = StateRock.stop;
                             arrayBusyColumns[j] = false;
                             this.audioSources[0].Play();
                         }
                     }
                 }
              …
              …
              //Implementación del estado de caer a la izquierda y a la derecha
```

Figura 5.13. Método que actualiza la posición de las rocas y gemas.

En la figura 5.13 se muestra la parte del método “updatePhysicsRocksGems” que actualiza la posición de una roca o gema que está en el estado “down” (caída hacia abajo). Mientras no llegue al recuadro destino, seguirá actualizando su posición. Si llega al recuadro destino y éste no tiene ningún objeto debajo, el estado seguirá siendo “down”. En caso contrario, su estado pasará a “stop” y desbloqueará la columna.

3. Por último comprobamos si el personaje ha empujado alguna roca.

```
private void checkRocksPushedByPlayer() {
       if (!pushDir.Equals("stop")){

            // Left:
           if (pushDir.Equals("left") && arraySquare[pushi, pushj] != null
               && arraySquare[pushi, pushj].tag.Equals(TAG_ROCK)
                     && pushj - 1 > 0 && arraySquare[pushi, pushj - 1] == null
                            && (pushi + 1 == rows || (pushi + 1 < rows
                                   && arraySquare[pushi + 1, pushj] != null))) {

                if (arrayBusyColumns[pushj - 1] == false) {
                    arrayTimeRocks[pushi, pushj] += Time.deltaTime;
                    while (arrayTimeRocks[pushi, pushj] > TICK_INICIAL
                           + TICK_ADICIONAL_EMPUJAR) {

                        arrayTimeRocks[pushi, pushj] = 0;
                        arrayStateRocks[pushi, pushj] = StateRock.left;
                        arrayBusyColumns[pushj - 1] = true;
                    }
                }
            }
             …
             …
            //Implementación de Right (derecha)
```

Figura 5.14. Método que comprueba si el personaje ha empujado alguna roca.

En el script asociado al personaje, indicábamos que una piedra era movida con las variables “pushDir”, “pushi” y “pushj” que son públicas y estáticas, es decir, pueden ser accedidas desde otros scripts. Moveremos la roca que se encuentra en la posición indicada al estado que hemos indicado. Al igual que cuando cae, con un temporizador simularemos el “empujón” para que no salga despedida instantáneamente.

6. Añadiendo enemigos con inteligencia artificial.

Unity3d incorpora desde versiones anteriores un sistema de pathfinding formado por mallas de navegación denominado "*navmesh*", sin embargo, este sistema está enfocado a juegos más complejos donde hay una gran cantidad de objetos moviéndose de forma dinámica. En nuestro caso al tratarse de un juego de tipo puzle con pocos objetos en movimiento y un movimiento muy básico sobre una matriz, resulta ineficiente utilizar este sistema de pathfinding. Por ello, utilizaremos algoritmo A* [Bourg y otros, 2004] implementado por nosotros para dotar de inteligencia a los enemigos. La implementación de este algoritmo está descrita en el Anexo 6.1.

Partiendo del algoritmo A* implementado, los enemigos se moverán por estados (*up*, *down*, *left*, *right*) al igual que el personaje y demás elementos dinámicos del escenario. La lógica para moverse será comprobar si existe un camino hasta el personaje y en caso afirmativo lo seguirá, y en otro caso se moverá de forma aleatoria. En el esquema de la figura 6.1 se detalla la lógica del enemigo.

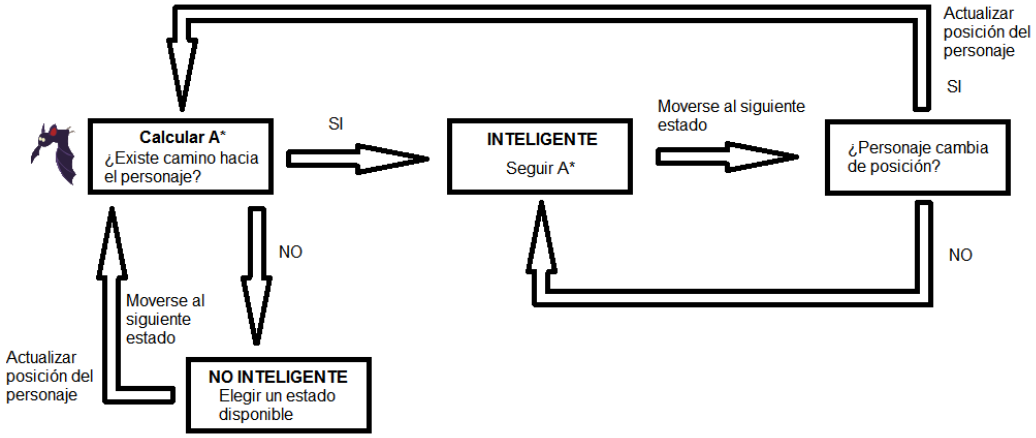


Figura 6.1. Diagrama de la lógica del enemigo.

A continuación explicaremos el código perteneciente al método "Update" que se encargará de aplicar el diagrama anterior. Los métodos auxiliares están desarrollados en el Anexo 6.2.

6. Añadiendo enemigos con inteligencia artificial.

```
if (!smart) {
	if (state.Equals(State.stop)) {
		/**
		*Implementación de código:
		*Comprobar si el jugador ha cambiado de posición
		**/

		//Comprobar si existe un camino hacia el jugador
		pathToPlayer = this.InicializateStar();

		/**
		*Implementación de código:
		*Si no existe un camino -> elige un estado disponible
		*(movimiento) de forma aleatoria.
		*
		*Si existe un camino y está a una distancia menor o igual
		*de la especificada -> Smart = true.
		**/
	}
}
```

Figura 6.2. lógica del enemigo no inteligente.

En la figura 6.2 se describe el código que se encarga de la lógica cuando el enemigo no es inteligente (será su estado inicial). Comprobamos si existe un camino independientemente de si ha cambiado la posición del jugador porque puede darse el caso que una piedra u otro objeto que obstaculizase el camino ya no lo obstaculice. Para darle más realismo y evitar que un enemigo pueda perseguir al jugador desde una punta a otra del mapa, sólo lo perseguirá si está a un número máximo de movimientos del jugador.

```
else {
	if (state.Equals(State.stop)) {
		//Comprobamos si la posición del jugador ha cambiado.
		int posRowsPlayerAux = MovePlayer.posRows;
		int posColumsPlayerAux = MovePlayer.posColumns;

		if (posRowsPlayer != posRowsPlayerAux || posColumnsPlayer
			!= posColumsPlayerAux){
				posRowsPlayer = posRowsPlayerAux;
				posColumnsPlayer = posColumsPlayerAux;

				pathToPlayer = this.InitializateAStar();
		}
		if (pathToPlayer == null) {
			smart = false;
		}
		else {
			if (pathToPlayer.Count <= numStepsChase){
				//Extraemos el primer nodo del camino:
				NodoAStar node = this.extractFirstElementPath();

				if (node == null){
					smart = false;
				}
				else {
					state = this.getStateFromPosition(node);
				}
			}
			else{
				smart = false;
			}
		}
	}
}
```

Figura 6.3. Lógica del enemigo inteligente.

En la Figura 6.3 se muestra el código de la lógica cuando el enemigo es inteligente. En este caso solo generamos un nuevo camino A* si el jugador ha cambiado de posición para que los objetos dinámicos como piedras afecten al enemigo y puedan matarlo si caen sobre él. Si sigue existiendo un camino, extraemos el siguiente nodo y calculamos el estado al que se tiene que mover.

6. Añadiendo enemigos con inteligencia artificial.

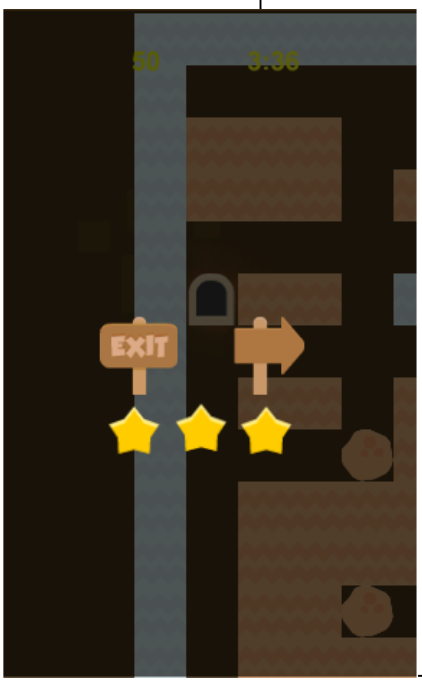
7. Lógica del juego y conectividad con redes sociales.

Este capítulo lo dedicaremos a darle una lógica al juego y a los pequeños detalles que falta para convertir el juego en un producto finalizado. En cuanto a herramientas 2D de Unity3D, solo usaremos el componente *Distance Joint 2d* pero profundizaremos en el sonido y en las redes sociales, dos componentes imprescindibles en los videojuegos a día de hoy.

7.1. Lógica del juego y uso del componente *Distance Joint 2d*.

Hemos construido un nivel pero aún nos falta implementar la lógica del juego, es decir, que el jugador gane o pierda. En el Anexo 4 están definido las diferentes formas de superar un nivel.

Cuando finalicemos el nivel, nos aparecerá la pantalla de resultado de un nivel (Figura 7.1) que cargará las variables previamente almacenadas (antes de acceder a un determinado nivel, almacenamos los datos del nivel y los requisitos para superarlo) y calculará el número de estrellas que otorgará al jugador, activando o desactivando cada estrella. Las estrellas tienen un ligero balanceo con respecto a un eje. Esto se consigue añadiéndoles a cada una un componente *Distance Joint 2D.*




```
void Update () {
       if (!initCheck && MovePlayer.gameOver) {
              initCheck = true;

              string levelActive =
                     PlayerPrefs.GetString("levelActive");
              int levelWin = PlayerPrefs.GetInt("levelWin");
              int pointsLevel = PlayerPrefs.GetInt("pointsLevel");
              int enemiesDead = PlayerPrefs.GetInt("enemiesDead");

              int minPoints = PlayerPrefs.GetInt("minPoints");
              int maxPoints = PlayerPrefs.GetInt("maxPoints");

       if (levelWin == 0) {

          if (pointsLevel >= minPoints) {
                //Cargar y activar la primera estrella:
              string nameStar1 = "star_1_" + levelActive;
              PlayerPrefs.SetInt(nameStar1, 1);
              star1.SetActive(true);

              if (pointsLevel >= maxPoints)
              {
                  //Cargar y activar la segunda estrella.
              }
              if (enemiesDead <= 0)
              {
                  //Cargar y activar la tercera estrella.
              }
       }
```

Figura 7.1. Pantalla de resultado de un nivel y su script asociado.

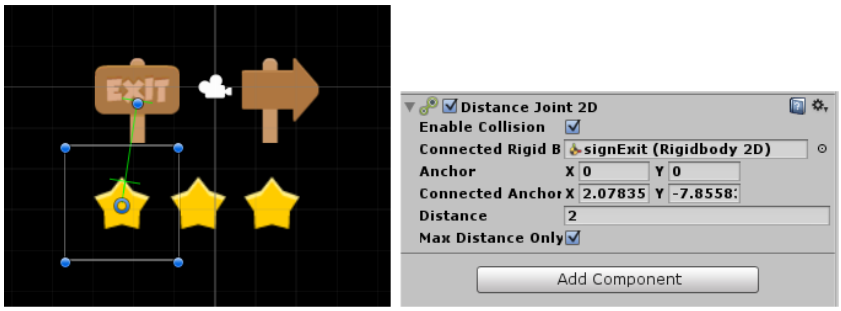


Figura 7.2. Distance Joint 2D asociado a un GameObject.

En la figura 7.2 podemos ver los campos que incluye el componente *Distance Joint 2D*. Podemos conectarlo a otro *GameObject* (con Rigid Body 2D) y activar o desactivar la colisión con dicho objeto. En "Distance" elegimos la distancia máxima entre los dos puntos y activando la última opción solo se balanceará a la distancia máxima.

7.2. Añadiendo redes sociales a nuestro juego.

El uso de las redes sociales en nuestros juegos es muy útil para que nuestro juego llegue a más gente pero hay que analizar bien su uso para que no resulte molesto al usuario. En nuestro caso añadiremos un botón de Facebook que aparecerá tras superar cada nivel y permitirá compartir la puntuación.

Dado que es una herramienta externa lo cual no está incluido en el objetivo del proyecto, en el Anexo 7.1 se comenta brevemente como añadir el SDK de Facebook para compartir una publicación y además menciona la herramienta "*Soomla*" que nos facilitará la integración con todas las redes sociales y con las compras "*in-Apps*".

8. Conclusiones.

En este proyecto se ha creado un tutorial práctico de las nuevas herramientas aparecidas en la versión 4.3 de *Unity3d* para crear contenido en 2D, a la vez que se ha implementado un videojuego completo. Para realizar esta tarea, se ha intentado que cada tema se centre en una herramienta o funcionalidad específica de la versión 4.3 a la vez que se avanza en la implementación del juego.

Aunque el tutorial se centra en dichas herramientas nativas de *Unity3d*, dado que implementar un videojuego es una tarea compleja y bastante extensa, me ha resultado imposible no incluir diversos temas alejados del propósito principal del tutorial pero que son necesarios en el desarrollo de cualquier videojuego como son:

- Herramientas de desarrollo: hemos explicado e instalado las herramientas de desarrollo. *Unity3d* es un motor gráfico basado en scripts y aunque incluya su propio IDE, disponemos de varias alternativas para programar los scripts.

- Documento de diseño del videojuego (GDD): en la implementación de un videojuego es común que trabaje un equipo de personas por lo que es necesario un documento más laxo que el documento de Especificación de Requisitos para detallar el videojuego.

- Inteligencia artificial: en cualquier juego por simple que sea, como por ejemplo el *Pong*, incluye inteligencia artificial. En este tutorial hemos implementado un algoritmo A* y hemos comentado la alternativa que ofrece *Unity3d*.

- Redes sociales: al tratarse de un videojuego enfocado a plataformas móviles, la integración con redes sociales es algo esencial por lo que se ha integrado con una de las redes sociales más conocidas, Facebook, y además se ha hecho una introducción con las demás redes sociales y plataformas.

- Otros: la importancia de la música, gráficos y otros detalles para que un videojuego tenga un acabado profesional.

Estas nuevas herramientas de *Unity3d* nos han proporcionado todo lo necesario para la implementación de un videojuego en 2D. Aunque es cierto que en todo momento trabajamos sobre un entorno en 3D adaptado al 2D no he notado ninguna caída de rendimiento en el entorno de desarrollo ni en los dispositivos que ejecutan el videojuego gracias a la potencia del hardware actual. *Unity3d* nos ha facilitado mucho la implementación del videojuego resultante, el cual está adaptado a dispositivos móviles pero es posible exportarlo a otras plataformas como videoconsolas realizando unos cambios mínimos. Gracias al uso de las escenas, es fácil añadir contenido al videojuego y mejorarlo al contrario que con *frameworks* más livianos donde hay que tener en cuenta las futuras actualizaciones a la hora de programar la arquitectura del videojuego. Por otra parte, *Unity3d* dispone de mucha comunidad que crea contenido por lo que siempre encontraremos alguna solución para cualquier cosa que necesitemos, como por ejemplo la facilidad con la que hemos integrado las redes sociales en el videojuego.

En resumen, aunque utilizar Unity3d para implementar un videojuego en 2D pueda ser como "matar moscas a cañonazos", en este tutorial hemos visto que las facilidades que nos proporciona Unity3d, su versatilidad, su comunidad detrás y la potencia del hardware actual, hace que compense utilizar este motor gráfico para este tipo de desarrollos.

8.1. Mejoras y trabajo futuro.

Como tutorial de las nuevas herramientas nativas para 2D, he abarcado la mayor parte de ellas por lo que es un tutorial práctico bastante completo. Sin embargo, durante la realización de este proyecto *Unity3d* ha lanzado varias versiones, entre ellas la 4.6 que incluye las nuevas herramientas GUI para facilitar la implementación de menús. En el videojuego elegido, no ha requerido de grandes menús por lo que no ha sido necesario utilizar las antiguas herramientas GUI incluidas desde las primeras versiones de *Unity3d*, sin embargo, dado el potencial de estas herramientas en la versión 4.6, sería interesante ampliar este tutorial con ellas.

Por otra parte, la integración con redes sociales y plataformas móviles es una parte interesante en el videojuego implementado. No forma parte del objetivo del proyecto pese a que hemos realizado la integración con la red social Facebook, sin embargo hubiese sido interesante profundizar en el *framework* open-source y gratuito "*Soomla"* el cual al ser un *framework* muy extenso, requiere mucho más tiempo de dedicación que el especificado en un proyecto fin de grado.

8.2. Aprendizaje personal.

A nivel de tecnología, el proyecto me ha servido para conocer más a fondo el motor de desarrollo *Unity3d*, tanto su interfaz y herramientas como su forma de trabajar con él. También he mejorado mis conocimientos en *C#*, lenguaje que ya conocía de antes pero a nivel de desarrollo de aplicaciones de escritorio y además adquirir buenas prácticas de programación ya que en un videojuego que requiere el uso de más recursos hardware, es indispensable utilizar buenas prácticas a la hora de programar y definir los objetos.

En cuanto a técnicas y algoritmos, con la inteligencia artificial de los enemigos he tenido que refrescar todos los conocimientos adquiridos durante la carrera y analizar que algoritmo era el más adecuado para este caso. Además de hacer uso de las buenas prácticas aprendidas como he comentado en el párrafo anterior.

A nivel de producto, me ha servido para conocer todos los entresijos del desarrollo de un videojuego y cómo intervienen muchos otros campos además de la programación. Por lo tanto, aunque en este proyecto hemos implementado un juego completo, en general un videojuego debería ser desarrollado por un grupo de personas con diferentes roles para asegurarnos de que el producto se lance con un mínimo de calidad.

8.3. Problemas técnicos.

Un problema que he tenido ha sido utilizar una herramienta de control de versiones con *Unity3d*. En mi caso he utilizado *Bitbucker*, un servicio de alojamiento de código basado en *Git* y *Sourcetree* que es un cliente *Git* que provee de una interfaz visual para utilizar los comandos de *Git*. *Unity3d* está basado en "escenas" las cuales guardan su configuración en ficheros .meta (*xml*) que identifican los objetos de la escena y otras configuraciones. Estos ficheros pueden cambiar con solo abrir una escena por lo que es difícil saber si se han realizado cambios reales. Para solucionar este problema, hay que configurar el fichero ".gitignore" para *Unity3d* (*GitHub* tiene dispone de un fichero configurado[1]) y trabajar en la medida de lo posible con objetos *Prefab* (objetos que almacenan componentes y propiedades).

Otro problema ha sido que pese a que *Unity3d* ofrece todas las herramientas necesarias, el tema de los gráficos y las animaciones requieren de un "toque artístico" del que no dispongo. La solución ha sido utilizar recursos gratuitos en la medida de lo posible y realizar animaciones sencillas.

Por otra parte, a nivel personal he tenido que compaginar el proyecto con un trabajo a jornada completa por lo que el tiempo de desarrollo del proyecto se ha alargado más tiempo de lo previsto.

1. Fichero de configuración en Github: https://github.com/github/gitignore/blob/master/Unity.gitignore

8. Conclusiones.

Referencias bibliográficas.

Unity3d.

[Unity3d, 2015] Recursos oficiales de Unity3d (Accedido en Septiembre del 2015).
Documentación: http://docs.unity3d.com/es/current/Manual/UnityManualRestructured.html
Tutoriales: https://unity3d.com/es/learn/tutorials
Foros: https://unity3d.com/es/community

[Asset Store, 2015] Tienda oficial de Unity3d (Accedido en Septiembre del 2015).
https://www.assetstore.unity3d.com/en/

[Kyaw y otros, 2013] Aung Sithu Kyaw y Thet Naing Swe, Unity 4.x Game AI Programming, Editorial Packt, 2013.

[Dave Calabrese, 2014] Dave Calabrese, Unity 2D Game Development, Editorial Packt, 2014.

[Sue Blackman, 2013] Sue Blackman, Beginning 3D Game Development with Unity 4, Editorial Apress, 2013.

Tecnologías.

[Cocos2d-X, 2015] Web oficial de Cocos2d-X (Accedido en Septiembre del 2015).
http://www.cocos2d-x.org/

[DirectX11, 2015] Web oficial de DirectX 11 (Accedido en Septiembre del 2015).
http://www.microsoft.com/en-us/download/details.aspx?id=17431

[Jesse Liberty, 2002] Jesse Liberty, Learning C#, Editorial O'Reilly, 2002.

[LibGDX, 2015] Web oficial de LibGDX (Accedido en Septiembre del 2015).
https://libgdx.badlogicgames.com/

[Monodevelop, 2015] Web oficial de Monodevelop (Accedido en Septiembre del 2015).
http://www.monodevelop.com/

[OpenGL, 2015] Web oficial de OpenGL (Accedido en Septiembre del 2015).
https://www.opengl.org/

[Shelley Powers, 2008] Shelley Powers, Learning Javascript, Editorial O'Reilly, 2008.

[Unreal Engine 4, 2015] Web oficial de Unreal Engine 4 (Accedido en Septiembre del 2015).
https://www.unrealengine.com/what-is-unreal-engine-4

Algoritmos.

[Antonio J. Fernández, Universidad de Málaga] Antonio J. Fernández, Asignatura de Análisis y Diseño de Algoritmos.

[Bourg y otros, 2004] David M.Bourg y Glenn Seeman, AI for Game Developers, Editorial O'reilly, 2004.

[Mandow Andaluz Lorenzo, Universidad de Málaga] Mandow Andaluz Lorenzo, Asignatura de Sistemas Inteligentes.

Gestión del proyecto.

[Bitbucker] Repositorio de código basado en Git (Accedido en Septiembre del 2015).
https://bitbucket.org/

[Razón Artificial] Documento de diseño de videojuegos (Accedido en Septiembre del 2015).
http://razonartificial.com/2010/12/documento-de-diseno-de-videojuegos/

[SourceTree] Cliente Git con una interfaz visual (Accedido en Septiembre del 2015).
https://www.sourcetreeapp.com/

Redes sociales.

[Facebook Developers] SDK de Facebook para Unity3d y documentación oficial (Accedido en Septiembre del 2015).
https://developers.facebook.com/docs/unity

[Repositorio Soomla] Repositorio Soomla, framework open source (Accedido en Septiembre del 2015).
https://github.com/soomla

[Soomla] Framework Soomla y documentación oficial (Accedido en Septiembre del 2015).
http://know.soom.la/unity/

Recursos gratuitos.

[Beatlab] Web para crear melodías de forma gratuita (Accedido en Septiembre del 2015).
http://www.beatlab.com/

[Bfxr] Web para efectos de sonido gratuitos (Accedido en Septiembre del 2015).
http://www.bfxr.net/

[Kenney] Artista gráfico que proporciona recursos gratuitos y de pago (Accedido en Septiembre del 2015).
http://opengameart.org/users/kenney

Anexo técnico.

1. Instalación de la herramienta *Unity3d*.

La descarga de la herramienta la realizaremos a través de la página web oficial[1]. La página muestra la información de forma clara y está completamente localizada al inglés y al español entre otros idiomas, por lo que es fácil navegar a través de ella para encontrar lo que buscamos. Nos aparecerá las dos versiones actualmente disponibles: *Unity3d* 5 *Personal* y *Professional*. Si buscamos una versión anterior en concreto, abajo en el pie de página encontraremos varios enlaces, entre ellos uno para acceder al archivo[2] que contiene las anteriores versiones de *Unity3d*. Como hemos comentado anteriormente, la diferencia entre la *Personal* y la *Professional* radica principalmente en las herramientas de soporte que proporciona ésta última como se muestra en la Figura 1.1, por tanto, descargamos la versión *Personal*.

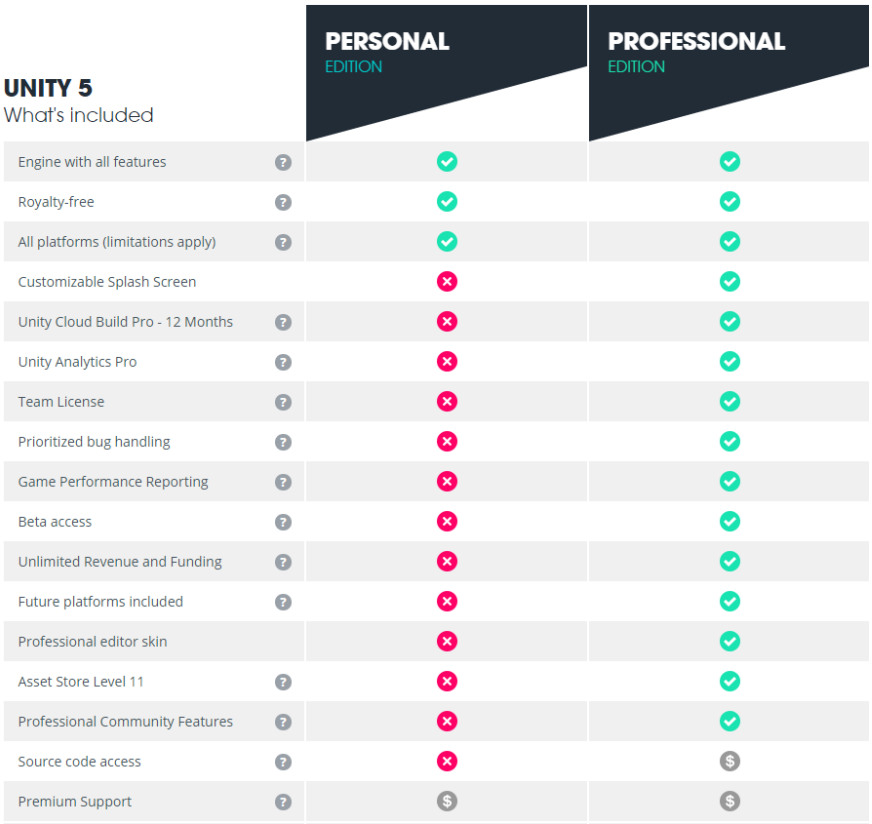
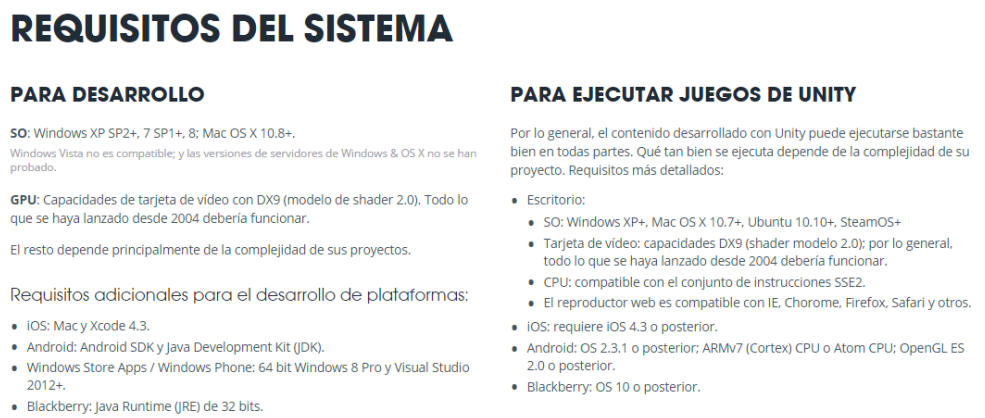


Figura 1.1. Diferencia entre versiones. Para más información visitar la web oficial: http://unity3d.com/get-unity

1. Página oficial de Unity3d a fecha 2015/07/15: http://unity3d.com/es/get-unity
2. Versiones anteriores de Unity3d a fecha 2015/07/15: http://unity3d.com/es/get-unity/download/archive



REQUISITOS DEL SISTEMA

PARA DESARROLLO

SO: Windows XP SP2+, 7 SP1+, 8; Mac OS X 10.8+.
Windows Vista no es compatible; y las versiones de servidores de Windows & OS X no se han probado.

GPU: Capacidades de tarjeta de vídeo con DX9 (modelo de shader 2.0). Todo lo que se haya lanzado desde 2004 debería funcionar.

El resto depende principalmente de la complejidad de sus proyectos.

Requisitos adicionales para el desarrollo de plataformas:

- iOS: Mac y Xcode 4.3.
- Android: Android SDK y Java Development Kit (JDK).
- Windows Store Apps / Windows Phone: 64 bit Windows 8 Pro y Visual Studio 2012+.
- Blackberry: Java Runtime (JRE) de 32 bits.

PARA EJECUTAR JUEGOS DE UNITY

Por lo general, el contenido desarrollado con Unity puede ejecutarse bastante bien en todas partes. Qué tan bien se ejecuta depende de la complejidad de su proyecto. Requisitos más detallados:

- Escritorio:
  - SO: Windows XP+, Mac OS X 10.7+, Ubuntu 10.10+, SteamOS+
  - Tarjeta de vídeo: capacidades DX9 (shader modelo 2.0); por lo general, todo lo que se haya lanzado desde 2004 debería funcionar.
  - CPU: compatible con el conjunto de instrucciones SSE2.
  - El reproductor web es compatible con IE, Chorome, Firefox, Safari y otros.
- iOS: requiere iOS 4.3 o posterior.
- Android: OS 2.3.1 o posterior; ARMv7 (Cortex) CPU o Atom CPU; OpenGL ES 2.0 o posterior.
- Blackberry: OS 10 o posterior.

Figura 1.2. Requisitos necesarios para instalar Unity3d y ejecutar contenidos desarrollados con la herramienta.

Una vez descargado el instalador, lo ejecutamos y seguimos los pasos. Nos pedirá aceptar el acuerdo de licencia y seguidamente seleccionar los componentes que queremos instalar. Seleccionaremos los tres primeros componentes (la herramienta *Unity3d* 5, el plugin *Web Player* para exportar y ver el contenido generado en el navegador web y los "assets" estándar) y dejaremos a elección del lector los proyectos de ejemplo porque ocupan mucho espacio en el disco duro. Por último, seleccionaremos la localización de la instalación y de la descarga de los ficheros temporales de instalación ya que el contenido de este motor gráfico no viene incluido dentro del instalador, sino que se descarga a través de él.

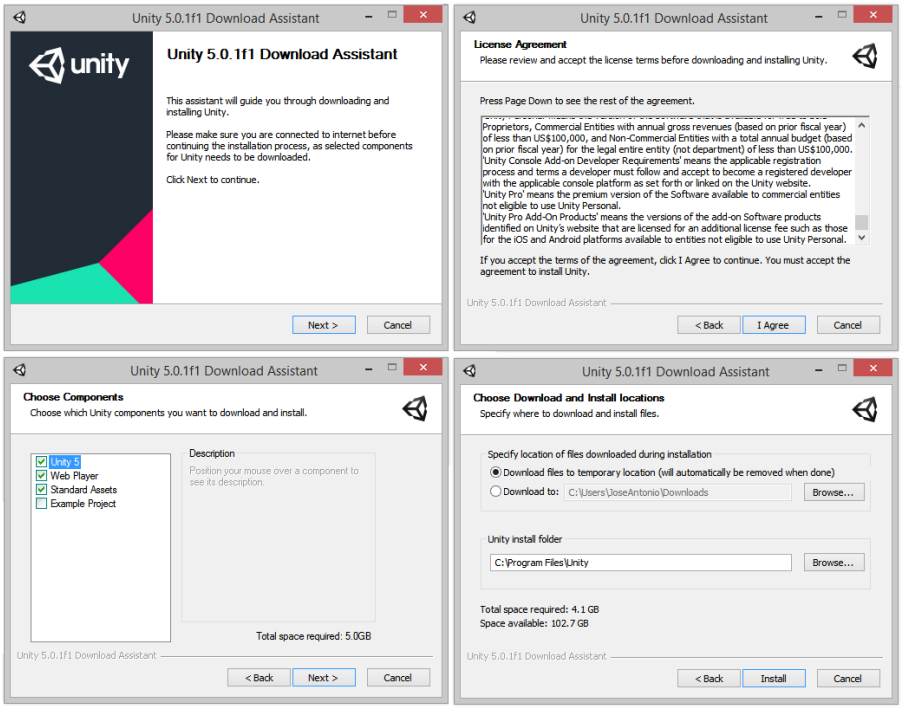


Figura 1.3. Pasos de la instalación de Unity3d.

Una cosa que llama la atención, es que tanto para la descarga del instalador de *Unity3d* como para su instalación, no ha sido necesario realizar ningún registro en su web. El registro solo es necesario si queremos instalar la versión Professional o si queremos acceder a la tienda de "assets".

Una vez instalado, ya podemos empezar a utilizar la herramienta. *Unity3d* trae integrado el IDE *Monodevelop* que es libre, gratuito y está enfocado para entornos *.NET* (*C#*) aunque también es utilizable con otros lenguajes. Aunque este IDE funciona bien, muchos desarrolladores prefieren utilizar *Microsoft Visual Studio* ya que es más estable y proporciona algunas herramientas adicionales. Por lo tanto, antes de empezar a utilizar *Unity3d*, instalaremos *Visual Studio* y lo integraremos en *Unity3d*.

2. Instalación de Microsoft Visual Studio.

Para descargarnos *Microsoft Visual Studio*, accederemos a su página web oficial[1]. Al igual que con la herramienta *Unity3d*, disponemos de una versión gratuita y otra de pago. Nos descargamos la versión gratuita "Visual Studio Community" puesto que es perfectamente utilizable y no tiene ningún tipo de restricción en cuanto a la utilización que le vamos a dar. Un requisito de esta versión es disponer de *Windows 8* u otro sistema operativo posterior de *Microsoft*, por lo que en el caso de no cumplir este requisito, accedemos a la pestaña "descargas" de la web y nos descargamos la versión posterior más reciente que es la *Express 2013*.

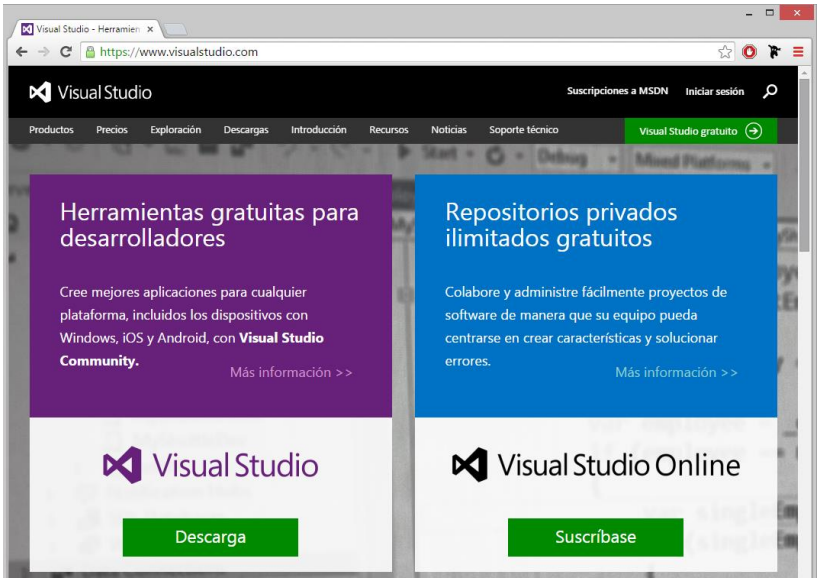


Figura 2.1. Página de descarga de Microsoft Visual Studio.

1. Página oficial de *Microsoft Visual Studio* a fecha 2015/07/15: https://www.visualstudio.com/

Una vez descargado el instalador, lo ejecutamos. Si en nuestra máquina no tenemos instalado otras herramientas de Microsoft, es posible que nos avise mediante un error de que no tenemos instalado "Microsoft .Net Framework 4.5" y/o el paquete "Microsoft Visual C++ 2005". En dicho caso, accedemos al centro de descargas de Microsoft[1] descargar e instalar los componentes que nos pide. Una vez superado este pequeño problema, aceptamos el acuerdo de licencia que nos muestra la Figura 2.2 y seguimos con la instalación.

Figura 2.2. Primera ventana del instalador de Visual Studio.

En la siguiente ventana (Figura 2.3) nos pide elegir las herramientas opcionales a instalar. Aunque en un principio no son necesarias, es recomendable dejarlo por defecto porque *Visual Studio* es un IDE muy potente y quizás en otros proyectos necesitemos algunas de dichas herramientas. Al igual que pasa con el instalador de *Unity3d*, el contenido necesario para *Visual Studio* se descarga e instala a través de este instalador. Tras la instalación, aunque esta versión es gratuita, nos pedirá una dirección de correo para tener registrado el producto.

1. Centro de descargas de *Microsoft* a fecha 2015/07/15: http://www.microsoft.com/es-es/download/

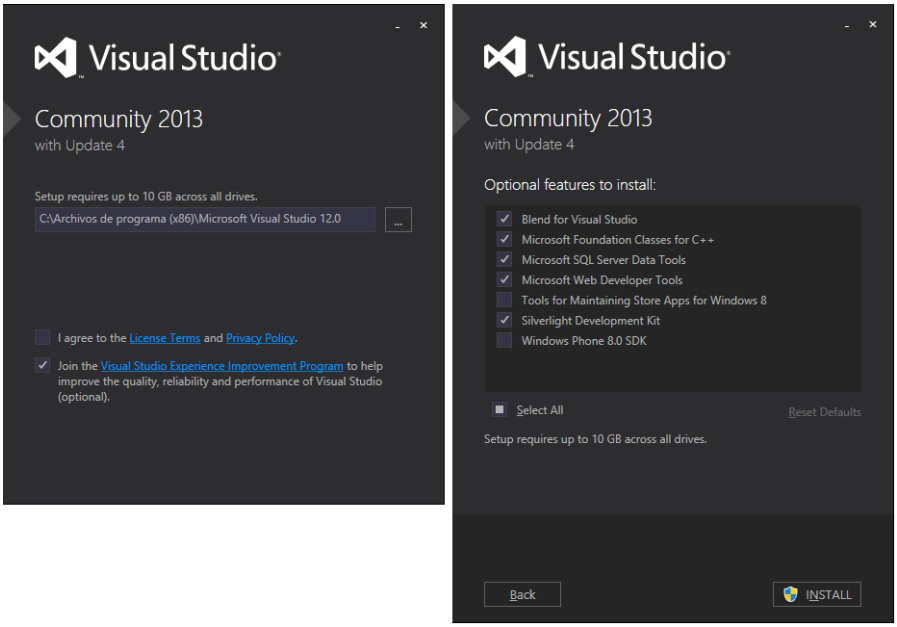


Figura 2.3. Instalación de Visual Studio.

Tras esto, deberíamos poder integrar Visual Studio con *Unity3d*. Para ello, desde *Unity3d* con un proyecto abierto, vamos a "*Assets/ Import Package/ Visual Studio 2013 Tools*" para añadir el paquete necesario para integrar *Visual Studio* con *Unity3d* tal y como se muestra en la Figura 2.4.

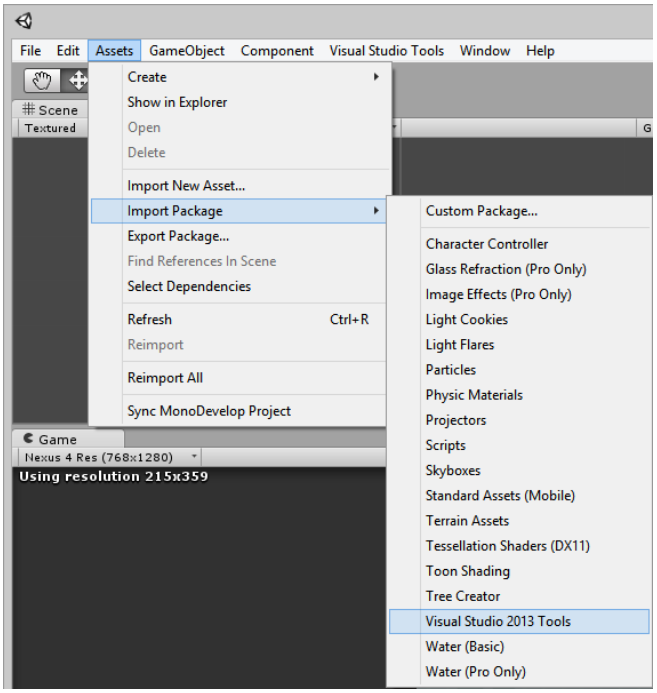


Figura 2.4. Añadir el paquete necesario para integrar Visual Studio con Unity3d.

Tras añadir el paquete, vamos a "*Edit/ Preferences*" (Figura 2.5) y se abrirá una pequeña ventana con las preferencias generales de Unity. Vamos a la opción "External Tools" y como editor elegimos "UnityVS.OpenFile". Cerramos la ventana y ya tendremos enlazado las dos herramientas por lo que cada vez que abramos un script, éste se abrirá con *Visual Studio*.

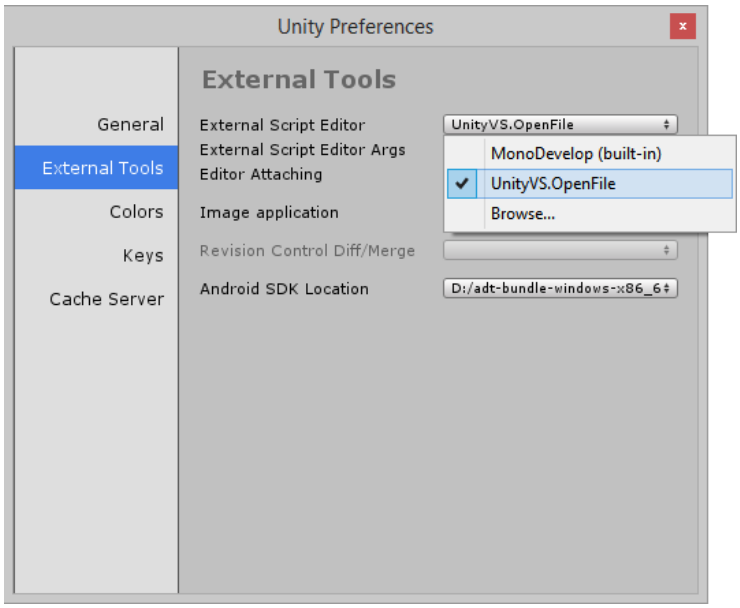


Figura 2.5. Ventana de preferencias de Unity3d.

Es posible que no aparezca la opción "UnityVS.OpenFile" en el desplegable de elegir editor. La causa de ello puede ser por haber elegido rutas de instalación distintas de las rutas por defecto con *Unity3d* y/o *Visual Studio*, o haber instalado una versión posterior de alguna de dichas herramientas. Para solucionarlo volvemos al centro de descargas y buscamos "Visual Studio 2013 Tools for Unity" y lo instalamos.

3. Instalación de un editor liviano de ficheros.

Aunque con *Unity3d* y *Visual Studio* (o *Monodevelop*) tenemos lo necesario para empezar con el desarrollo y editar nuestros scripts, es recomendable un editor liviano para ediciones rápidas de scripts, ficheros de carga o simplemente para visualizar de forma rápida algún fichero de nuestro proyecto. Para estas tareas podemos utilizar un editor como *Sublime Text*[1] o *Notepad++*[2]. Ambas opciones disponen de versiones instalables o autoejecutables sin necesidad de instalación.

1. Página oficial de Sublime Text a fecha 2015/07/15: http://www.sublimetext.com/
2. Versiones anteriores de Notepad++ a fecha 2015/07/15: http://notepad-plus-plus.org

4. Documento de diseño de nuestro videojuego.

En el capítulo 1 hemos introducido el Documento de diseño de un videojuego o GDD. El tamaño del documento depende entre otras cosas del proyecto y no existe una plantilla estándar a seguir, aun así, debería incluir un mínimo de apartados que detallen las características del juego. A continuación se describe un breve documento GDD para nuestro videojuego.

- Concepto del juego.

  "Dig Boy" es un videojuego en el que controlamos a un minero que debe recoger diamantes por los distintos mundos mientras mueve piedras y evita a los enemigos. Es una remasterización y adaptación a los tiempos actuales del videojuego "*Boulder Dash*" publicado para una antigua videoconsola, la *Commodore 64*.

- Género.

  El juego pertenece al género de puzles. El jugador navegará a través de escenarios en 2D recolectando diamantes y moviendo piedras para acceder a los diamantes o para acabar con los enemigos.

- Características principales.

  El juego se basa en los siguientes pilares:

  - Planteamiento sencillo: la historia del juego es muy simple pero lo suficiente explicita para darle contexto al juego y que el jugador tenga un objetivo.

  - Partidas rápidas: cada partida no requiere de un tiempo excesivo por lo que el juego es idóneo para jugar en cualquier sitio.

  - Ampliable: el contenido del juego será independiente del motor gráfico por lo que se podrá generar nuevos contenidos como nuevos niveles o enemigos de forma sencilla.

- Público objetivo.

  El juego va dirigido a jugadores casuales de un amplio rango de edad que suelen jugar partidas cortas mientras esperan el autobús o cuando tienen un hueco libre.

- Estilo visual.

  Dig Boy tendrá un estilo sencillo y simpático haciendo uso de gráficos vectoriales.

- Mecánicas y jugabilidad.

  El juego tendrá una jugabilidad muy simple. La movilidad del personaje que controlamos se limita a izquierda, derecha, arriba y abajo, y con ello debemos buscar los diamantes y mover las piedras para evitar ser alcanzados por los enemigos. Los controles estarán adaptados a pantalla táctiles, sin tener una ubicación fija, es decir, el juego deberá comprobar la pulsación del usuario y determinar hacía que dirección quiere mover el personaje.

  En cuanto a los niveles, habrá varios mundos y cada uno con varios niveles, los cuales se desbloquearán tras superar con éxito el nivel anterior. Para superar un nivel, hay que recoger un determinado número de diamantes en un determinado tiempo que viene especificado para cada nivel. Además, para fomentar la rejugabilidad, en cada nivel superado aparecerá un máximo de tres estrellas, una por cada requisito:

  - Superar el objetivo básico del nivel: conseguir una determinada puntuación recogiendo un determinado número de diamantes en un determinado tiempo.

  - Superar el objetivo básico del nivel además de conseguir una puntuación aún mayor.

  - Superar el objetivo básico del nivel en un tiempo además de eliminar todos los enemigos del nivel.

- Flujo del juego.

Pasos a seguir desde que el jugador inicia el juego hasta que supera un nivel completo:

1. El jugador inicia el juego y se presenta una pantalla de presentación con una animación y el botón “jugar”.

2. En el caso de ser la primera vez que se inicia el juego, aparecerá una pantalla con un teclado virtual para escribir nuestro “*nick*” como jugador.

3. La siguiente pantalla es el menú principal donde se elige los mundos, los cuales cada uno tendrán diferentes niveles. Los mundos disponibles se organizan en base a una imagen de un planeta y el personaje se desplaza hasta él, apareciendo un botón con el nombre del mundo para acceder a él. Inicialmente solo hay un mundo disponible y los siguientes mundos se desbloquearán tras superar con éxito todos los niveles de los mundos previos.

4. Seguidamente se muestra una breve pantalla de carga mientras carga los niveles del mundo elegido y después la pantalla de niveles. Al igual que con los mundos, es necesario superar el nivel previo para desbloquear el siguiente nivel. Por cada nivel superado, se montará el número de estrellas conseguidas.

5. Tras seleccionar un nivel, accederemos a ese nivel en concreto. Si lo superamos, nos aparecerá una pantallita con las estrellas conseguidas y un botón para volver a la pantalla de selección de niveles. Si no lo superamos, aparecerá dos botones: uno para volver a intentarlo y otro para volver a la pantalla de selección de niveles.

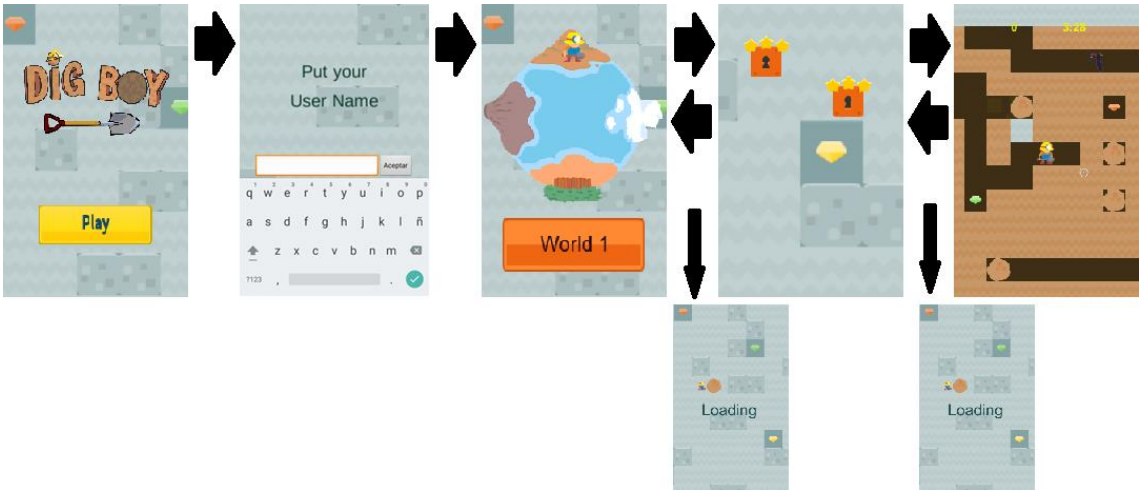


Figura 4.1. Diagrama de las pantallas de nuestro juego.

- Personajes.
    - Protagonista: es el personaje que controlamos, se mueve en las cuatro direcciones y tiene la capacidad de empujar rocas.
    - Enemigos: en esta versión solo existe un tipo de enemigo que a su vez se divide en dos:
        - Enemigo no inteligente: sigue un camino aleatorio y no persigue al personaje. Este enemigo aparecerá en los primeros niveles.
        - Enemigo inteligente: persigue al personaje siempre que exista un camino directo y esté a determinada distancia. En caso de no cumplirse esos requisitos, seguirá un camino aleatorio. Este personaje aparecerá en los niveles más avanzados.
- Objetos.
    - Diamantes: son los objetos que debe recolectar el personaje y que otorgarán una determinada puntuación según el color de éstos.

- Piedras: estos objetos no son recolectables pero los usaremos para empujarlos y así movernos por la pantalla o eliminar a los enemigos.

- Audio:

  - Música principal: en todas las pantallas, excepto en los niveles, se reproducirá la misma melodía que debe ser alegre y relajada.

  - Música niveles: será una melodía animada que provoque al jugador una sensación de tensión sin que llegue a ser molesta.

  - Superar un nivel: sonará una breve música alegre para que el jugador se sienta recompensado.

  - Game Over en un nivel: música breve que evocará a derrota pero que anime a jugar de nuevo al nivel.

  - Botones de las pantallas: se reproducirá un breve sonido que indiquen que han sido pulsados.

5. Implementación de la pantalla del menú principal.

En el capítulo 4 vimos cómo utilizar la herramienta *Animator* para crear un diagrama de estados para las animaciones y cómo navegar a través de los estados a nivel de código. En este apartado del anexo terminaremos de implementar la pantalla y su lógica.

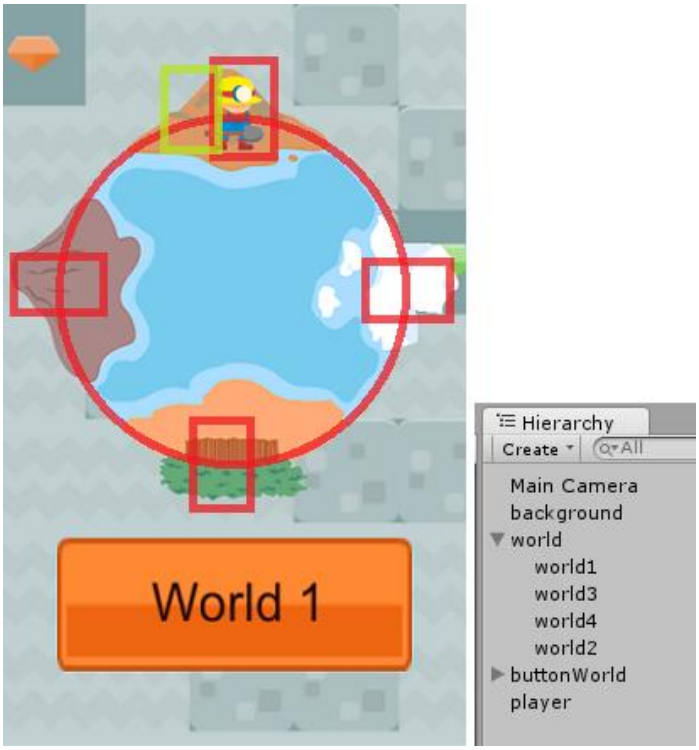


Figura 5.1. Diferentes GameObjects de la pantalla del menú principal.

En la Figura 5.1 se muestra los diferentes *GameObject* de la pantalla del menú principal. El planeta, representado como el *GameObject* "world" contiene su textura y además también contiene cuatro hijos sin ninguna textura asociada. El giro del planeta podríamos calcularlo matemáticamente pero lo realizaremos mediante colisiones para así utilizar las funciones de *Unity3d* para 2D como la función "OnMouseDown" vista en el capítulo 3 o la función "OnTriggerStay2D".

Cuando pulsemos sobre uno de los mundos, el planeta girará hasta que dicho mundo colisione con el personaje. Como los mundos son hijos del planeta, éstos también girarán al heredar sus transformaciones. Para que los mundos puedan detectar las colisiones, deben tener marcado sus "Box Collider 2D" como "trigger", es decir, disparadores. Además, también será necesario añadir un "RigidBody 2D" como se puede apreciar en la Figura 5.2.

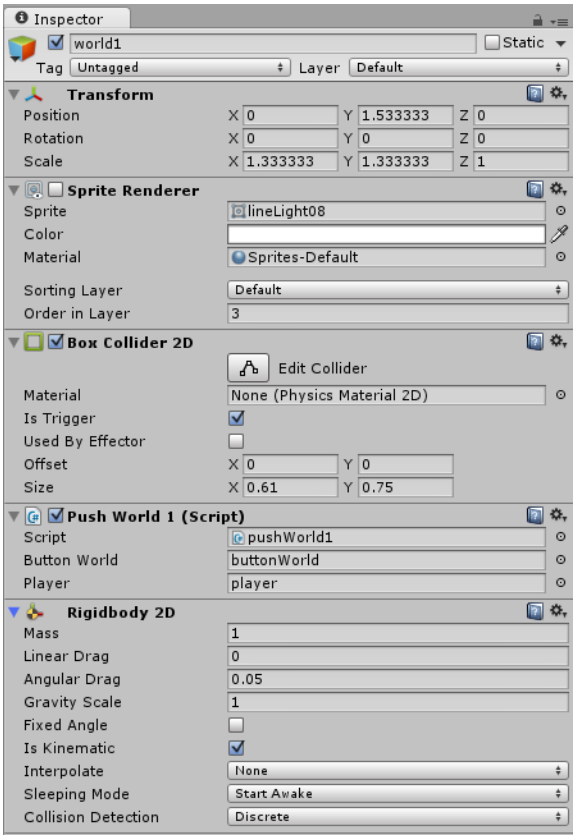


Figura 5.2. Inspector de un "mundo" con los diferentes componentes agregados.

En el planeta tendremos asociado el siguiente script de la Figura 5.3 con varias variables estáticas globales que controlarán su movimiento desde otros scripts.

```
public class RotateWorld : MonoBehaviour {

	public enum StateWorld{
	  world1, world2, world3, world4, stop
	};

	public static StateWorld stateWord;
	public static StateWorld lastStateWorld;
	private float speed;

	void Start () {
		speed = 120f;
		stateWord = StateWorld.stop;
	}

	void Update () {
		if (!stateWord.Equals(StateWorld.stop)) {
			this.transform.Rotate(new Vector3(0, 0, speed
				* Time.deltaTime));
		}
	}
}
```

Figura 5.3. Script asociado al planeta para controlar su giro.

Guardaremos un estado para conocer en qué mundo estamos, lo cual es necesario para establecer la lógica del botón para acceder a los niveles de un determinado mundo.

```
public GameObject buttonWorld;
public GameObject player;

private bool activateButton;
private string text;

void Start () {
	activateButton = false;
	text = "World 1";
}

void OnMouseDown(){
	if (RotateWorld.stateWord.Equals(RotateWorld.StateWorld.stop)
		&& !RotateWorld.lastStateWorld.Equals(RotateWorld.StateWorld.world1)){
		RotateWorld.stateWord = RotateWorld.StateWorld.world1;
		RotateWorld.lastStateWorld = RotateWorld.StateWorld.world1;
		activateButton = true;
		buttonWorld.SetActive(false);
		Animator animatorPlayer = player.GetComponent<Animator>();
		animatorPlayer.SetBool("isWalk", true);
	}
}

void OnTriggerStay2D(Collider2D other) {
	if (activateButton) {
		activateButton = false;
		RotateWorld.stateWord = RotateWorld.StateWorld.stop;
		buttonWorld.SetActive(true);
		GameObject textWorld =
			buttonWorld.transform.FindChild("textWorld").gameObject;
		textWorld.GetComponent<TextMesh> ().text = text;
		Animator animatorPlayer = player.GetComponent<Animator>();
		animatorPlayer.SetBool("isWalk", false);
	}
}
```

Figura 5.4. Script asociado a un "mundo".

En el script de la Figura 5.4 se pueden ver dos funciones. La primera, "OnMouseDown", comprueba cuando pulsamos sobre dicho objeto y activa el giro del planeta y el estado "walking" de nuestro personaje. La segunda función, "OnTriggerStay2D", se dispara cuando un *GameObject* con un "collider" toca al objeto. Esta función se dispararía en el momento en el que un *GameObject* entrase dentro del objeto. En este caso, cuando el personaje toca el mundo, se dispara la función y desactiva el giro del planeta, guarda el estado en el que se encuentra, cambia el estado de nuestro

personaje a “resting” y activa el botón con el texto del mundo. Para que la colisión con el personaje se produzca en el centro de éste, en su “Box Collider 2D” se modifica la propiedad “offset” con un valor negativo para que se quede ligeramente a su izquierda como se puede apreciar el recuadro verde de la Figura 5.5.

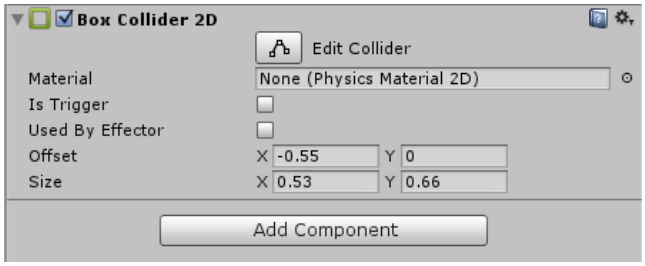


Figura 5.5. Box Collider 2D del personaje con el Offset con un valor negativo.

En el botón para ir a un nivel controlaremos su pulsación y según en el mundo en el que estemos, cargaremos un nivel u otro.

6. Profundizando en la inteligencia artificial de los enemigos.

Este apartado del Anexo lo dividiremos en dos partes: la primera sobre la implementación del algoritmo A* y la segunda parte la implementación de los métodos auxiliares que nos faltó por desarrollar en el capítulo 6.

6.1. Implementación del algoritmo A*.

El algoritmo A* es un algoritmo de búsqueda o "*pathfinding*" que se utiliza para calcular los caminos mínimos en una red. Es un algoritmo heurístico ya que utiliza una función heurística para calcular el camino más óptimo entre dos nodos.

f(n) = g(n) + h(n) donde:

g(n): es el coste real del camino para llegar al nodo actual (n) desde el nodo inicial.

h(n): representa el valor heurístico desde el nodo actual (n) hasta el nodo final.

Este algoritmo es fácil de implementar pero requiere de un elevado coste computacional. En nuestro caso al tratarse de escenarios (matrices) relativamente pequeños, ese coste es inapreciable con el hardware actual, sin embargo la complejidad será exponencial con respecto al espacio requerido para realizar la búsqueda por lo que en juegos con escenarios muy grandes y complejos, es recomendable buscar otras alternativas como la navegación por *NavMesh* que incluye *Unity3d*. A continuación detallamos la implementación en C# para *Unity3d*.

```
public class NodoAStar : IComparable{

	private int x;
	private int y;

	private Boolean isTransitable;
	private int cost;

	//F(n) = g(n) + h(n)
	private int F;
	private int G; //valor desde el nodo al nodo inicial.
	private int H; //valor desde el nodo al nodo final.

	private NodoAStar nodeFather;
	private NodoAStar nodeFinal;

	public void recalculateG(){
		G = nodeFather.G;
		if (x == nodeFather.x || y == nodeFather.y) {
			G += 10;
		}
		Else   {
			G += 14;
		}
		recalculateF();
	}

	private void recalculateH(){
		if (nodeFinal != null) {
			H = (Math.Abs(x - nodeFinal.x) + Math.Abs(y - nodeFinal.y))
				* 10;
		}
		else{
			H = 0;
		}
		recalculateF();
	}
```

Figura 6.1. Implementación de la clase NodoAStar.

Lo primero es crear la clase "NodoAStar" (Figura 6.1) que contendrá la función de evaluación de un nodo en concreto. En nuestra matriz todos los nodos tendrán el mismo coste (0) y los nodos donde existan otros objetos (tierra, piedras, diamantes) no serán transitables (*isTransitable = false*). Tendrá métodos para calcular las funciones F, G y H.

```
Public class Deap{
      Private ArrayList elements; //elementos del montículo.
```

Figura 6.2. Implementación de la clase Deap.

La clase "Deap" o "monticulo" de la Figura 6.2 representa una lista de nodos. La utilizaremos para guardar la lista de nodos descartados, la de nodos disponibles y la lista final con el camino más óptimo.

```
public class Astar {

      private NodoAStar[,] matrix;

      private NodoAStar nodeInit;
      //nodeInit no tiene porqué ser transitable y no tiene ningún coste

      private NodoAStar nodeFinal;//debe ser transitable

      private bool forceEightConnect;//indica si las diagonales son transitables.

      public Astar(NodoAStar[,] matrix, NodoAStar nodeInit, NodoAStar nodeFinal,
            bool forceEightConnect)
      {
            this.matrix = matrix;
            this.nodeInit = nodeInit;
            this.nodeFinal = nodeFinal;
            this.forceEightConnect = forceEightConnect;
      }
```

Figura 6.3. Implementación de la clase Astar.

Por último implementamos la clase "Astar" que contendrá el algoritmo A*. Como se puede ver en la Figura 6.3, esta clase recibirá la matriz de nodos, el nodo inicial, el nodo final y un booleano para indicarle si utilizaremos las diagonales. En nuestro juego no nos podemos mover diagonalmente por lo que este parámetro será falso.

```
public ArrayList calculatePath(){

	Deap listOpen = new Deap();
	ArrayList listClose = new ArrayList();
	NodoAStar nodeAct = null;
	bool pathIsFound = false;

	int rows = matrix.Length;
	int columns = 0;
	if (rows > 0)
		columns = matrix.GetLength(0);

		listOpen.push(nodeInit); //Añade el nodo inicial a listOpen

		int iterations = 0;

		// Buscar el camino mientras queden nodos:
		while (!listOpen.isEmpty() && !pathIsFound) {
			iterations++;

			//Extrae el nodo con el F más pequeño de listOpen a listClose.
			nodeAct = (NodoAStar)listOpen.popBottom();
			listClose.Add(nodeAct);
```

Figura 6.4. Primera parte de la función que calcula el camino más óptimo.

Para calcular el camino más óptimo (Figura 6.4), lo primero es tener una lista con los nodos disponibles (“listOpen”) y otra lista con los nodos descartados (“listClose”). Mientras queden nodos disponibles y no se haya encontrado el camino, extraeremos nodos de la primera lista.

```
//Extrae el nodo adyacente del nodo actual.
ArrayList nodeAdjacentList = new ArrayList();

if (0 <= nodeAct.getX() + 1 && nodeAct.getX() + 1 < columns
       && 0 <= nodeAct.getY() && nodeAct.getY() < rows) {
       if (matrix[nodeAct.getY(),nodeAct.getX() + 1].getIsTransitable()){
              nodeAdjacentList.Add(matrix[nodeAct.getY(),nodeAct.getX() + 1]);
       }
}
if (0 <= nodeAct.getX() - 1 && nodeAct.getX() - 1 < columns
       && 0 <= nodeAct.getY() && nodeAct.getY() < rows) {
       if (matrix[nodeAct.getY(),nodeAct.getX() - 1].getIsTransitable()){
              nodeAdjacentList.Add(matrix[nodeAct.getY(),nodeAct.getX() - 1]);
       }
}
if (0 <= nodeAct.getX() && nodeAct.getX() < columns
       && 0 <= nodeAct.getY() - 1 && nodeAct.getY() - 1 < rows) {
       if (matrix[nodeAct.getY() - 1,nodeAct.getX()].getIsTransitable()){
              nodeAdjacentList.Add(matrix[nodeAct.getY() - 1,nodeAct.getX()]);
       }
}
if (0 <= nodeAct.getX() && nodeAct.getX() < columns
       && 0 <= nodeAct.getY() + 1 && nodeAct.getY() + 1 < rows) {
       if (matrix[nodeAct.getY() + 1,nodeAct.getX()].getIsTransitable()){
              nodeAdjacentList.Add(matrix[nodeAct.getY() + 1,nodeAct.getX()]);
       }
}
```

Figura 6.5. Segunda parte de la función que calcula el camino más óptimo.

En la segunda parte del algoritmo (Figura 6.5), del nodo actual que hemos extraído de “listOpen”, extraemos sus nodos adyacentes y los insertamos en otra lista a parte.

```
//Por cada nodo encontrado, comprobamos si es el nodo final.
while (nodeAdjacentList.Count > 0 && !pathIsFound){

       NodoAStar nodeAdjacent = (NodoAStar)nodeAdjacentList[0];
       nodeAdjacentList.Remove(nodeAdjacent);

       if (!listClose.Contains(nodeAdjacent)) {
              if (!listOpen.contains(nodeAdjacent)) {
                     nodeAdjacent.setNodeFather(nodeAct);
                     listOpen.push(nodeAdjacent);

                     if (nodeFinal == nodeAdjacent)
                            pathIsFound = true;
              }
              else{
                     int newG = nodeAct.getG();
                     if (nodeAdjacent.getX() == nodeAct.getX()
                            || nodeAdjacent.getY() == nodeAct.getY()){
                            nuevoG += 10;
                     }
                     else{
                            newG += 14;
                     }

                     if (newG < nodeAdjacent.getG()){
                            nodeAdjacent.setNodeFather(nodeAct);
                            listOpen.reorder();
                     }
              }
       }
}
```

Figura 6.6. Tercera parte de la función que calcula el camino más óptimo.

Lo siguiente (Figura 6.6) será comprobar si cada nodo adyacente está en la lista abierta y en caso contrario, insertarlo en ella enlazándolo con su nodo padre. En caso de que esté en la lista cerrada, comprobaremos si su valor G es menor que el nodo actual y en ese caso lo volvemos a añadir a la lista abierta.

```
//Si hemos encontrado el camino, hacemos "backtracking" desde el final al inicial
if (pathIsFound) {
	ArrayList path = new ArrayList();
	NodoAStar nodoAuxiliar = nodeFinal;

	while (nodoAuxiliar != null){
		path.Insert(0, nodoAuxiliar);
		nodoAuxiliar = nodoAuxiliar.getNodeFather();
	}
	return path;
}
else
	return null;
```

Figura 6.7. Cuarta parte de la función que calcula el camino más óptimo.

Una vez encontrado el camino (Figura 6.7), hacemos "backtracking" desde los nodos padre para devolver el camino más óptimo.

```
public ArrayList InitializateAStar(){
	NodoAStar[,] matriz = new NodoAStar[ControllerLevel.limitRows,
		ControllerLevel.limitColumns];

	//Generar matriz:
	for (int i = 0; i < ControllerLevel.limitRows; i++){
		for (int j = 0; j < ControllerLevel.limitColumns; j++){

			GameObject ob = ControllerLevel.arraySquare[i, j];
			NodoAStar nodo = new NodoAStar();
			nodo.setX(j);
			nodo.setY(i);
			if (ob != null) {
				nodo.setIsTransitable(false);
			}
			matriz[i, j] = nodo;
		}
	}

	Astar astar = new Astar(matriz, matriz[posRows, posColumns],
		matriz[posRowsPlayer, posColumnsPlayer], false);
	ArrayList listNodes = astar.calculatePath();

	return listNodes;
}
```

Figura 6.8. Creación de la matriz e instanciación de la clase Astar.

Por último en la figura 6.8 se muestra la forma en la que se crea la matriz de nodos que será utilizada por la clase "Astar" para calcular el camino más óptimo.

6.2. Métodos auxiliares utilizados en la inteligencia artificial de los enemigos.

```
private State selectStateFromNoSmart(State preState) {
	State newState = State.stop;

	switch (preState){
		case (State.up):
		if (posRows > 0 && ControllerLevel.arraySquare[posRows - 1,
			posColumns] == null) {

			newState = State.up;
			distanceMov = OFFSET_Y - widthSquare * posRows;
			posRows--;
		}
		else if (posColumns > 0 && ControllerLevel.arraySquare[posRows,
			posColumns - 1] == null) {

			newState = State.left;
			distanceMov = OFFSET_X + widthSquare * posColumns;
			posColumns--;
		}
		else if (posColumns < limitColumns - 1
			&& ControllerLevel.arraySquare[posRows,posColumns + 1] == null){

			newState = State.right;
			distanceMov = OFFSET_X + widthSquare * posColumns;
			posColumns++;
		}
		break;
		case (State.right):
		…
		…
```

Figura 6.9. Función selectStateFromNoSmart.

La función de la Figura 6.9 es utilizada para devolver el estado más próximo del estado que se le pasa por parámetro. Comprobamos primero el estado actual para que siga en la misma dirección. Esta función es utilizada cuando el enemigo no es inteligente.

```
private void moveFromState(){
       switch (state) {
              case State.up:
              if (System.Math.Abs(transform.position.y - distanceMov)
                     >= widthSquare){
                     state = State.stop;
              }
              else{
                     transform.Translate(0, speed * Time.deltaTime, 0);
              }
              break;
              case State.right:
              if (!isFlip) {
                     Flip();
                     isFlip = true;
              }
              if (System.Math.Abs(transform.position.x - distanceMov)
                     >= widthSquare){
                     state = State.stop;
              }
              else{
                     transform.Translate(speed * Time.deltaTime, 0, 0);
              }
             break;
             case State.left:
              …
              …
```

Figura 6.10. Función moveFromState.

La función “moveFromState” (Figura 6.10) es similar a la función del personaje para moverse en el estado actual. Al igual que con el personaje, volteamos horizontalmente la imagen del enemigo según la dirección.

```
public State getStateFromPosition(NodoAStar node)
{
        State state = State.stop;

        if (posRows == node.getY() && posColumns > node.getX()){

            state = State.left;
            distanceMov = OFFSET_X + widthSquare * posColumns;
            posColumns--;
        }
        if (posRows == node.getY() && posColumns < node.getX()){

            state = State.right;
            distanceMov = OFFSET_X + widthSquare * posColumns;
            posColumns++;
        }
        if (posRows < node.getY() && posColumns == node.getX()){

            state = State.down;
            distanceMov = OFFSET_Y - widthSquare * posRows;
            posRows++;
        }
        if (posRows > node.getY() && posColumns == node.getX()){

            state = State.up;
            distanceMov = OFFSET_Y - widthSquare * posRows;
            posRows--;
        }
        return state;
}
```

Figura 6.11. Función getStateFromPosition.

En la función de la Figura 6.11 le pasamos el nodo y en función del estado actual calculamos el estado al que debe moverse.

7. Conectividad con redes sociales.

En este apartado del anexo, primero integraremos la red social *Facebook* en nuestro videojuego, y luego introduciremos el framework "*Soomla*".

7.1. Añadiendo Facebook a nuestro juego.

Lo primero de todo es añadir el SDK de *Facebook* a nuestro proyecto. Aunque está en la *Unity Store* de forma gratuita, lo descargamos desde la web de *Facebook Developers*[1] para asegurarnos de añadir la última versión.

Una vez descargado, lo añadimos desde nuestro proyecto mediante "*Assets/Import Package/Custom Package*". En la barra de menús nos aparecerá un menú nuevo de Facebook. Antes de seguir con el código, debemos crear una aplicación de Facebook que nos proporcionará el ID necesario para utilizar el SDK.

Creamos una "App" en developers.facebook.com desde "*My Apps/Add a New App*".

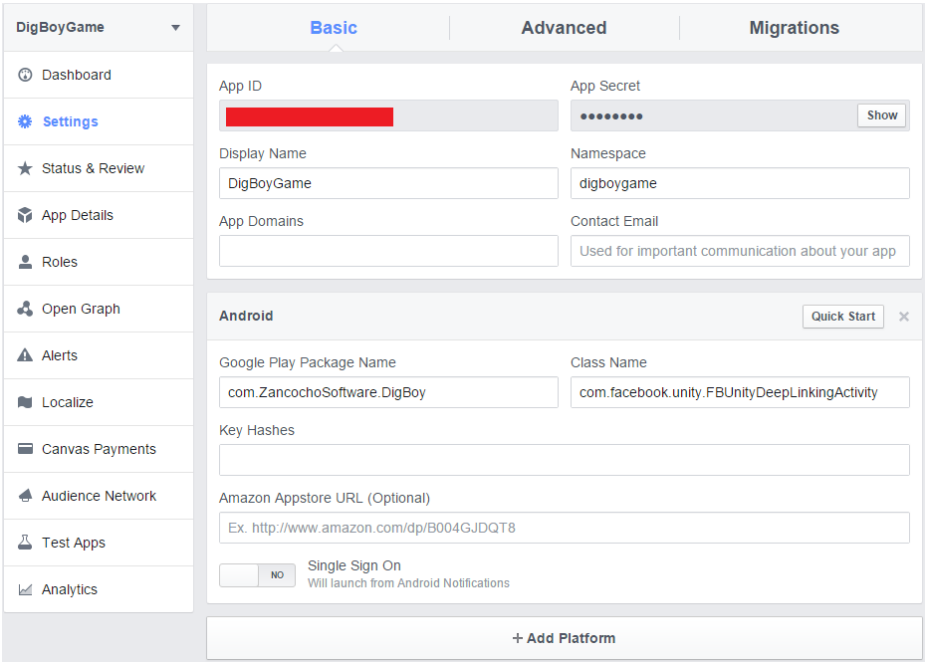


Figura 7.1. App de Facebook.

1. Página descarga del SDK de Facebook a fecha 2015/07/15: https://developers.facebook.com/docs/unity/downloads

Como se muestra en la figura 7.1, en “Settings” nos aparecerá el “App ID” que deberemos utilizar pero antes de ello debemos añadir una plataforma (*Android*) para que la aplicación de *Facebook* reconozca desde que plataforma es llamada. Los datos mínimos necesarios son el “Package Name” y el “Class Name” y para conseguirlos basta con ir al nuevo menú de *Facebook* que aparece en *Unity3d* y elegir la opción “Edit Settings”. Una vez rellenado los datos de la plataforma, copiamos el “App ID” y lo pegamos en el campo que aparece en la Figura 7.2.

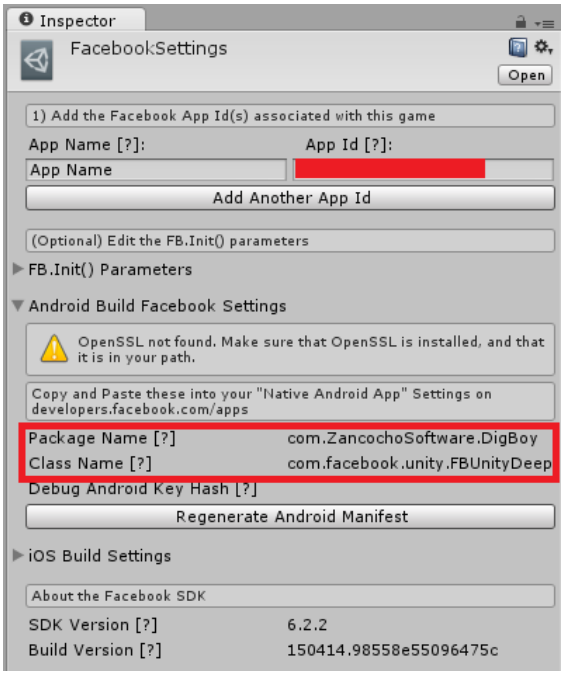


Figura 7.2. Settings de Facebook.

Si queremos comprobar que el SDK está correctamente configurado, podemos probar la escena “Interactive Console” dentro del directorio “Examples” que se añadió cuando añadimos el SDK. Para compartir una publicación, añadimos al botón de Facebook el script explicado a continuación.

```
private void SetInit(){
	enabled = true;
	if(FB.isLoggedIn){
		OnLoggedIn();
	}
	else{
		FB.Login("email,publish_actions", LoginCallback);
	}
}

private void OnHideUnity(bool isGameShown){
	if(¡isGameShown){
		Time.timeScale = 0;
	}
	else{
		Time.timeScale = 1;
	}
}
```

Figura 7.3. Primera parte del script para conectarnos con Facebook.

Como se muestra en la figura 7.3, el método "SetInit" lo utilizaremos para loguearnos en *Facebook* y el método "OnHideUnity" lo utiliza el SDK de Facebook para ocultar el juego. En este último método deberemos asegurarnos de que nuestro juego está pausado para que las acciones con Facebook no interfieran en el juego.

```
public void PublishInFB(int stars, string level){
	FB.Feed(
		linkCaption: "I have obtained "+stars+" stars in the level "+level,
		picture: http://i58.tinypic.com/25f1z71.png,
		linkName: "Prueba el juego Dig Boy!,
		link: "www.digboygame.com"
	);
}

void OnMouseDOwn(){
	FB.Init(SetInit, OnHideUnity);
	if(FB.IsLoggedIn){
		String levelActive = PlayerPrefs.GetString("levelActive");
		PublishInFB(calculateStarsForFB(levelActive), levelActive);
	}
}
```

Figura 7.4. Segunda parte del script para conectarnos con Facebook.

Cuando pulsemos sobre el botón, comprobamos que el usuario está logueado y llamamos al método “FB.Feed” añadiendo los campos que se muestran en la figura 7.4.

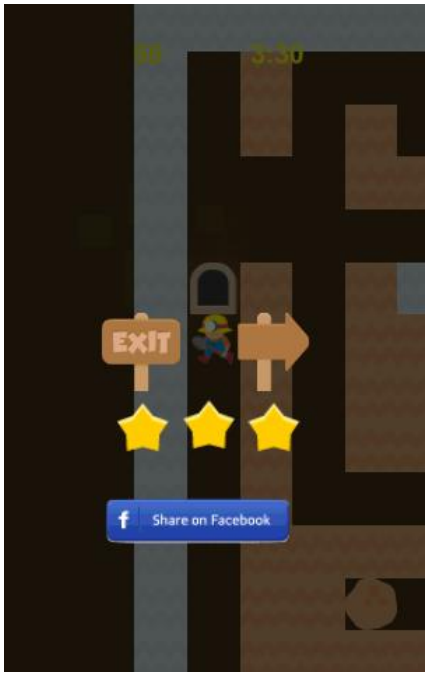


Figura 7.5. Botón de Facebook.

7.2. Introducción de *Soomla*.

Soomla es un *framework* open-source y gratuito que es un wrapper para integrar las distintas redes sociales y las compras "*in-Apps*" con *Android* e *iOS*.

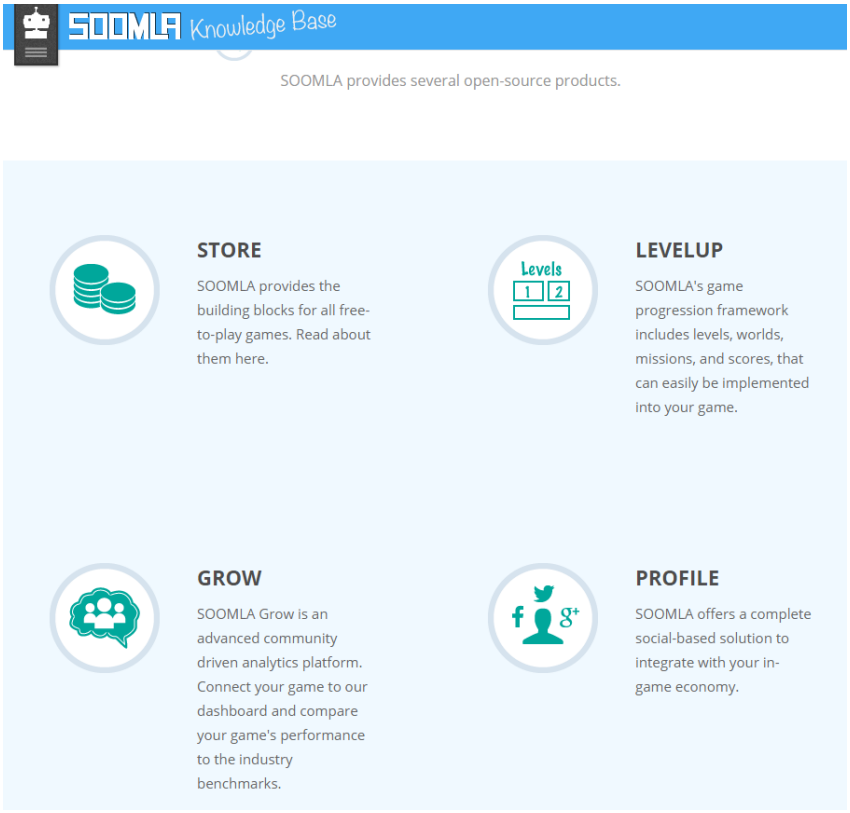


Figura 7.6. Enlace de Soomla[1] a la información para Unity3d

Al ser un wrapper, para utilizar *Soomla* es necesario tener añadido previamente los SDK de las redes sociales o plataformas que utilicemos. Tras ello y haber configurado el SDK correspondiente, añadimos los paquetes "soomla-unity3d-core.unitypackage" y "unity3d-profile.unitypackage" que están en su repositorio GitHub[2]. Tras ello, dentro del menú "Windows" se habrá creado un submenú "Soomla". En él podemos añadir las distintas redes sociales como se muestra en la figura 7.7. A partir de aquí, en los scripts simplemente instanciaremos *Soomla* y llamaremos a sus métodos.

1. Página de Soomla a fecha 2015/07/15: http://know.soom.la/unity/
2. Repositorio GitHub de Soomla a fecha 2015/07/15: https://github.com/soomla

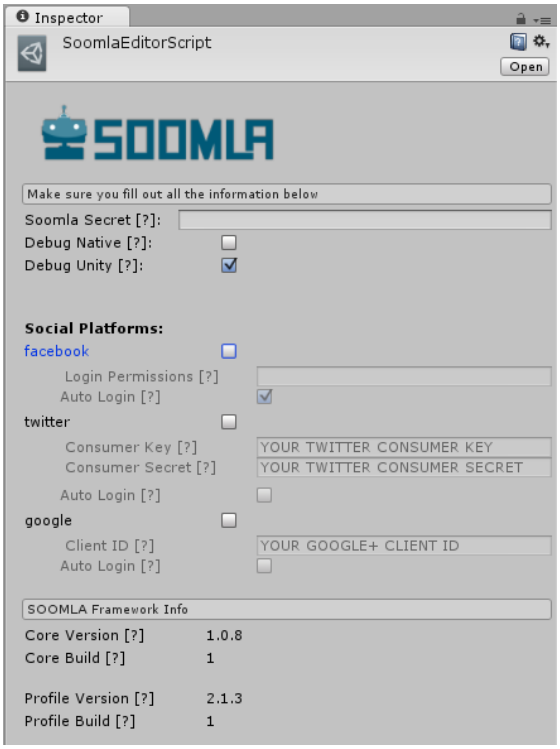


Figura 7.7. Settings de Soomla.

Dado que *Soomla* es un framework muy extenso y para muchas plataformas, no sólo *Unity3d*, no profundizaremos más en él. Aun así la forma de utilizarlo en otros casos es similar a la descrita (primero agregar el SDK correspondiente y luego Soomla) y en la web hay tutoriales detallados para cada plataforma o red social.

8. Añadiendo sonidos mediante el uso del *Audio Mixer*.

La música y sonidos es una parte importante de cualquier videojuego y no solo basta con añadirlo y reproducirlo. *Unity3d* incorpora el componente *Audio Mixer* que se encarga de controlar los diferentes canales de audio y nos permite configurar cada canal de forma separada. Para crear un *Audio Mixer*, hacemos clic derecho sobre un directorio de *Assets* donde nos interese guardarlo y elegimos "*Create/Audio Mixer*". Una vez creado, hacemos doble clic sobre él.

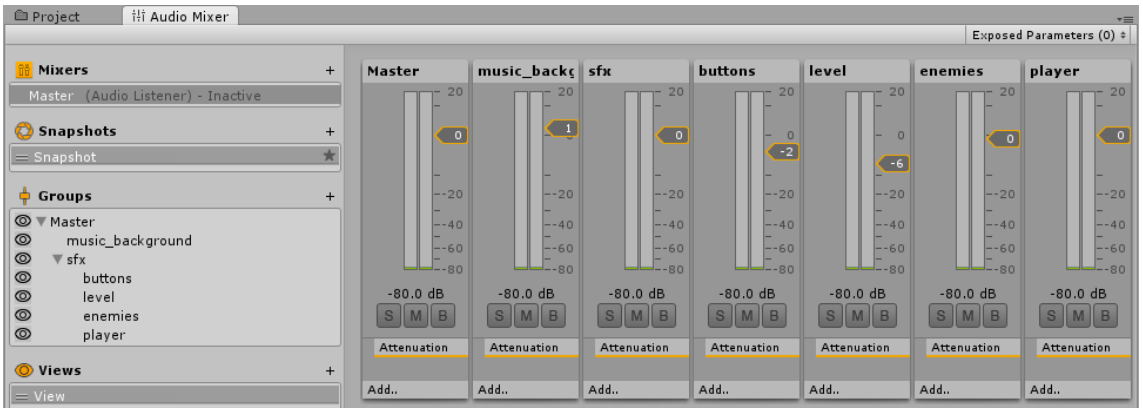


Figura 8.1. Audio Mixer con diferentes canales.

Cada vez que cambiamos de escena, todos los objetos de la nueva escena se cargan y el sonido no es ninguna excepción, por lo que se cortaría la música principal al pasar de escena. Para evitar ello, implementaremos un patrón singleton para la música principal como se muestra en la Figura 8.2.

```
Public class SoundManager : MonoBehaviour{
	public static SoundManager instance;

	void Awake(){
		if(instance == null){
			instance = this;
			DontDestroyOnLoad(gameObject);
		}
	}
}
```

Figura 8.2. Script que implementa un patrón singleton.

Creamos un *GameObject* vacío y le añadimos el script y un componente de tipo "Audio Source". A este último componente le añadimos el fichero de audio en "AudioClip" como salida elegimos el canal "music_background" del Mixer que hemos creado.

En el caso de los efectos de sonido no es necesario crear un singleton por lo que únicamente añadimos los "Audio Source" a cada objeto y los activamos desde los correspondientes scripts.

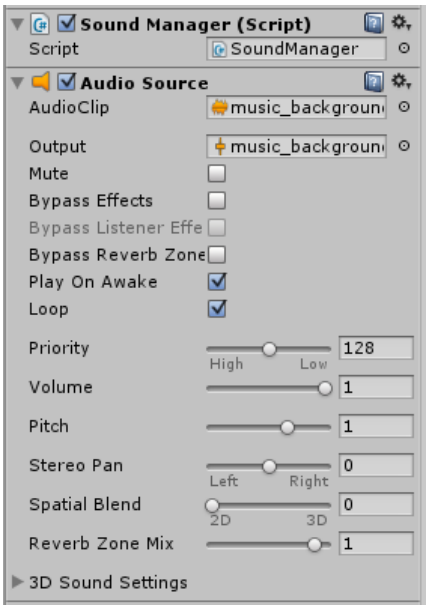


Figura 8.3. Componente Audio Souce y el script singleton.

9. Otras herramientas nativas de 2D enfocadas a las físicas.

Dado la temática de nuestro videojuego (o de cualquier otro videojuego) es muy difícil hacer uso en él de todas las funcionalidades de *Unity3d* enfocadas al desarrollo en 2D. Es por ello que dedicamos este último apartado del Anexo a comentar varias funcionalidades interesantes por medio de pequeños ejemplos.

9.1. *Area Effector 2D* y *Point Effector 2D*.

El *Area Effector 2D* aplica una fuerza a todos los *GameObjects* que están en un área.

Figura 9.1. Escena de Area Effector 2D.

Para explicar esta funcionalidad, hemos creado la escena de la Figura 9.1. En esta escena una pelota amarilla cae y entra dentro del área (“Box Collider 2D”) de la fecha que apunta a la derecha. Al entrar dentro de esta área, el área le aplicará una fuerza hacia la derecha y la siguiente fecha le aplicará otra fuerza hacia arriba.

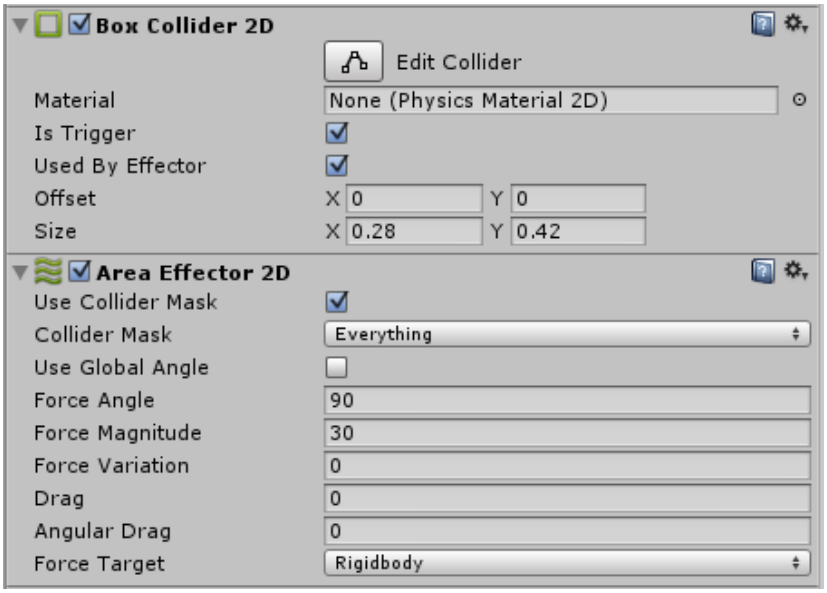


Figura 9.2. Componentes de la flecha.

En la Figura 9.2 se muestra los componentes de la flecha. Para que se active la fuerza, su "Box Collider 2D" debe estar marcada la opción "Is Trigger", es decir, como disparador. Además de ello, también tiene que tener activado la opción "Used By Effector" junto con un componente *Area Effector 2D*. En este último componente configuraremos la fuerza a aplicar en el área y dispone de los siguientes parámetros a configurar:

- Collider Mask: Selecciona las capas permitidas para interactuar con el "Effector".
- Force Direction: La dirección de la fuerza a ser aplicada.
- Force Magnitude: La magnitud de la fuerza a ser aplicada.
- Force Variation: La variación de la magnitud de la fuerza en ser aplicada.
- Drag: La fricción lineal que aplica a los "Rigidbody".
- Angular Drag: La fricción angular angular drag) que aplica a los "Rigidbody".
- Force Target: El objetivo donde el "Effector" aplica cualquier fuerza.

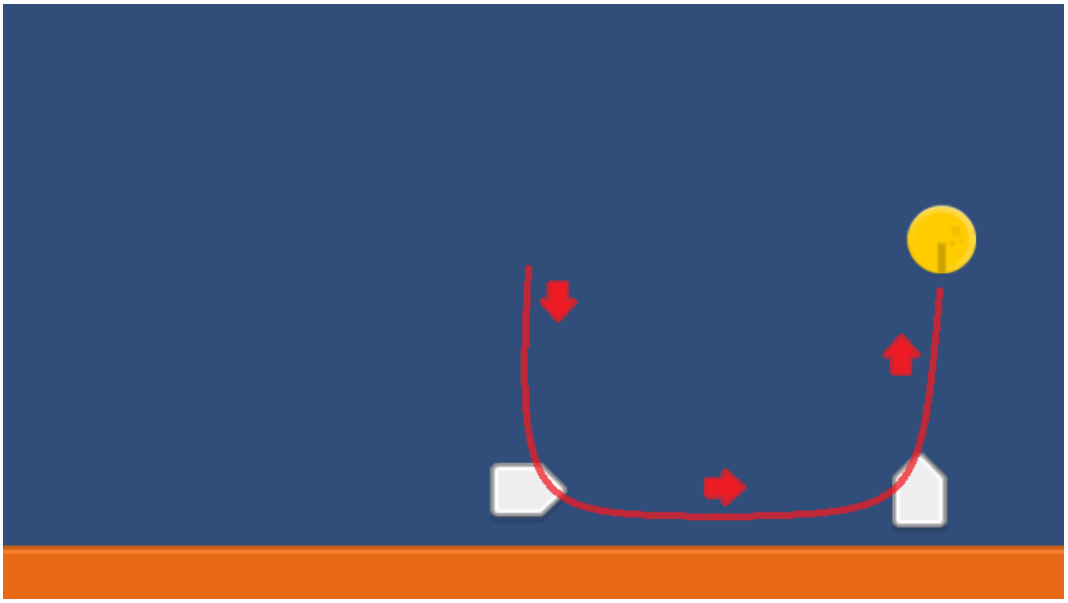


Figura 9.3. Fuerza que ejerce el área con Area Effect 2D sobre la pelota.

Point Effector aplica una fuerza para atraer o repulsar un *GameObject* contra un punto. En este caso disponemos de la escena de la Figura 9.4 en la que una pelota cae sobre una seta que hará que la pelota rebote, es decir, aplicará una fuerza de repulsión a la pelota.

Figura 9.4. Escena de Area Effector 2D.

Nuevamente el *GameObject* de la seta dispone de un "Box Collider 2D" con las opciones "Is Trigger", y "Used By Effector" activadas, con la diferencia que esta vez añadiremos un componente "Point Effector 2D" como se muestra en la Figura 9.5.

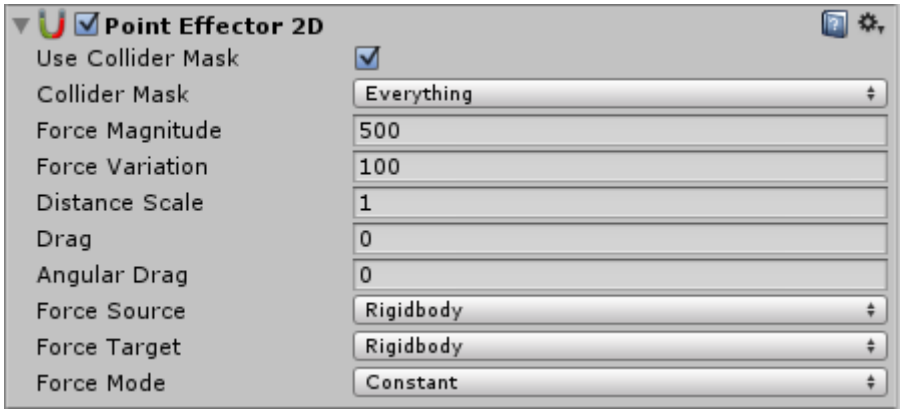


Figura 9.5. Componente Point Effector 2D.

Este componente dispone de los siguientes parámetros a configurar:

- Collider Mask: Selecciona las capas permitidas para interactuar con el "Effector".
- Force Magnitude: La magnitud de la fuerza a ser aplicada.
- Force Variation La variación de la magnitud de la fuerza en ser aplicada.
- Drag: La fricción lineal que aplica a los "Rigidbody".
- Angular Drag: La fricción angular angular drag) que aplica a los "Rigidbody".
- Force Target: El objetivo donde el "Effector" aplica cualquier fuerza.
- Force Source: Punto centro del "Effector".
- Force Mode: El modo utilizado para aplicar la fuerza (constante, lineal, etc).

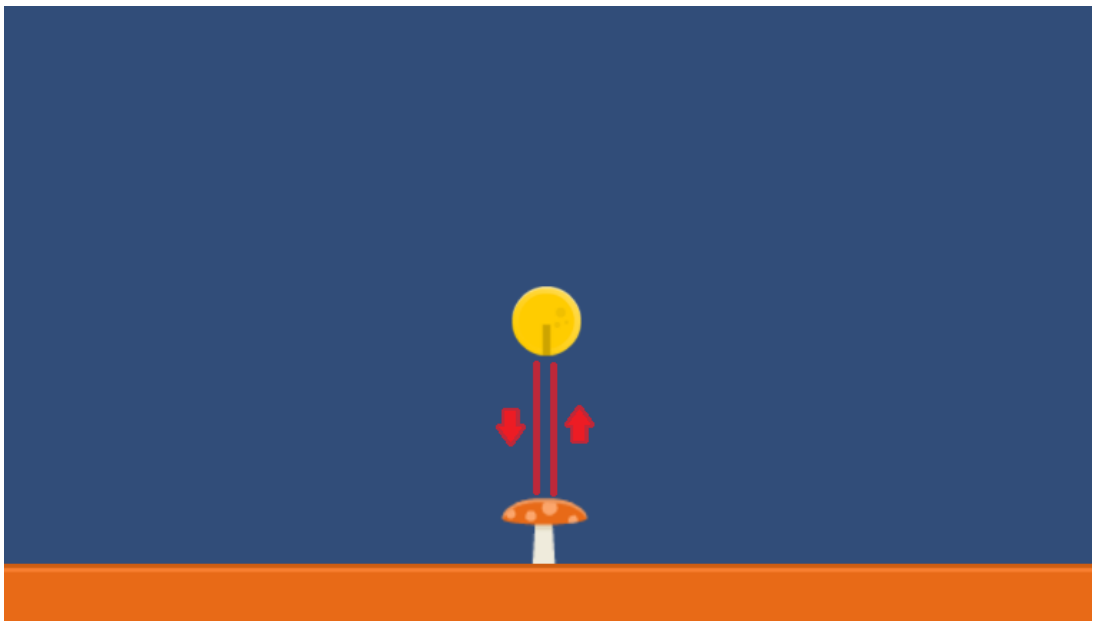


Figura 9.6. Fuerza que ejerce el área con Point Effector 2D sobre la pelota.

9.2.Creación de un material de físicas customizadas.

En los casos anteriores utilizando el "Effector" transmitimos una fuerza a los *GameObject* con los que colisionamos. En otros casos, nos puede interesar que un *GameObject* tenga un determinado comportamiento cuando colisione con otro *GameObject* y que no sea otros los que alteren su comportamiento. Para ello utilizamos un material de físicas customizado.

Figura 9.7. Escena de explicación de Physic 2D Material.

En la Figura 9.7 se muestra la escena que utilizaremos para crear un material de físicas customizadas mediante el componente "Physic 2D Material". La pelota amarilla dispone de las físicas por defecto por lo que al colisionar con el suelo se posará sobre el automáticamente, sin embargo la pelota naranja dispone de un material de físicas customizado que rebotará contra el suelo como si fuese una pelota real como se muestra en la Figura 9.8.

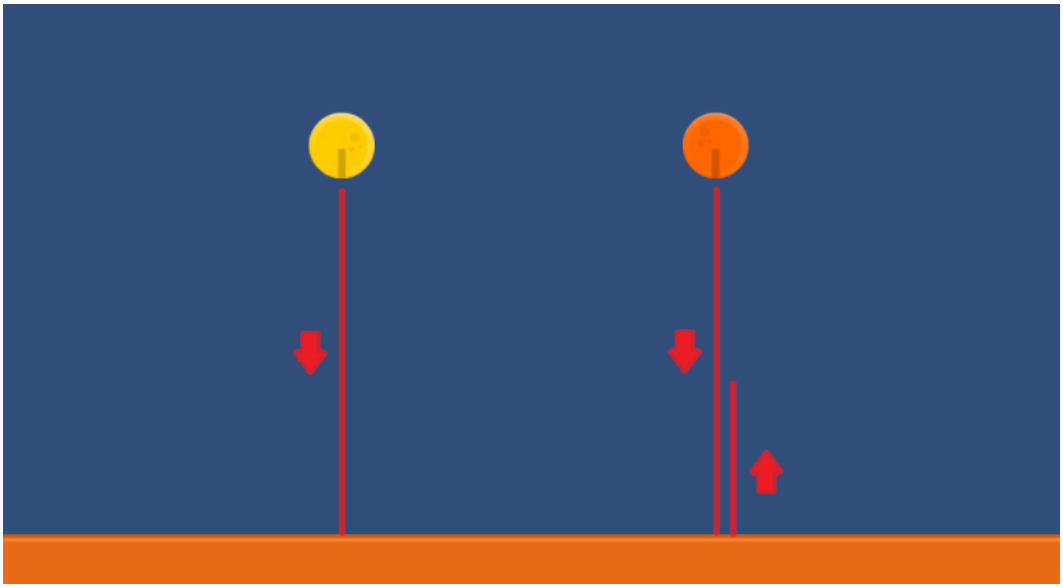


Figura 9.8. Diferencias entre físicas por defecto y material de físicas customizado.

Para ello, dentro de “Assets” creamos una carpeta “Material” y sobre ella hacemos clic derecho y elegimos “*Create/ Physic2D Material*” y en este caso lo nombramos como “Bouncy”.

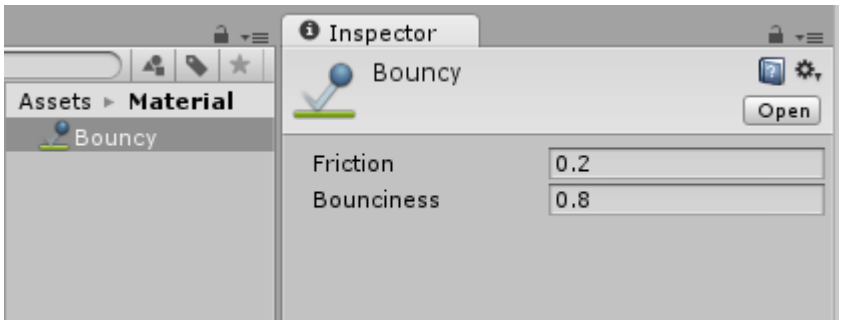


Figura 9.9. Parámetros del Physic 2D Material.

En la Figura 9.9 se muestra los parámetros del material de físicas, por una parte el coeficiente de fricción y por otra el grado de rebote sobre una superficie (un valor 1 sería un rebote perfecto sin pérdida de energía).

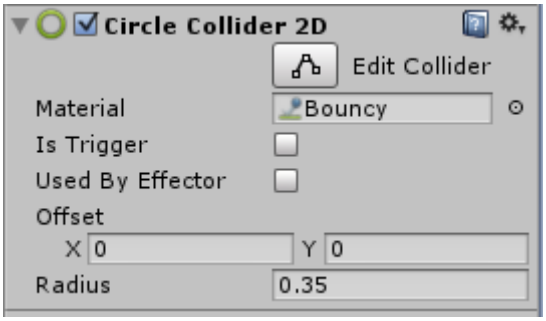


Figura 9.10. Circle Collider 2D con un Physic 2D Material.

Para añadirlo al *GameObject* de la pelota naranja, arrastramos el componente creado hacía el campo “Material” de su “Circle Collider 2D” (ó “Box Collider 2D”) y ya dispondrá de este comportamiento.