UVa

Introducción a Unity Engine

Prácticas de Diseño Interactivo

UVa

Universidad de Valladolid

Máster en Ingeniería de Diseño Industrial, Universidad de Valladolid

Diseño Interactivo

Luis Ignacio Jiménez Gil [nacho.jimenez@uva.es], v2.2 Junio 2023

Índice general

1 Introducción 7

1.1 Contextualización y Justificación 7

1.2 Contenidos 7

1.3 Motivación 7

2 Instalación de Unity 9

2.1 Objetivo de la práctica 9

2.2 Requisitos previos 9

2.3 Instalación de Unity paso a paso 9

2.4 Conclusiones 14

3 Entorno de desarrollo Unity 17

3.1 Objetivo de la práctica 17

3.2 Requisitos previos 17

3.3 Creación de un nuevo proyecto 17
3.3.1 Tipos de plantillas 17
3.4 Fundamentos de la interfaz de Unity 18
3.4.1 Controles Básicos 19
3.4.2 Otras herramientas básicas 21
3.5 Conclusiones 23

4 Niveles 25
4.1 Objetivo de la práctica 25
4.2 Requisitos previos 25
4.3 Construcción del modelo 25
4.4 Posicionamiento de la cámara 27
4.5 Objetos interactuando con otros objetos 28
4.6 Materiales 28
4.6.1 Asociar la textura al material y al objeto 29
4.7 Prefabs 30
4.8 Luces 31
4.9 Construir y publicar un proyecto 32
4.10 Conclusiones 34

5 Scripting 35
5.1 Objetivo de la práctica 35
5.2 Requisitos previos 35
5.3 Creación y composición de los scripts 35
5.4 Transformaciones básicas de objetos mediante scripts 37
5.5 Crear y Destruir Objetos desde un script 38

5.6 RayCast 39
5.7 Eventos 40
5.7.1 Eventos GUI y de Física 41
5.8 Añadir audio a nuestra escena 43
5.8.1 Adjuntar sonidos a los eventos de la escena 44
5.8.2 Audio Manager 45
5.9 Conclusiones 47

6 Interfaz de Usuario 49
6.1 Objetivo de la práctica 49
6.2 Requisitos previos 49
6.3 Elementos de la Interfaz 49
6.4 Asociar funciones a elementos de la interfaz 51
6.5 Personalización 52
6.6 Interacción con el Juego 54
6.7 Conclusiones 55

1. Introducción

1.1 Contextualización y Justificación

Durante esta parte de la asignatura se va a desarrollar un proyecto en el que el alumno seleccionará un producto previamente modelado en 3D y ofrecerá una interfaz para la presentación y personalización interactiva del mismo. Para ello el alumno aprenderá las nociones básicas del motor de videojuegos Unity, con el que se generará una aplicación a tal efecto.

1.2 Contenidos

Los contenidos de aprendizaje de Unity Engine se dividirán en una serie de prácticas de carácter incremental que componen los capítulos de este guion, donde los alumnos aprenderán paso a paso los conceptos necesarios para el desarrollo de su proyecto.

- El entorno de desarrollo Unity Engine: interfaz, previsualización, escenas y vistas, objetos, jerarquías, componentes y *assets*.
- Jerarquía de Niveles de objetos: Modelos 3D, iluminación, cámaras, audio, colisiones, *raycast*.
- *Scripting*: variables de componentes, bucles de eventos, input de usuario, transformaciones básicas, creación y destrucción de objetos, comunicación entre objetos.
- Interfaz de usuario: Canvas y escalado, *widgets*, *sprites*, *labels*, *layout*.

1.3 Motivación

A lo largo de las diez sesiones programadas de prácticas se pretende aprender a utilizar Unity Engine para el desarrollo de escenas interactivas. Las sesiones estarán

divididas en dos bloques: el primero de ellos estará dedicado a conocer y aprender la bases de Unity mediante prácticas guionizadas proporcionadas por el profesor. El segundo bloque consistirá en el desarrollo del proyecto planteado por el alumno en sesiones tutorizadas. Se reservará la última sesión para realizar una pequeña demostración de los proyectos de cada alumno.

2. Instalación de Unity

2.1 Objetivo de la práctica

El objetivo principal de esta práctica es conseguir instalar Unity (versión ***2021.3.25f1***) en nuestro equipo de trabajo. Unity está disponible como plataforma de desarrollo para Microsoft Windows, Mac OS y Linux, aunque por cuestiones de sencillez, el guión de esta práctica detallará la instalación en Microsoft Windows 11.

2.2 Requisitos previos

Para el correcto seguimiento de la práctica, el alumno necesitará un equipo con las características mínimas exigidas por el software Unity:

- OS: Windows 7 SP1+, 8, 10, 64-bit versions only; Mac OS X 10.13+; Ubuntu 16.04, 18.04, and CentOS 7.
- GPU: Graphics card with DX10 (shader model 4.0) capabilities.

2.3 Instalación de Unity paso a paso

Lo primero que debemos realizar es la descarga del instalador de Unity Hub desde la página de Unity[1] (Figura 2.1). AVISO: Algunas de las figuras que se muestran en el guión pueden cambiar respecto a lo que se puede ver en la web de Unity.

Una vez dentro, en la sección *Get Started* podemos acceder a un botón donde pone *Download Unity*. Dentro de los distintos planes que proporciona Unity podremos elegir entre varios. A efectos de este tutorial, seleccionaremos una licencia **Individual y**

[1] `https://unity.com/`

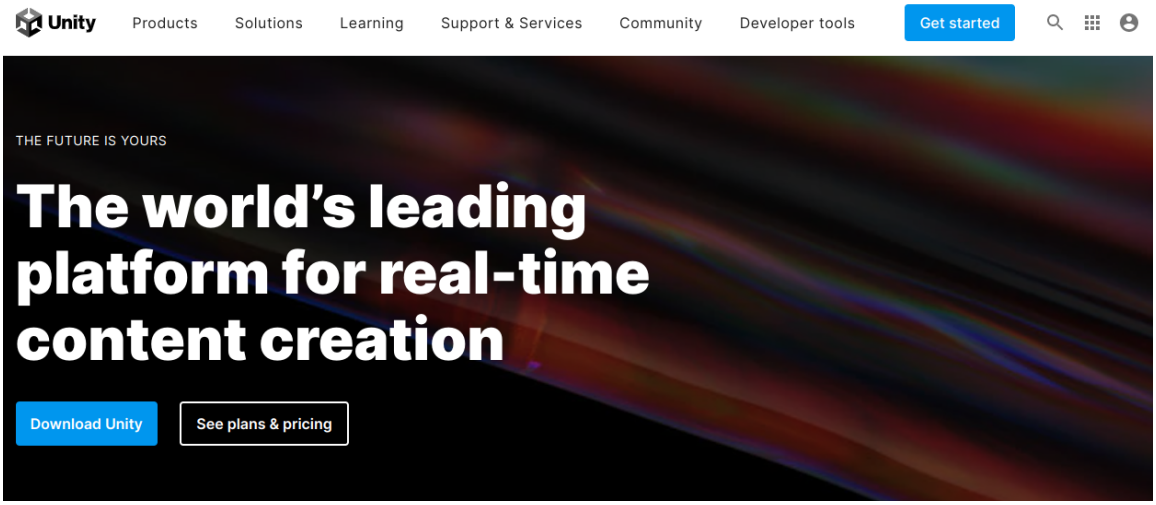


Figura 2.1: Imagen de la web de Unity donde se puede descargar el software Unity.

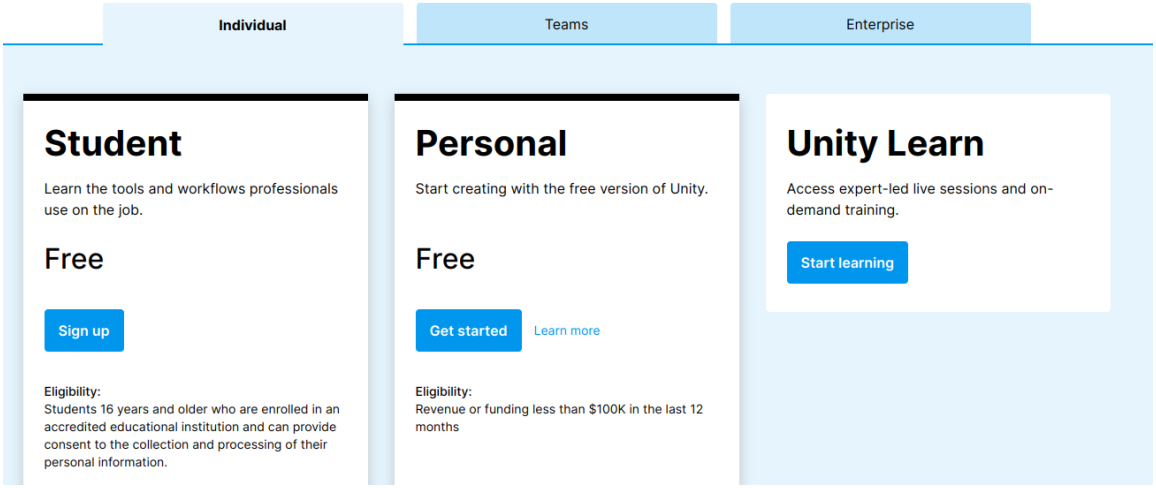


Figura 2.2: Opciones de licencias para la descarga del software Unity.

Personal(Figura 2.2).

Por defecto, la opción de descarga es para la versión de Windows (Figura 2.3, de ahora en adelante la versión que se utilizará a lo largo del tutorial). Aquellos alumnos que deseen utilizar versiones de MacOS o Linux pueden hacerlo, aunque es posible que requieran de utilizar versiones anteriores del motor por razones de incompatibilidad (por lo menos en Linux).

Al hacer click en *Download for Windows* nos descargaremos el instalador *UnityHubSetup.exe*. Cuando termine la descarga del archivo lo ejecutaremos y nos aparecerá un *Wizard Setup* donde deberemos aceptar los Términos de Servicio (Figura 2.4).

En ese momento se nos pedirá introducir la ruta de instalación (Figura 2.5) que por defecto se encuentra en Archivos de Programa ->Unity Hub. Podremos utilizar la que

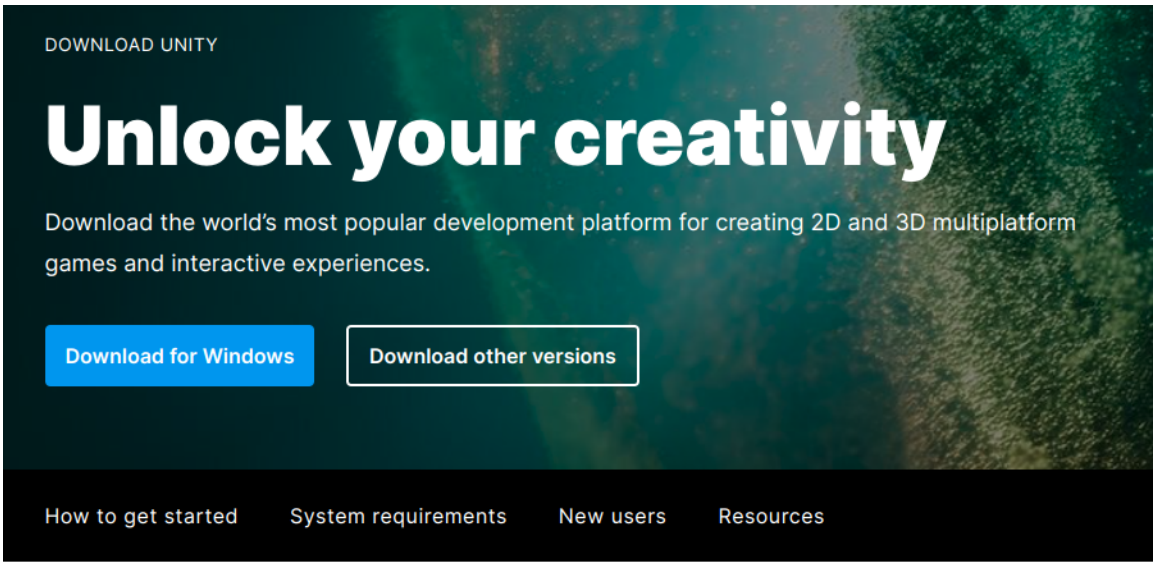


Figura 2.3: Pantalla final para la descarga del software Unity.

Figura 2.4: Acuerdo de Términos y Condiciones de servicio del software Unity.

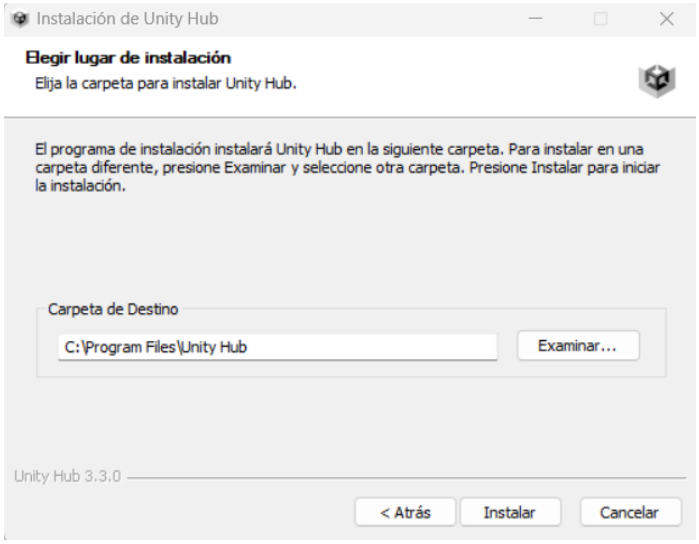


Figura 2.5: Ventana del wizard de instalación donde se puede especificar la carpeta de instalación de Unity.

queramos pero se recomienda el uso de una partición de disco con bastante espacio disponible, ya que esta aplicación tendrá posteriormente que descargar e instalar la/s versiones del Editor de Unity.

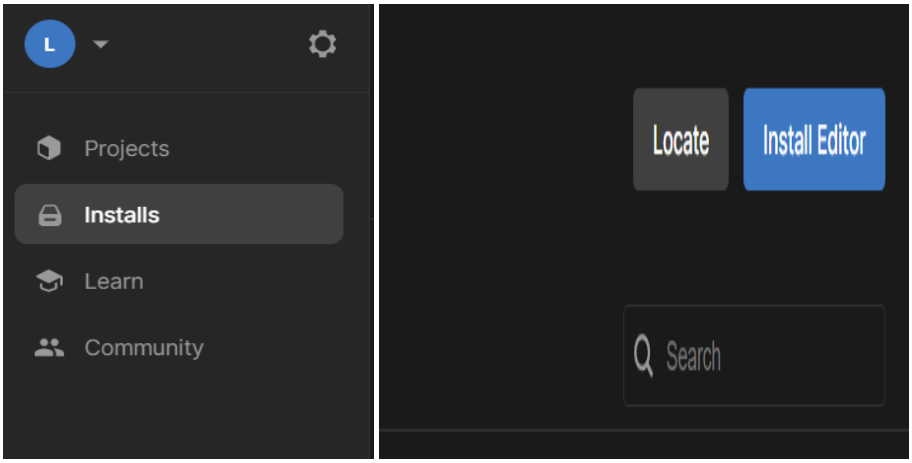
Al cabo de un tiempo, que no debería ser más de unos minutos, el instalador finalizará dando la posibilidad de ejecutar Unity Hub al salir. Marcando esta opción terminamos la instalación.

Cuando Unity Hub se inice, veremos una interfaz que contiene un menú lateral (Figura 2.6(a)) a la izquierda con cuatro opciones: *Projects*, *Installs*, *Learn* y *Community*. Inicialmente seleccionaremos la opción *Installs* y más adelante explicaremos la función de las demás.

Dentro de la vista *Installs*, en la esquina superior derecha (Figura 2.6(b)) de la misma podremos ver un par de botones. La opción *Locate* nor permite cambiar el directorio de instalación del Editor de Unity en cualquier momento. Una vez elegido el directorio hacemos click en *Install Editor*.

En la ventana emergente podremos seleccionar la versión del Editor de Unity que deseamos instalar. Existen distintas versiones oficiales (Figura 2.7). Nos interesa obtener la última versión con soporte de larga duración (*long term Support -LTS*) aunque también estarán disponibles otras versiones, así como, las versiones en desarrollo actuales. **A efectos de este tutorial, trabajaremos con la versión del Editor de Unity *2021.3.25f1*.**

Al seleccionar la opción de instalar esta versión aparecerá otra ventana emergente, la cual nos informará del tamaño de la instalación en función de los módulos que añadamos. Según la plataforma en la queramos que se ejecute nuestra aplicación deberemos instalar unos paquetes u otros. Si se dispone de espacio suficiente se recomienda instalar todos los módulos de plataforma. Los módulos mínimos que necesitaremos serán los indicados para crear aplicaciones de escritorio en Windows; el módulo *WebGL*



(a) (b)

Figura 2.6: En la Figura (a) se muestra el menú lateral de Unity Hub. La Figura (b) muestra la sección superior derecha de la interfaz de Unity Hub.

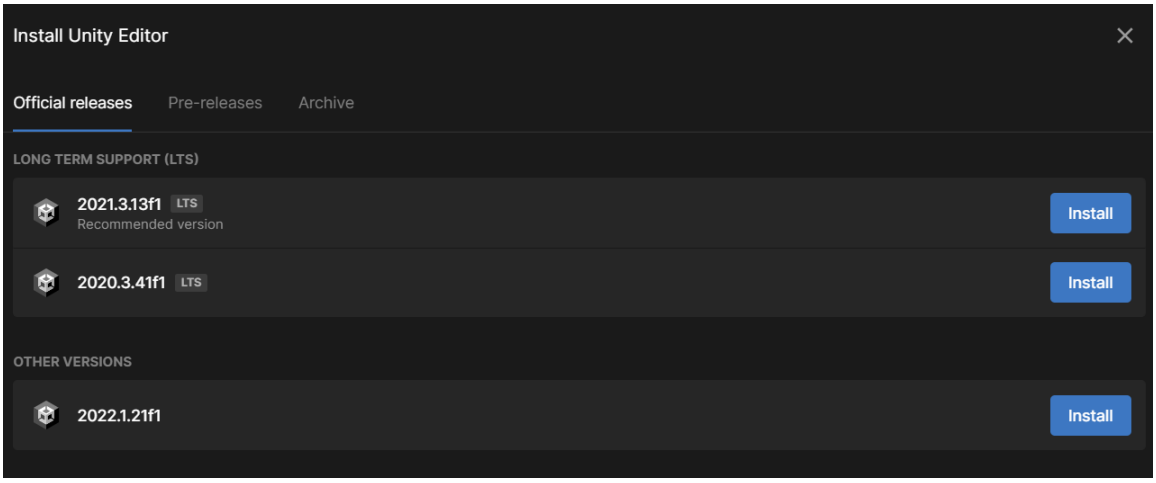


Figura 2.7: Elección del Editor de Unity que se va a instalar.

Build Support que nos permitirá crear aplicaciones que se ejecuten en navegadores; y el módulo *Documentation* para acceder a la documentación de ayuda.

La instalación del editor puede durar un tiempo considerable e incluso dar fallos. En el caso de error en la instalación se recomienda consultar con el profesor o acudir a algún foro en Internet.

A continuación veremos qué podemos encontrar en las otras pestañas del menú de Unity Hub. La opción *Learn* (Figura 2.8(a)) nos permite acceder a los distintos cursos y tutoriales que Unity ofrece a través de su plataforma. Estos cursos también son accesibles desde su página web[2] y son muy recomendables para aquellos alumnos que no se hayan acercado nunca a Unity o entornos similares como puede ser *Blender*[3], ya que puede resultar abrumador al principio. Durante las prácticas se enseñarán los conceptos básicos para manejarse con Unity pero de cara a un conocimiento más experto la recomendación es explorar los cursos.

En la opción *Community* (Figura 2.8(b)) tendremos acceso a las distintas plataformas que ofrece Unity de intercambio de conocimiento: foros, blogs, tienda, etc. A pesar de no incluirse en esta pestaña, existe el canal de *Youtube* de Unity[4] donde hay colgados vídeos con toda clase de contenido relacionado con Unity desde nivel principiante al más experto.

Por último tenemos la pestaña *Projects* (Figura 2.8(c)), donde podremos encontrar todos los proyectos actuales que tenemos en nuestro equipo. Además en la esquina superior derecha encontramos dos botones: uno para crea un nuevo proyecto y el segundo que nos permite abrir proyectos existentes tanto en local como en remoto. De momento, haremos click en el botón *New Project*.

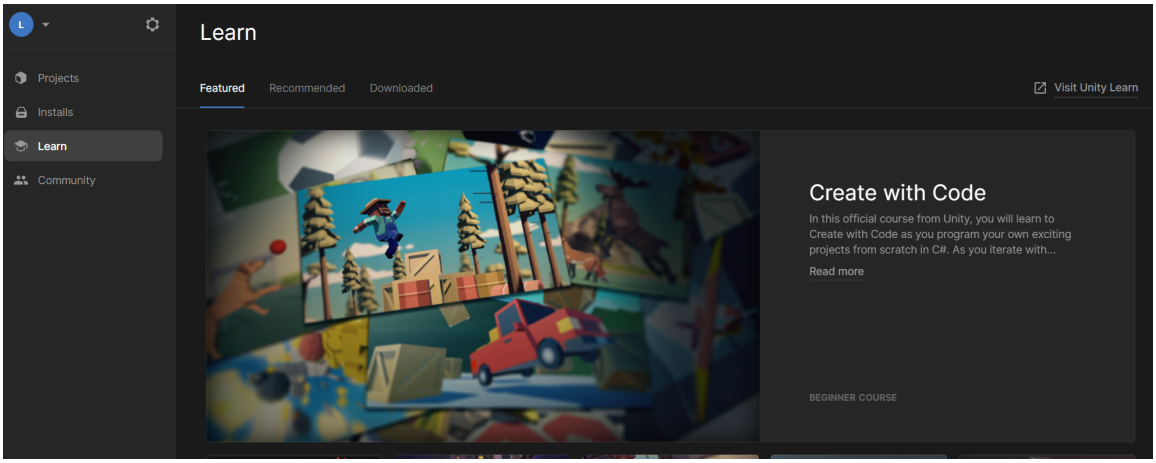
En la ventana emergente que aparecerá podremos navegar entre todas las plantillas de proyectos disponibles en Unity Hub. Para utilizar una plantilla, primero deberán descargarse. Cuando dispongamos de ella, nos aparecerá la opción de nombrar nuestro proyecto, así como, elegir la ubicación del mismo.

Los distintos tipos de plantilla van desde los proyectos vacíos de distinto tipo y plataforma, hasta ejemplos funcionales. También están disponibles unos microjuegos que forman parte de los cursos de Unity y que ofrecen tutoriales guiados paso a paso.

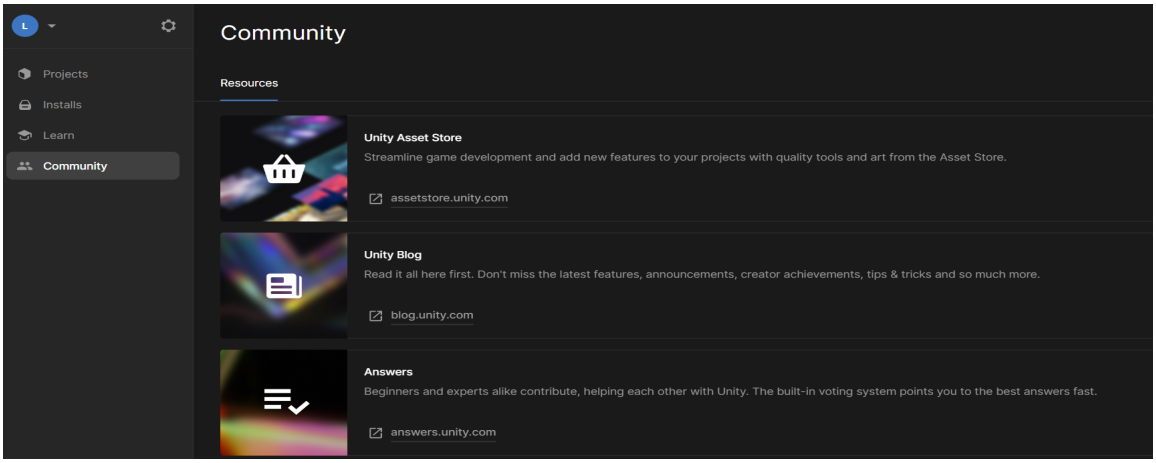
2.4 Conclusiones

En este punto, deberíamos tener Unity instalado y funcional en nuestro equipo. En la próxima práctica crearemos un proyecto y veremos los fundamentos para navegar por la interfaz de Unity y realizar operaciones básicas.

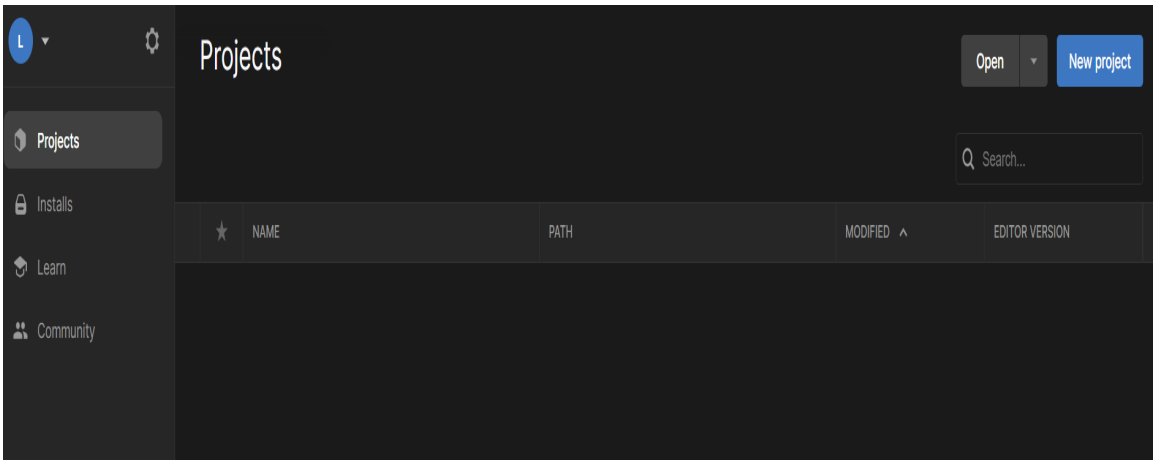
[2] `https://learn.unity.com/`
[3] `https://www.blender.org/`
[4] `https://www.youtube.com/c/unity`



(a)



(b)



(c)

Figura 2.8: (a) Unity Learn. (b) Unity Learn. (c) Unity Projects.

3. Entorno de desarrollo Unity

3.1 Objetivo de la práctica

Los objetivos principales de esta práctica son aprender los fundamentos necesarios para poder crear un proyecto desde cero y conocer las bases para navegar a través de la interfaz de Unity. En particular dividiremos esta práctica en dos bloques, uno por cada objetivo comentado anteriormente.

3.2 Requisitos previos

Haber cumplido los objetivos de la Práctica 0 y **tener instalado el Editor de Unity**.

3.3 Creación de un nuevo proyecto

Para crear un nuevo proyecto lo primero que debemos saber es qué tipo es. En la ventana emergente que aparece al hacer click en *New Project* podemos ver un repertorio de plantillas que sirven a propósitos diferentes.

3.3.1 Tipos de plantillas

- **2D** Genera un proyecto vacío configurado para una aplicación en dos dimensiones.
- **3D** Genera un proyecto vacío configurado para una aplicación en tres dimensiones.
- **Canalización de renderizado universal (URP)** La plantilla de canalización de renderizado universal configura los ajustes del proyecto para proyectos en los que el rendimiento, la compatibilidad con una amplia variedad de plataformas y la facilidad

para personalizar gráficos son las consideraciones principales. URP también incluye un renderizador 2D optimizado completo con luces 2D y píxeles, renderizado perfecto y una solución integrada de posprocesamiento.

- **Canalización de renderizado de alta definición (HDRP)** La plantilla incluye la configuración y los activos que necesita para comenzar a crear con HDRP. Utilizando técnicas de iluminación basadas en la física, ofreciendo materiales avanzados, iluminación y opciones de posprocesamiento , compatibles con *Ray Tracing*[1] en tiempo real y *Pathtracing*[2] fuera de línea. Aprovecha una arquitectura híbrida configurable *tile/cluster* diferida/directa. HDRP ofrece las herramientas para crear proyectos con un alto estándar gráfico para plataformas de gama alta como son Windows, Linux, Mac, PS4. PS5, XBox One, XBox Series y desktop VR.
- **Mobile** Estas plantillas incluyes los paquetes y la configuración para generar proyectos en entornos móviles como son iOS y Android.
- **VR** Esta plantilla está preparada para un comienzo rápido con aplicaciones de realidad virtual.
- **AR** Esta plantilla está preparada para un comienzo rápido con aplicaciones de realidad aumentada.
- **First Person** Esta plantilla preconfigura el proyecto para controlar un personaje desde una perspectiva de primera persona.
- **Third Person** Esta plantilla preconfigura el proyecto para controlar un personaje desde una perspectiva de tercera persona.

Según la aplicación que deseemos crear deberemos elegir una opción u otra. Para la realización de estas prácticas se recomienda comenzar con una plantilla del tipo **3D Sample Scene (HDRP)**, disponible en Unity Hub, con el objetivo de familiarizarnos con el entorno de Unity.

Al seleccionar una plantilla primero deberemos descargarla. Posteriormente introduciremos el nombre del proyecto y su localización. Al generar el proyecto Unity debe construir la plantilla, lo cual puede llevar unos minutos. Al terminar, se abrirá el Editor de Unity que tengamos instalado con la configuración por defecto de la plantilla.

3.4 Fundamentos de la interfaz de Unity

Con nuestro proyecto abierto deberemos ver una disposición del editor similar a la mostrada en la Figura 3.1.

El editor está dividido en cinco ventanas diferentes:

- **A) Tutorials** Los proyectos de ejemplo como este vienen con pequeños tutoriales para aprender las bases de cada tipo de proyecto. Es recomendable realizarlos co-

[1]Ray Tracing es una tecnología que se encarga de mejorar las iluminaciones, sombras y reflejos de los videojuegos. Las palabras Ray Tracing significan en español trazado de rayos, haciendo referencia a un mucho mejor cálculo de los reflejos y sombras creados por las iluminaciones del entorno. Con este algoritmo, las escenas en los videojuegos se acercan más a la realidad pero también tienen sus desventajas, y es que el rendimiento en FPS globales se ve bastante penalizado.

[2]https://la.blogs.nvidia.com/2022/05/10/que-es-el-path-tracing/

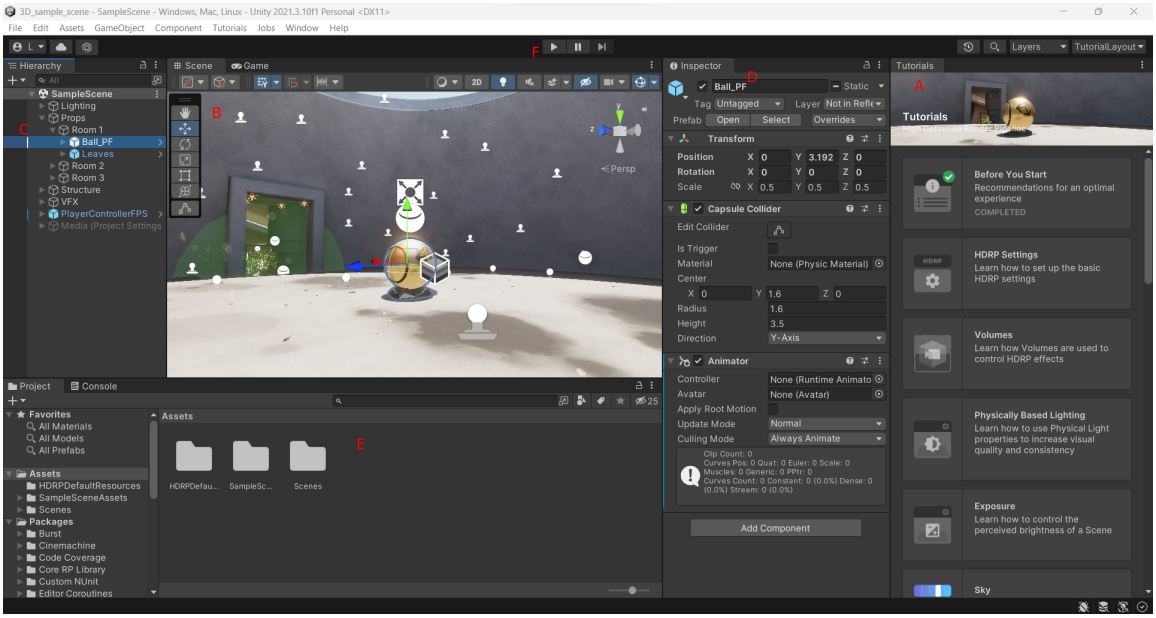


Figura 3.1: Interfaz del Editor de Unity con el proyecto de ejemplo abierto. A) *Tutorials*. B) *Scene*. C) *Hierarchy*. D) *Inspector*. E) *Project*. F) Controles de Reproducción.

mo primer paso, ya que en última instancia serán recordatorios de funcionalidades básicas que utilizaremos en nuestro proyecto.

- **B) Scene** Esta ventana nos mostrará la Escena. La Escena es el conjunto de objetos y elementos que vamos a mostrar y poder utilizar dentro de nuestro proyecto. Según el proyecto sea 2D o 3D los elementos se mostrarán en su correspondiente perspectiva.
- **C) Hierarchy** Esta ventana nos mostrará la jerarquía de objetos de la escena. En ella podremos navegar hasta seleccionar el elemento que deseemos tratar a continuación.
- **D) Inspector** Aquí podremos ver las características y variables pertenecientes a cada elemento del proyecto. No es una ventana constante sino que cambiará las propiedades según el tipo de elemento que hayamos seleccionado.
- **E) Project** El navegador de ficheros del proyecto. Se diferencia de la ventana *Hierarchy* en que esta permite navegar y seleccionar entre todas las carpetas del proyecto.
- **F) Reproductor** En la parte superior del Editor encontramos unos botones de *Play*, *Pause* y *Step* que sirven para construir la escena y controlar su ejecución.

3.4.1 Controles Básicos

Hasta el momento lo único que sabemos es la organización del Editor pero antes de entrar en materia necesitamos aprender a movernos por esta organización. A continuación detallaremos algunas de las operaciones básicas, controles y accesos rápidos a funciones del Editor.

Figura 3.2: Recorte de la ventana *Scene* del editor de Unity.

En primer lugar aprenderemos a movernos en la ventana *Scene* (Figura 3.2). El botón con un símbolo de una mano sirve para arrastrar, mover objetos y moverse por la escena. Con el botón izquierdo del ratón *Left-Click* presionado podremos desplazar la vista de la escena dentro del plano actual. Usando *Alt + Left-Click* el puntero cambiará a un ojo, y podremos orbitar sobre el foco de la escena (el foco será el punto central imaginario de la escena). Mediante *Right-Click* podemos orbitar también pero aquí el foco estará situado desde el punto de vista de la cámara. Por último, utilizando *Wheel* o *Alt + Right-Click* se puede acercar o alejar la cámara (*zoom in/out*) de la escena.

Si queremos centrar la escena en un objeto concreto, seleccionaremos dicho objeto en la ventana *Hierarchy* y presionaremos *F* en el teclado. Como ejemplo, desplazaremos la vista de la escena con el ratón hasta que la bola del centro de la sala no se vea. Buscaremos el objeto *Ball_PF* y haremos centraremos la vista de nuevo en ese objeto.

Dentro de la ventana *Hierarchy* podemos ver como el objeto *Ball_PF* contiene otros objetos. Esto se debe a que *Ball_PF* es un contenedor o agrupación de objetos. Si hacemos click en el triángulo a la izquierda del objeto podemos desplegar sus elementos. En este caso hay dos: la bola y la peana. Si hacemos click en el símbolo > la escena cambia y solo estarán disponibles los objetos del contedor. Es interesante agrupar objetos cuando el modelo es complejo y cada una de sus partes ha de editarse de forma separada.

Estamos hablando de la escena en todo momento pero **¿cómo sabemos si la composición de la escena es correcta o similar a lo que queríamos en un principio?**

Para ejecutar la escena que actualmente se tiene seleccionada hacemos click en el símbolo *Play* de la parte superior del editor. El programa tardará unos segundos en compilar el proyecto. Sobre la venta *Scene* aparecerá otra pestaña llamada *Game* donde podremos ver la escena en ejecución. Podremos recorrer la escena utilizando el puntero y las teclas *WASD* de nuestro teclado, de la misma manera que en cualquier juego de tipo *First Person Shooter (FPS)*. Haciendo doble click en la pestaña *Game* podremos

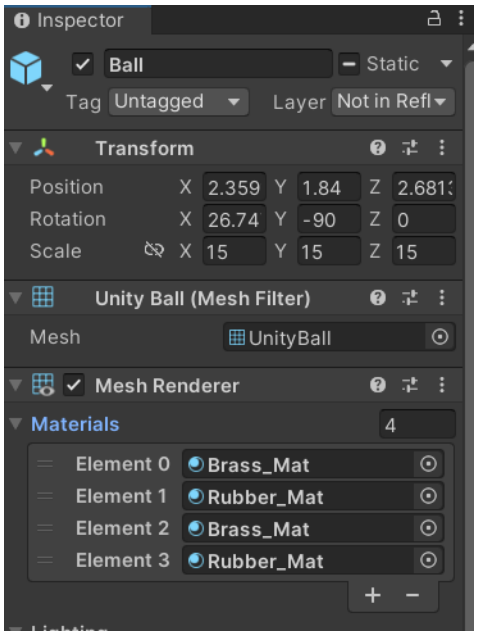


Figura 3.3: Ventana *Inspector* del objeto seleccionado.

ampliar la ventana a tamaño completo.

Una vez explorada la escena y habiéndonos familiarizado con los controles vamos a realizar un cambio simple sobre la escena. La idea es cambiar el aspecto de la bola de tal manera que pase de ser metálica a hecha de madera. Para ello necesitamos seleccionar en *Hierarchy* el objeto *Ball* dentro del contenedor *Ball_PF*. En la ventana *Inspector* (Figura 3.3) ver las distintas características de este objeto. Entre ellas veremos la pestaña *Materials*. En este caso las texturas del objeto *Ball* está compuesto por cuatro elementos cada uno de ellos con una textura asociada: latón (*brass*) o goma (rubber). Si variamos el orden de los materiales podremos observar a que pieza de la bola corresponde cada elemento.

Haciendo click en el selector a la derecha de cada elemento podremos cambiar el material seleccionado para cada uno de ellos. La idea es asignar diferentes texturas a la bola y comprobar como cambia la escena acorde al material. Como hemos dicho antes, probaremos con el material madera (Figura 3.4).

3.4.2 Otras herramientas básicas

El problema de los ejemplos es que ya vienen hechos. Normalmente se aprende más construyendo desde cero la escena (aunque de cara a la entrega final de esta asignatura es bueno saber reutilizar elementos).

Vamos a crear un nuevo proyecto del tipo 3D HDRP vacío. Una vez construido veremos que la escena se compone de una rejilla vacía y en la jerarquía solo existe la cámara principal, un simulador de cielo y niebla y una luz simulando el Sol (de momento vamos a ignorar estos elementos).

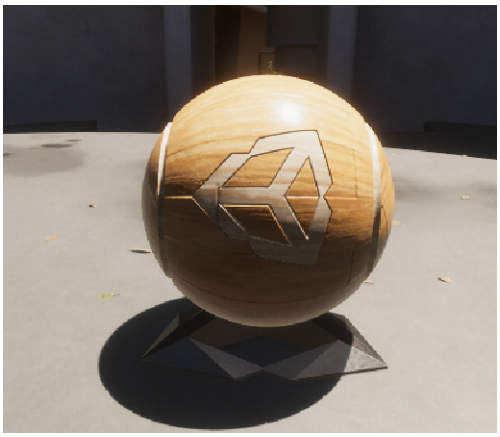


Figura 3.4: La objeto con el material madera aplicado.

¿Cómo añadimos sólidos a nuestra escena?

Todos los sólidos de una escena parten en algún momento de una primitiva básica. Estas primitivas básicas pueden encontrarse en el menú *GameObject ->3D Object* o bien haciendo *Right-Click* en la ventana *Hierarchy* en el submenú *3D Model*.

De cara a este ejemplo seleccionaremos bien una esfera o bien un cubo, el cual aparecerá en el centro de la escena. En la ventana *Scene*, debajo del símbolo mano, aparecen otros botones que se corresponden con las transformaciones básicas: traslación, rotación y escalado. Adicionalmente aparecen dos botones más que se corresponden con combinaciones de estas trasformaciones. Mediante estas trasformaciones podemos convertir nuestras primitivas en objetos más diferenciados.

¿Cómo cambio el color de un objeto?

Al igual que antes, se trata de cambiar las texturas asociadas a un objeto, pero en este caso el proyecto no incluye nada más que unos poco materiales básicos. La idea es crear nuestro propios materiales. Para ello, en el menú *Assets* de la barra superior, mientras tenemos seleccionado un objeto, buscamos la opción *Create->Material*. El nuevo material se añadirá a nuestro proyecto en la ruta actual en la que nos encontremos. Es de buenas prácticas tener el directorio del proyecto organizado debidamente.

Al seleccionar este material podremos cambiar sus características a nuestro gusto y asignarlo a nuestro objeto arrastrándolo a la pestaña de materiales.

Ejercicio

Para comprobar los conocimientos aprendidos durante esta sesión se pide que el alumno añada diferentes primitivas básicas a la escena y juegue con las características de los objetos, realizando transformaciones básicas y cambiando las texturas de los objetos.

3.5 Conclusiones

En esta práctica hemos aprendido el funcionamiento básico del editor y algunas de las posibilidades que ofrece. Es recomendable realizar los tutoriales de las plantillas de ejemplo para continuar familiarizándose con el entorno.

4. Niveles

4.1 Objetivo de la práctica

Mediante la realización de esta práctica aprenderemos los conceptos necesarios para la creación de entornos complejos formados por múltiples componentes que interactúan entre sí. Una vez acabada tendremos los conocimientos necesarios para configurar la física de la escena y los elementos audiovisuales que nos permitan simularla con cierto realismo.

4.2 Requisitos previos

Para la consecución de la práctica se desarrollará un modelo inicial relativamente complejo. Este modelo inicial deberá ser creado por el alumno utilizando los conocimientos adquiridos en la práctica anterior sobre **disposición y manejo de la interfaz del Editor de Unity**.

4.3 Construcción del modelo

Lo primero que debemos hacer es generar un proyecto 3D vacío, el cual solo constará de una luz direccional y una cámara principal. Dentro de él crearemos un objeto vacío en el menú *Hierarchy* de nombre **Escalera**. Añadimos a continuación un objeto *Cubo* al que pondremos el nombre **Peldaño**.

En la ventana *Inspector* de **Peldaño**, estableceremos los siguientes valores:

- Scale -> X: 20 Y: 0.2 Z: 20

Veremos como resultado una especie de plataforma. La idea es construir la escalera

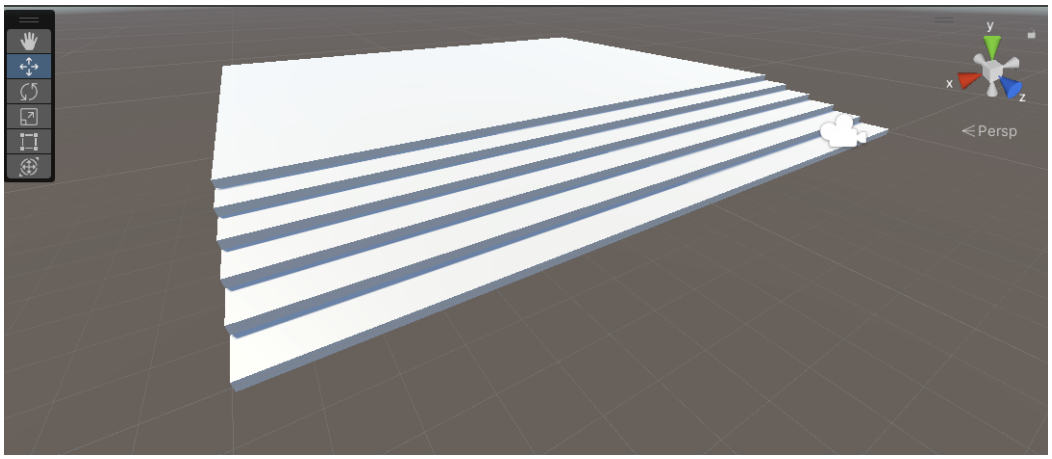


Figura 4.1: Modelo de una escalera de seis peldaños.

sobre este primer peldaño añadiendo nuevos mediante la duplicación de objetos (*Ctrl + D* o *Right Click ->Duplicate*). Generaremos el número de escalones que queramos (mínimo cinco más) que configuraremos de la siguiente manera de forma acumulativa:

- Position -> X: +0 Y: 0.2**num_peldaño* Z: -0.5**num_peldaño*
- Scale -> X: 20 Y: 0.2 Z: 20-*num_peldaño*

El resultado debería ser parecido a la mostrado en la imagen que se muestra en la Figura 4.1.

Ejercicio

El modelo que defina el entorno puede hacerse todo lo complejo que se quiera y de cara a la práctica de la asignatura seguramente lo sea. Como ejercicio complementario **se pide agregar dos paredes laterales de diez metros de alto y un techo con las mismas dimensiones que el suelo al modelo de escalera propuesto**. Todas la paredes creadas irán dentro de un objeto vacío **Paredes**, y el techo en otro objeto vacío llamado **Techo**. Una vez creados, los meteremos en otro objeto vacío llamado **Estructura** junto con el elemento **Escalera**. De esta manera se crea una jerarquía de objetos más intuitiva que nos permitirá navegar por la escena a medida que su complejidad aumente.
Seguramente, tras crear el techo, se proyectará sombra sobre gran parte de la escalera, si no en su totalidad. De cara a una mejor visualización de los resultados, cambiaremos la posición de la **Luz Direccional** configurándola con los siguientes parámetros:

- Position -> X: 2 Y: 12 Z: 33
- Rotation -> X: 36 Y: 128 Z: -50
- Scale -> X: 1 Y: 1 Z: 1

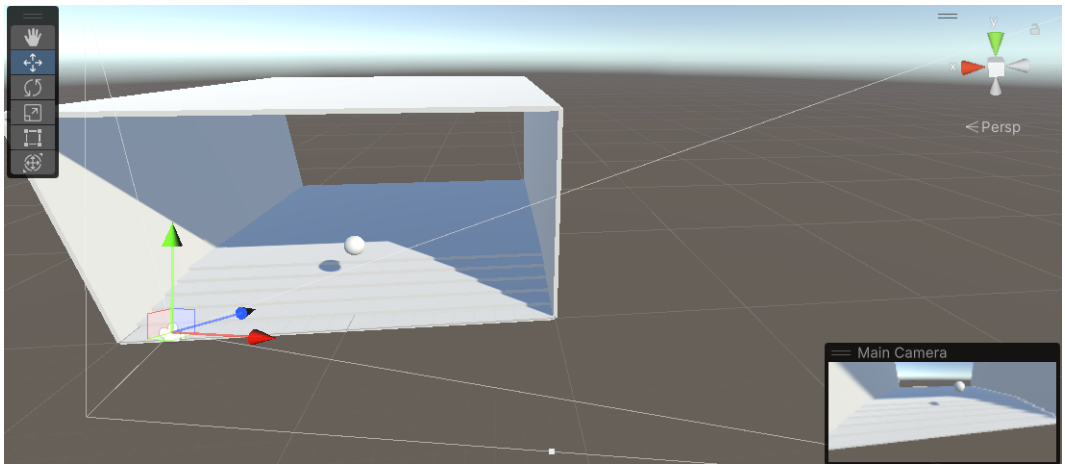


Figura 4.2: Modelo de ejemplo después de configurar la cámara principal.

4.4 Posicionamiento de la cámara

El elemento de *Hierarchy* **Main Camera** representa la vista del usuario cuando se ejecute la escena. Está representado por un símbolo de cámara en la escena y si lo seleccionamos aparecerá una mini ventana que nos muestra la vista previa desde su posición. Esta vista previa es equivalente a la vista que se muestra en la pestaña *Game*.

El elemento *Camera* es un objeto como cualquier otro y podremos modificar sus características a nuestra conveniencia obteniendo diferentes perspectivas. La excepción es la modificación del tamaño, donde no veremos ningún cambio sobre el elemento.

Antes de continuar, vamos a crear un objeto *Sphere*, que simulará un balón, y que colocaremos sobre la vertical del escalón superior. El objetivo es modificar las configuración de la cámara para que enfoque tanto la esfera como las escaleras.

También se puede mover la cámara para alinearla con tu vista de escena actual. Para hacerlo, se selecciona en la ventana *Hierarchy* y luego presiona *Ctrl + Shift + F* o haciendo *Right Click ->Align With View* .

Por ejemplo una configuración posible de la posición de la cámara es la siguiente:

- Position -> X: 3 Y: 5 Z: 15
- Rotation -> X: 23 Y: -170 Z: 0
- Scale -> X: 1 Y: 1 Z: 1

Como resultado, deberíamos tener algo similar a lo que aparece en la Figura 4.2. Se podrá variar la colocación de la cámara más tarde para poder visualizar correctamente los posteriores ejercicios.

4.5 Objetos interactuando con otros objetos

Por defecto, ningún objeto que creemos en nuestra escena tendrá masa, por lo que en consecuencia no se verá afectado por la fuerza de la gravedad. Para que un objeto de la escena se comporte como un objeto físico del mundo real, se le debe otorgar propiedades físicas agregándole un componente *Rigidbody*.

Para agregar un componente *Rigidbody* seleccionamos el elemento esfera (a partir de ahora **Balón**) en la ventana *Hierarchy* o directamaente en la escena (en este caso si la escena es muy compleja puede resultar difícil) y en *Inspector* haremos *Left Click* en el botón *Add Component* que está situado en la parte inferior de la ventana (posiblemente se tenga que hacer *scroll* hacia abajo si no se ve). Aparecerá una ventana de búsqueda entre todos los componentes que se pueden agregar. Buscaremos *Rigidbody* y lo añadiremos.

En nuestra ventana *inspector* tendremos el nuevo componente y deberemos asegurarnos que esté marcada la opción *Use Gravity*. Para comprobar como afecta a nuestra escena la ejecutamos utilizando el botón *Play* de la parte superior del Editor.

El **Balón** caerá por los escalones y rodará hasta que caiga al infinito desde el último escalón. Si el **Balón** se queda quieto en un escalón deberemos modificar su posición para que acabe rodando.

Ejercicio

Para comprobar como interactúan los objetos entre sí **se pide agregar otras primitivas básicas a la escena añadiéndoles el componente *Rigidbody***. Estos nuevos elementos serán colocados en diferentes posiciones de la escena.

4.6 Materiales

En la práctica anterior ya vimos el concepto de Material y conceptos básicos sobre cómo cambiarlos o editarlos. En esta práctica profundizaremos en estos conceptos y añadiremos más herramientas de cara al tratamiento de los materiales de cara a simular efectos más realistas.

Los archivos de materiales no dejan de ser imágenes que simulan dicho material. Por ejemplo, en la web de la tienda de Unity `https://assetstore.unity.com/` o en `https://freepbr.com/` se pueden encontrar archivos de materiales gratuitos que podemos utilizar en nuestros proyectos. Existen muchas webs diferentes donde poder conseguir texturas para nuestro modelos de manera gratuita o de pago. Hay que tener en cuenta que el desarrollo de materiales conlleva cierto trabajo y si necesitamos alguno en particular es siempre más fácil buscarlo que hacerlo desde cero.

Para importar un material a nuestro proyecto primero debemos crearlo. **Revisar la creación de materiales en la Práctica 1**. Cuando tenemos creados los materiales, abrimos en el explorador de archivos el directorio y copiamos en el los archivos de las

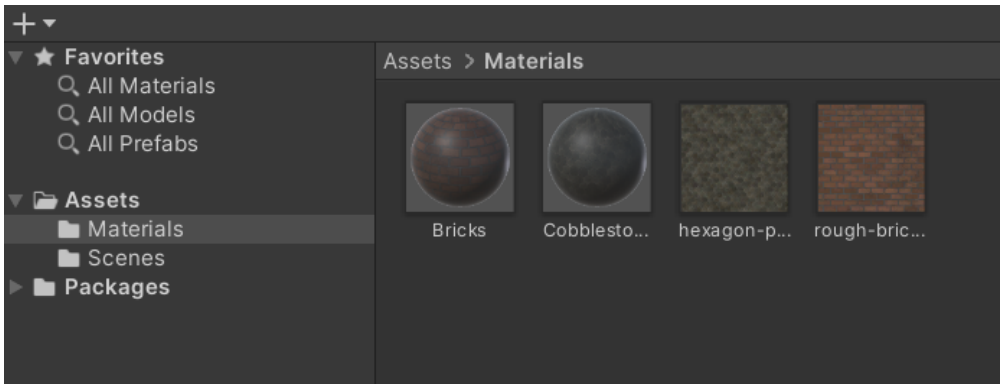


Figura 4.3: Elementos contenidos en la carpeta *Materials* del proyecto de Unity.

texturas.

Ejercicio

Buscar y descargar dos materiales en alguna de las páginas propuestas anteriormente que puedan usarse en Unity para representar las texturas de de la estructura de nuestra escena. Uno de los materiales se utilizará para los **Peldaños** y otro para **Paredes y Techo**. Ambos materiales deberán ser añadidos a nuestro proyecto.
Además crearemos un tercer material y lo editaremos sus propiedades para darle color y reflectancia.

4.6.1 Asociar la textura al material y al objeto

Seleccionando uno de los materiales nuevos, en su ventana *Inspector* buscamos la característica *Albedo*. Justo al lado hay un icono de círculo. Al hacer click en él podremos seleccionar la imagen de textura que hemos copiado en la carpeta buscándola por su nombre.

Una vez tengamos asociados el material con la textura que queramos solo debemos arrastrar el material al objeto que queramos de nuestra escena para asociarlo a este.

Para modificar el tamaño del material en el objeto, abre el componente del material seleccionado en la ventana *Inspector* y cambia las propiedades *Tiling X* e *Y*. Los números más bajos hacen que los ladrillos se vean más grandes y los más altos que se vean más pequeños.

Previamente, se agregó un componente *Rigidbody* al elemento **Balón** para otorgarle propiedades físicas. También se puede agregar características físicas con otro tipo de material llamado *Physic Materials*. Mediante este tipo de materiales se pueden cambiar propiedades como la fricción o el arrastre siempre y cuando el objeto se vea afectado por la gravedad. La diferencia entre un *Material* y un *Physic Material* es que estos últimos no implican un cambio en el aspecto visual de nuestros objetos.

Para hacer que nuestro **Balón** rebote, crearemos un *Physic Material* llamado *BouncyBall*. Para asociarlo a nuestro objeto, en la ventana *Inspector* debería haber ya una propiedad *Collider* (en este caso *Sphere Collider* aunque dependerá de la primitiva que se haya usado inicialmente), y arrastramos este material a la propiedad de mismo nombre.

Si ejecutamos nuestra escena veremos como ha cambiado el efecto del **Balón** sobre los **Peldaños**.

Ejercicio

Modificar las propiedades del **Balón** y de los materiales asociados a la **Estructura** de diversas formas y comprobar cómo afecta a la escena. Igualmente modificaremos las propiedades del elemento *Directional Light* para cambiar su posición, color e intensidad para comprobar cómo afectan a los materiales utilizados.

4.7 Prefabs

Un inconveniente de una escena compleja es el número creciente de objetos que se añaden. Normalmente muchos de ellos serán copias de uno en particular, por ejemplo nuestros **Peldaños** o las **Paredes**. En nuestra escena sigue siendo un número manejable pero ya se empieza a volver incómodo cambiar las propiedades de un objeto por igual a todos los elementos similares.

Es por ello que se utilizan los objetos *Prefabs*. Deberemos considerarlos como elementos plantilla de los cuales podremos crear un número indeterminado de **instancias**. Cuando cambiemos las propiedades de objeto *Prefab*, los cambios se reproducirán automáticamente en todas sus instancias.

Para ejemplificarlo, vamos a generar un *Prefab* de nuestro **Balón**.

En la ventana *Project*, dentro de *Assets* crearemos una nueva carpeta que llamaremos *Prefabs* y entraremos en ella aunque de momento esté vacía. Solo deberemos arrastrar el obeto **Balón** a esta carpeta para convertirlo en *Prefab*. Se verá reflejado en la ventana *Hierachy* porque el icono pasará a ser de color azul.

Para añadir nuevas instancias solo es necesario arrastrar el *Prefab* desde la carpeta del proyecto a nuestra escena tantas veces como se quiera.

Se pueden modificar las instancias por sí solas o el *Prefab* en general si queremos transmitir el cambio a todas ellas. En la ventana *Inspector*, en la parte superior, está el menú *Overrides* que permite ver los elementos cambiados en esa instancia y revertirlos si es necesario a los valores del *Prefab*.

Igualmente se pueden crear variaciones de un *Prefab* arrastrando una instancia modificada a la carpeta de *Prefabs* en la ventana *Project*. En la ventana *Hierarchy* aparecerá reflejado como un icono azul con una de las caras rayada.

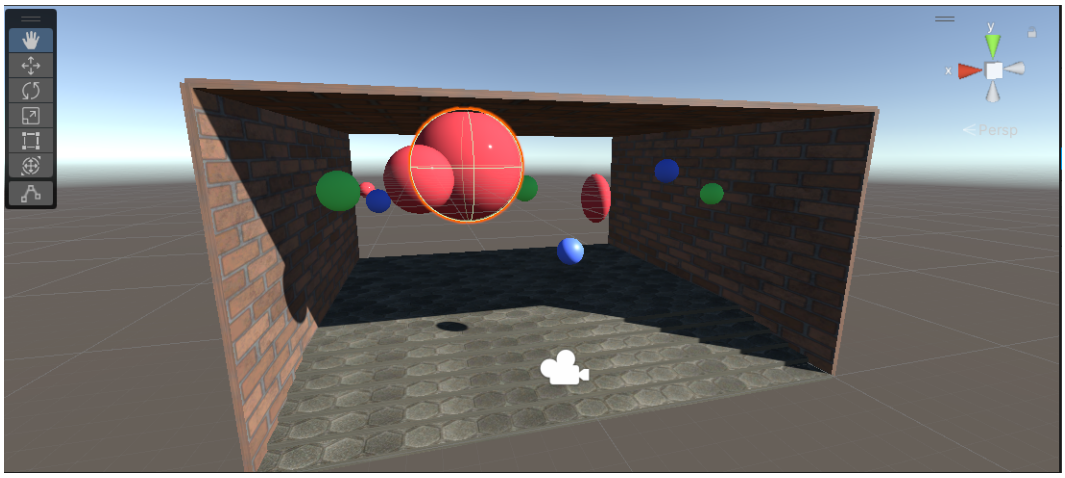


Figura 4.4: Resultado esperado del ejercicio propuesto en esta práctica.

Ejercicio

Cambiar el tamaño, color y el rebote de las instancias de **Balón** para hacerlas diferentes unas de otras. Al terminar ejecutamos la escena y vemos el resultado que debería ser similar al de la Figura 4.4.

4.8 Luces

Hasta ahora hemos trabajado con un único elemento iluminador en nuestra escena. Este elemento simula la luz proporcionada por el sol en nuestra escena, pero dependiendo de como lo hayamos colocado nuestra estructura generará más o menos sombras.

Este tipo de luz se conoce por *Directional Light* y podemos ver sus propiedades en la ventana *Inspector* (Figura 4.5).

Aquí podremos cambiar su posición y orientación como cualquier otro objeto de nuestra escena. Pero resulta más interesante las propiedades de la luz en si, por ejemplo el color o su intensidad.

En Unity, como en la vida real, podemos tener más de un foco de luz al mismo tiempo, pero si queremos simular una escena en las mismas condiciones que se dan en nuestro planeta es poco realista tener más de un Sol.

Los tipos de elementos generadores de luz son los siguientes:

- **Directional**: una luz que se encuentra infinitamente lejos y emite luz en una sola dirección
- **Spot**: una luz que se encuentra en un punto de la escena y emite luz en forma de cono.
- **Point**: una luz que se encuentra en un punto de la escena y emite luz en todas las direcciones por igual

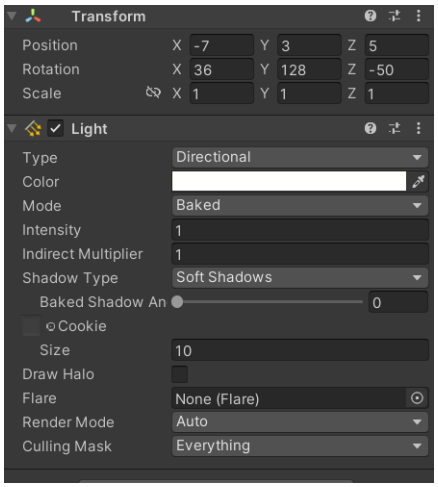


Figura 4.5: Ventana *Inspector* de un objeto del tipo luz.

- **Area**: una luz que está definida por un rectángulo o disco en la escena y emite luz en todas las direcciones uniformemente a lo largo de su superficie, pero solo desde un lado del rectángulo o disco.

Ejercicio

Crear un objeto vacío **Luces** donde guardaremos todos los objetos emisores de luz de nuestra escena. Añadir a la escena distintos tipos de luz y cambiar sus propiedades para iluminar los espacios en sombra que genere el objeto *Directional Light* que ya teníamos en el proyecto (si no generaba sombras, colocarlo para que así lo haga).

4.9 Construir y publicar un proyecto

Para convertir nuestra escena en una aplicación independiente deberemos construirla antes. Unity ofrece la posibilidad de generar aplicaciones ejecutables de escritorio a través de la opción *Build and Run* del menú *File* del Editor. Mediante esta opción, solo necesitaremos marcar el directorio destino donde se generará el binario de nuestra aplicación.

Por otra parte existe la posibilidad de publicar nuestra aplicación y compartirla con la comunidad a través del complemento *WebGl*. Para añadir este complemento a nuestro proyecto abriremos el *Package Manager* (Figura 4.6) que se encuentra en el menú *Window* del Editor.

Nos aparecerá un gestor de paquetes indicando cuáles tenemos instalados en el proyecto actualmente. Para añadir uno nuevo en la pestaña *Packages* seleccionaremos *Unity Register* donde tendremos acceso a todos los paquetes disponible en el Registro de Unity. Buscaremos el paquete *WebGL* y lo instalaremos en nuestro proyecto. Una

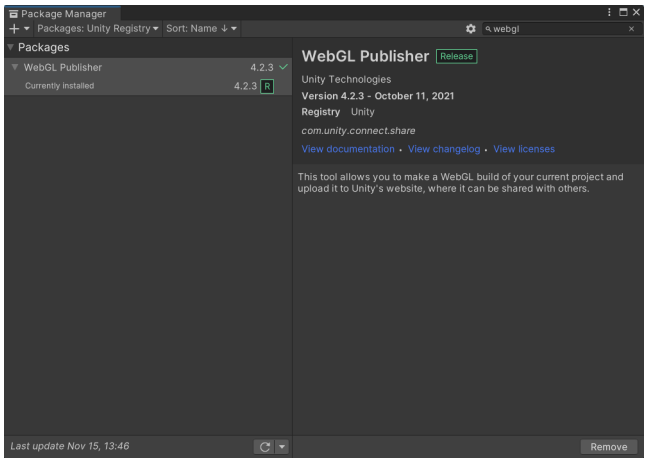


Figura 4.6: Instalador de paquetes del Unity.

vez terminado, salimos del gestor de paquetes y deberemos ver un nuevo menú *Publish* en la barra del Editor.

Si entramos en el menú solo aparece una opción donde haremos click y aparecerá la ventana *Publish*. En ella tendremos la opción de construir el proyecto y publicarlo, y la opción de localizar un proyecto existente. Si tratamos de construir el proyecto como web, nos dará un error similar al mostrado a continuación.

Error building Player: Incompatible color space with graphics API

WebGL a menudo carece de la extensión adecuada para admitir la lectura de datos de textura de forma lineal. Debido a esa limitación, es imposible admitir la representación en el espacio lineal.

Para solucionarlo vamos a *File ->Build Settings ->Plater Settings ->Other Settings ->Color Space* y seleccionaremos *Gamma* en vez de *Linear*. Esto provocará un cambio y permitirá realizar la constucción del proyecto con WebGL.

Puede ser también que al publicar el proyecto de un error del tipo:

HTTP/1.1 500 internal Server Error

Se produce al parecer cuando los servidores de Unity están caídos y solo se soluciona cuando vuelven a estar activos.

De cara a la entrega de la práctica de la asignatura solo será necesario la aplicación de escritorio del proyecto aunque el alumno podrá añadir a la entrega enlaces a la versión web de su proyecto.

4.10 Conclusiones

En esta práctica se ha profundizado en el manejo y construcción de objetos más complejos donde distintos elementos interaccionan entre ellos. También se ha aprendido a añadir propiedades físicas a los objetos con el fin de simular características reales, así como generar plantillas para hacer más sencillo el proceso de modificación de un número elevado de objetos similares entre si. En otro punto se ha aprendido a utilizar los objetos lumínicos que permiten proyectar luces y sombras sobre nuestra escena de forma básica. Por último se han dado las instrucciones necesarias para poder generar nuestras aplicaciones a partir del proyecto.

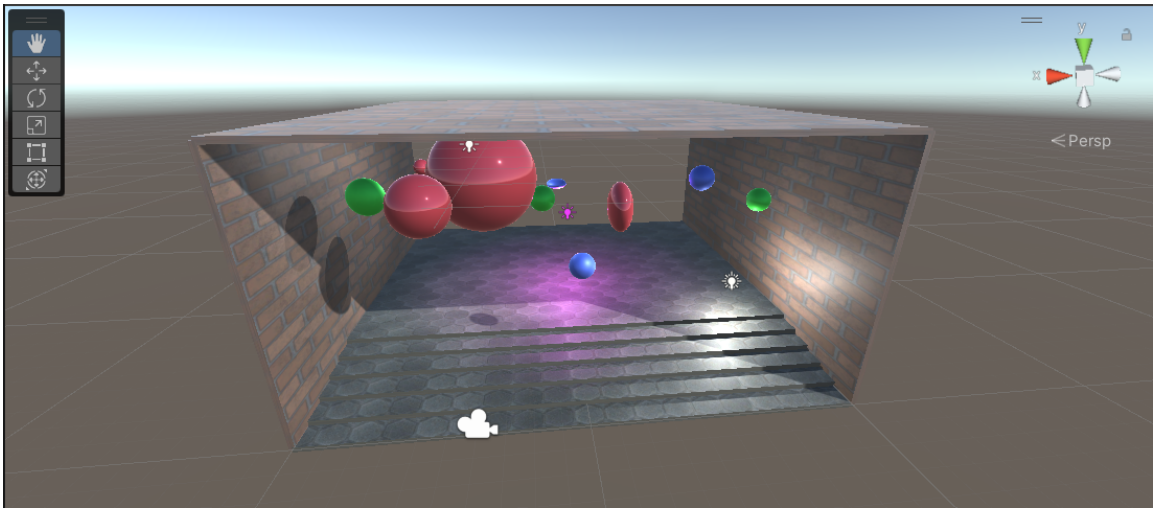


Figura 4.7: Aspecto final del proyecto seguido durante esta práctica.

5. Scripting

5.1 Objetivo de la práctica

En esta práctica aprenderemos los fundamentos de la creación y edición de scripts para Unity. También exploraremos las funciones más interesantes de las librerías de Unity para C# de cara a la realización de nuestro proyecto.

5.2 Requisitos previos

El alumno debe haber terminado con éxito las prácticas anteriores. Además, **un conocimiento básico de lenguajes de programación de alto nivel es recomendable** para la realización del resto de prácticas. Unity utiliza C# para la programación de comportamientos mediante scripts. Si el alumno no conoce C#, o ningún lenguaje derivado de C, se recomienda que antes de afrontar la práctica se realice algún tutorial básico de programación donde se aprenda la creación de variables, los tipos de datos, POO, etc.

5.3 Creación y composición de los scripts

Los scripts van a ser una herramienta muy útil y muy poderosa en el desarrollo de nuestra escena. Nos ofrecen, entre otras cosas, la posibilidad de inferir cambios en el estado de nuestros objetos durante su ejecución. Estos scripts se harán utilizando el **lenguaje de programación C#** por lo que si durante la instalación de Unity no se agregó el módulo de *Visual Studio* es recomendable instalarlo aunque no obligatorio. Se puede utilizar cualquier IDE que soporte C# o cualquier editor de texto como *Notepad++*, *Gedit*, etc.

El nivel necesario de conocimiento de este lenguaje de programación dependerá

siempre del objetivo que queramos realizar, y de cara a la realización de las prácticas, aprenderemos a utilizar las librerías propias de Unity. Toda la información referente a los componentes de Unity, incluido lo referente a los elementos de programación propios del motor pueden consultarse en `https://docs.unity3d.com/es/530/Manual/UnityManual.html`.

La creación de un script es muy sencilla. No deja de ser un componente que forma parte de un objeto y como tal **puede haber más de un script asociado a un objeto**. En nuestra ventana *Project* dentro del directorio *Assets* vamos a crear una nueva carpeta llamada *Scripts*. Evidentemente, es en esta ruta donde guardaremos todos los scripts que necesitemos a la hora de elaborar la escena.

Dentro de *Scripts* hacemos *Right click ->Create ->C# Script* lo cual nos creará un archivo de extensión *.cs*. Es recomendable nombrar los scripts de tal manera que sea fácil identificar su función, por ejemplo *MoveBall.cs*, nos va a permitir realizar un movimiento de translación del objeto **Balón** de nuestro proyecto.

Una vez creado habrá que editarlo. Para ello debemos conocer sus partes. Haciendo doble click en el fichero se abrirá automáticamente *Visual Studio* si lo tenemos instalado. En caso contrario, solo es necesario abrirlo desde cualquier otro editor.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class NombreClase : MonoBehavior
6  {
7    //Start is called before the first frame update
8    void Start()
9    {
10
11   }
12
13   // Update is called one per frame
14   void Update()
15   {
16
17   }
18 }
```

Como se puede ver en la en el código anterior, el script por defecto una vez creado contiene **una clase que se llamará igual que el script** con dos métodos ya implementados: *start* y *update*.

- *Start*. Este método se va a llamar al comienzo de la ejecución de la escena solo una vez. Lo utilizaremos para inicializar determinadas variables o valores que vayamos a necesitar.
- *Update*. Este método se llama al comienzo de cada *frame* de la escena. Por defecto, hay 24 *frames* por segundo, por lo que deberemos escalar los valores que utilicemos en base a este dato.

Cada vez que se modifique el archivo *.cs* y se guarde, al volver al Editor de Unity, este recargará el proyecto para coger los cambios realizados en los scripts. Para asociar

nuestro script a un objeto concreto, seleccionamos dicho objeto en la ventana *Hierarchy* y desde la ventana *Project* arrastramos el script hasta la ventana *Inspector* del objeto. Automáticamente nos creará un componente del tipo *Script* con nuestro fichero ya asociado.

Otra manera de crear un script, es directamente añadir un nuevo componente al objeto que queramos del tipo *New Script* donde nos dará opción a poner el nombre que queramos. A diferencia del método anterior, este script se creará automáticamente en la carpeta *Assets*.

Ejercicio

Vamos a crear un script asociado al objeto **Balón** con el nombre *TransformBall*. En el método *Start* añadiremos la siguiente línea de código:

```
1  Debug.Log("Metodo Start ejecutado.");
```

Guardamos el cambio y ejecutaremos nuestra escena. Podremos ver el mensaje en la pestaña *Console* de la ventana *Project*.
Volviendo al script añadimos al método *Update* la siguiente línea de código y volvemos a ejecutar comparando el resultado.

```
1  Debug.Log("Metodo Update ejecutado.");
```

5.4 Transformaciones básicas de objetos mediante scripts

En este apartado vamos a aprender a modificar las propiedades de los objetos de nuestra escena. Para ello es necesario conocer qué propiedad se quiere modificar y como se refleja en nuestro script. En nuestro caso, es la propiedad *Transform* que está presente en todos los objetos donde podemos modificar la posición, la rotación y la escala de un objeto.

En todos los casos necesitamos una variable del tipo *Vector3* para introducir el nuevo valor.

```
1  public Vector3 changePos;
2  public Vector3 changeSca;
3  public Vector3 changeRot;
```

Esta variable será un atributo de nuestra clase. Si guardamos y volvemos al Editor, veremos que en el componente *Script* del objeto **Balón** ahora refleja la nueva variable. El valor de esta variable se puede definir por código o directamente desde el Editor.

```
1 void Start()
2 {
3   transform.position = changePos;
4   transform.scale = changeSca;
5 }
```

Modificando el método *Start* de esta manera podremos asignar el tamaño y posición inicial de nuestro objeto con los valores introducidos en el Editor.

La rotación de un objeto funciona de manera ligeramente diferente a la escala o posición. Se realiza a través de la función *Rotate*.

```
void Start()
{
  transform.Rotate(changeRot);
}
```

Ejercicio

Hacer varias pruebas con distintos valores para visualizar los cambios. Una vez hechos, modificar el script para que en vez de configurar los valores iniciales, se introduzca el incremento de la posición y tamaño del **Balón** a medida que avance la escena. Como modificar la rotación de una esfera puede ser poco visual, añadir un nuevo script al objeto **Estructura** que haga que rote sobre el eje X.
Hacer distintas pruebas teniendo en cuenta el número de *frames* por segundo a la hora de introducir los valores.
Si se ha realizado bien el ejercicio, veremos como las bolas que habíamos creado caen al infinito al poco tiempo. Igualmente, los objetos como las luces permanecen fijas respecto a la rotación de la estructura.
Modificar el modelo, completando las paredes que faltan y hacer que las luces estén fijas a la estructura a medida que rota.
Por último, si cerramos la estructura, no veremos el comportamiento de los balones al ejecutar ya que la cámara está fuera. Modificar la cámara para ver como los balones rebotan dentro de la estructura.

5.5 Crear y Destruir Objetos desde un script

Puede ocurrir que para determinadas situaciones la existencia de un objeto no comience o termine a la vez que nuestra escena. Por ello debemos saber como crear y destruir objetos a través de un script.

Una manera de hacerlo es instanciando *prefabs*, que no deja de ser un objeto dentro de nuestro proyecto. Para ello crearemos un script que asociaremos a un objeto de nuestro proyecto, no necesariamente visible en nuestra escena. En el ejemplo a continuación codificaremos la creación y destrucción de un objeto del tipo **Balón** en nuestro objeto **Balones**. Lo haremos como métodos de la clase que luego podremos llamar desde cualquier otro sitio.

```
public GameObject prefab;
private GameObject nuevo_balon;

void create_ball()
{
  nuevo_balon = Instantiate(
    prefab, // Clase que se instancia
    new Vector3(1,1,1), // Posicion donde queremos que se cree
    Quaternion.identity, // Rotacion del objeto. Quaternion.identity es un conjunto de valores vacio.
     Utilizaremos este valor para conseguir un objeto sin rotacion.
```

```
10
11      transform // Establecemos la clase padre, el objeto al que esta asociado el script
12    );
13    nuevo_balon.name = "Balon"; // Asignamos un nombre al objeto
14  }
```

Para destruir un objeto es mucho más sencillo.

```
1  void destroy_ball(GameObject ball)
2  {
3    Destroy(ball); // Si queremos retrasar la destruccion del objeto un periodo de tiempo Destroy acepta
         un atributo float: Destroy(ball, 10.0f); Destruiria el Balon a los 10 segundos
4  }
```

Ejercicio

Elimina todos los balones de la escena. Además, crear tres balones en distintos puntos dentro de la **Estructura** desde el script y destruirlos pasados tres, cinco y diez segundos respectivamente.

5.6 RayCast

Cualquier punto en la vista de la cámara corresponde a una línea en el espacio del mundo. Algunas veces es útil tener una representación matemática de esa línea y Unity provee esto en la forma de un objeto *Ray*. El *Ray* siempre corresponde a un punto de la vista, entonces la clase de la cámara proporciona las funciones *ScreenPointToRay* y *ViewportPointToRay*.

La diferencia entre las dos es que *ScreenPointToRay* espera que el punto sea proporcionado como una coordenada de píxel, mientras que *ViewportPointToRay* toma unas coordenadas normales en el rango $0 \ldots 1$ (dónde 0 representa la parte inferior o izquierda y 1 representa la parte superior o derecha de la vista). Cada una de estas funciones devuelve un *Ray* que consiste en un punto de origen y un vector que muestra las direcciones de la línea desde ese origen. El *Ray* se origina desde la vista de la cámara en vez del punto *transform.position* de la Cámara.

Todo esto está muy bien, pero **¿Cómo lo utilizamos y para qué?**

El uso más común de un *Ray* de un objeto es realizar un ***raycast***. Un *raycast* manda un *laser beam* imaginario desde su origen hasta que choca con un *Collider* en la escena. La información acerca del objeto y el punto de choque es devuelta en un objeto *RaycastHit*. Esto es una manera muy útil para ubicar un objeto basado en su imagen de pantalla.

Pero antes de eso, necesitamos conocer el concepto de ***Collider***. Los componentes Collider definen la forma de un objeto a los efectos de las colisiones físicas. Un *Collider*, que es invisible, no necesita tener exactamente la misma forma que la malla del objeto y, de hecho, una aproximación a dicha forma suele ser más eficiente e indistinguible en el juego.

Ahora si, por ejemplo, el objeto determinado por la posición del ratón puede ser determinado con el siguiente código.

```
public Camera camera;

void Update(){
  RaycastHit hit;
  Ray ray = camera.ScreenPointToRay(Input.mousePosition);

  if (Physics.Raycast(ray, out hit)) {
    Transform objectHit = hit.transform;
    // ...
  }
}
```

Ejercicio

Mostrar por consola la posición donde se hace click en la escena.

5.7 Eventos

Utilizando lógica de programación y otras funciones que están disponibles en la documentación de Unity se pueden realizar cambios en los objetos, como puede ser variar la dirección de un objeto al pasar un periodo de tiempo determinado. Por ejemplo en este caso sería útil saber en que *frame* nos encontramos con ***Time.frameCount***.

```
static void function()
{
  if (Time.frameCount % 10 == 0)
    changePos *= -1;
}
```

Una escena es una animación dónde los *frames* son generados sobre la marcha. Un concepto clave en la programación en Unity es el de hacer cambios a la posición, estados, y comportamientos de objetos en el juego justo antes de que cada *frame* sea renderizado. Como ya hemos visto anteriormente, la función *Update* es el lugar principal para este tipo de código en Unity. *Update* es llamado antes de que el *frame* sea renderizado y también antes de que las animaciones sean calculadas.

El motor de la física de la escena también actualiza en pasos de tiempo discretos de una manera similar al renderizado de *frames*. Una función separada de evento llamada *FixedUpdate* se llama justo antes de cada actualización de física. La diferencia entre el refresco de *frames* y el refresco del sistema de física reside en que el primero depende de factores .externos", como la carga de la CPU, mientras que el segundo se produce en un *timestep* establecido (por defecto por Unity pero editable por el desarrollador) por razones de precisión y consistencia de la simulación.

Debido a que las actualizaciones de física y actualizaciones de *frames* no ocurren en la misma frecuencia, vamos a obtener un resultado más preciso de código de física si la codificación la realizamos en la función de *FixedUpdate* en vez de *Update*.

```
void FixedUpdate()
{
  transform.position = changePos;
  transform.scale = changeSca;
}
```

También es útil ser capaz de hacer cambios adicionales en un punto después de que funciones *Update* o *FixedUpdate* hayan sido llamadas para todos los objetos en la escena y después de que todas las animaciones hayan sido calculadas. Un ejemplo de uso dónde una cámara debería permanecer mirando un objeto destino; el ajuste a la orientación de la cámara debería ser hecho después de que el objeto destino haya sido movido. La función *LateUpdate* puede ser utilizada para este tipo de situaciones.

```
void LateUpdate()
{
  Camera.main.transform.LookAt(target.transform);
}
```

A veces es útil ser capaz de llamar código de inicialización con anterioridad a cualquier actualización que ocurra en el tiempo de juego. La función *Start* es llamada antes de la actualización del primer *frame* o física de un objeto, pero ¿Qué ocurre si la inicialización de un objeto es demasiado pesada? Pues que va a afectar a como percibimos la fluided de la escena. Para solucionar esto tenemos la función *Awake* es llamada para cada objeto en la escena en tiempo de carga de la escena. Todos los *Awakes* habrán finalizado antes de que el primer *Start* sea llamado.

```
void Awake()
{
  // Codigo anterior a la funcion Start
}
```

5.7.1 Eventos GUI y de Física

Aunque esto nos ofrece muchas posibilidades, posiblemente en nuestro proyecto habrá veces que queramos que nuestra escena reaccione a interacciones del usuario o entre los propios objetos. Ahí es donde entran los eventos GUI y de Física.

Los eventos de la interfaz de usuario o GUI son aquellos que requieren de una acción del usuario para activarse. Puede ser un click del ratón o el desplazamiento de los dedos en una pantalla táctil. Unity tiene un sistema para renderizar los controles GUI sobre la acción principal en la escena y responder a clicks hechos por estos controles.

Existe un conjunto de funciones de evento OnMouseXXX que están disponibles para permitir a un script reaccionar a las acciones del usuario con el mouse. Para entenderlos adecuadamente debemos definir antes el concepto de *Collider*, que son los componentes que definen la forma de un objeto para los propósitos de colisiones físicas. En otras palabras, son un elemento invisible que está asociado al objeto y que define es espacio que ocupa en la simulación, que no tiene que ser necesariamente el mismo que el del objeto pero que cuanto mejor se ajuste más realismo transmitirá. En la Figura

5.1 podemos ver un ejemplo de *Collider* mucho más amplio que el objeto al que está asociado.

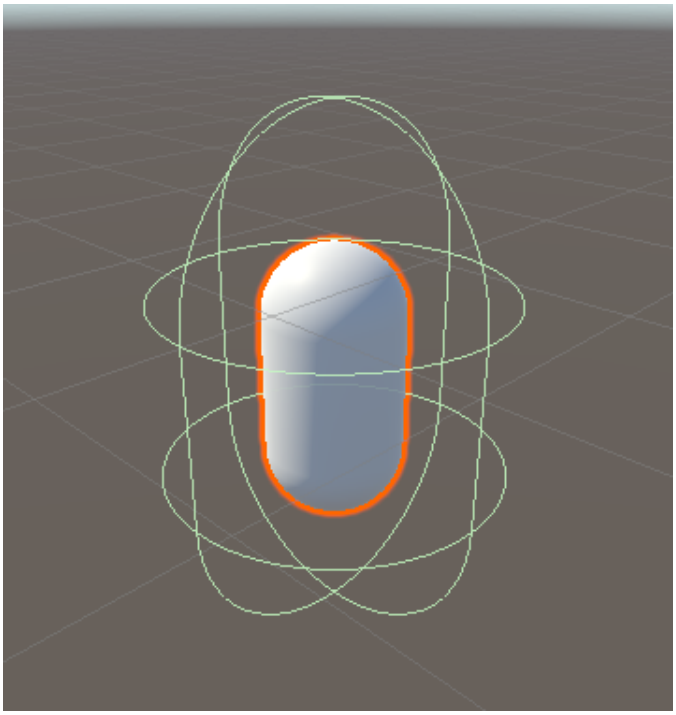


Figura 5.1: Ejemplo de *Collider asociado a un objeto Cylinder*.

- **OnMouseDown** se llama cuando el usuario ha presionado el botón del ratón mientras estaba sobre el *Collider*.
- **OnMouseDrag** se llama cuando el usuario mantiene presionado el click del ratón sobre el *Collider*.
- **OnMouseEnter** es llamado cuando el ratón entra en el espacio del *Collider*.
- **OnMouseExit** es llamado cuando el ratón sale en el espacio del *Collider*.
- **OnMouseOver** es llamado mientras el ratón está en el espacio del *Collider*.
- **OnMouseUp** es llamado cuando se deja de hacer click en el ratón.
- **OnMouseUpAsButton** solo se llama cuando se suelta el ratón sobre el mismo *Collider* que se presionó.

De la misma manera también podemos **interaccionar a través del teclado** mediante la función:

```
1 Input.GetKey(_tecla_); // ej: Input.GetKey(KeyCode.A);
```

Es recomendable conocer y familiarizarse con este tipo de eventos en este punto, aunque los utilizaremos más ampliamente en la práctica dedicada a la interfaz de usuario.

El motor de física va a reportar colisiones contra un objeto al llamar funciones de evento en el script de ese objeto. Las funciones ***OnCollisionEnter, OnCollisionStay*** y ***OnCollisionExit*** serán llamadas a medida que haya un contacto hecho, mantenido o terminado.

Las funciones correspondientes ***OnTriggerEnter, OnTriggerStay* y *OnTriggerExit*** serán llamadas cuando el *Collider* del objeto esté configurado como un *Trigger* (un *Collider* que simplemente detecta cuando algo le afecta en vez de reaccionar físicamente). Estas funciones pueden ser llamadas varias veces de forma sucesiva si un contacto o varios son detectados durante la actualización de física. Entonces, un parámetro es pasado a la función dando detalles de la colisión.

Ejercicio

Probar lo aprendido mediante las siguientes condiciones que deben ser llevadas a código:

- Utilizar las teclas **WASD** del teclado para mover la cámara. Tendremos que utilizar un atributo nuevo al que llamaremos *speed* para indicar la velocidad a la queremos se mueva la cámara.
- Eliminamos todos los balones de la escena menos uno. Cada vez que hagamos click en el **Balón** se deberá crear una copia de él dentro de la escena. La copia deberá estar desplazada respecto al original lo suficiente para no colisionar.
- Cada vez que dos balones choquen entre si se destruyen.

5.8 Añadir audio a nuestra escena

Los componentes de audio son elementos muy importantes en nuestro proyecto. Pueden ayudar a reconocer situaciones, marcar el comienzo o final de interacciones del usuario con el juego, o simplemente decorar.

Para ejemplificar como añadir una fuente de audio a nuestro proyecto vamos a crear un objeto vacío situado en el origen de coordenadas que vamos a llamar **SonidoAmbiente**. A este objeto debemos añadirle un componente *Audio Source*. Para continuar necesitamos ficheros de audio. En el Campus Virtual hay disponibles algunos archivos de audio que son los que vamos a utilizar en esta parte del guión.

En la Figura 5.2 podemos ver parte de las opciones que ofrece este componente. Nos van a interesar sobre todo los siguientes campos.

- **AudioClip**. Aquí añadiremos el archivo de audio concreto que vamos a reproducir.
- **Play On Awake**. Al marcar esta casilla le diremos a Unity que el archivo de audio comienza a reproducirse en el método *Awake* del objeto.
- **Loop**. Al marcar la casilla nos permite reproducir el archivo en bucle.
- **Volume**. Nos permite ajustar el volumen del audio.

Ejercicio

Añadir al componente **SonidoAmbiente** el archivo de audio *forest* y probar diferentes configuraciones.

Figura 5.2: Detalle del componente Audio Source de un objeto de la escena.

5.8.1 Adjuntar sonidos a los eventos de la escena

Añadir música de fondo ayuda a mejorar nuestro proyecto pero normalmente los archivos de audio están asociados a acciones que ocurren en situaciones concretas durante la ejecución de la aplicación. Ya hemos visto cómo se programan eventos dentro de nuestra juego, ahora vamos a mejorarlos asociando audio a cada uno de ellos. Para ello tenemos que saber cómo acceder a este tipo de componentes desde script.

```
1  private AudioSource audio;
2
3  void Start()
4  {
5    audio = GetComponent<AudioSource>();
6  }
```

Con este código podremos acceder al componente *Audio Source* de nuestro objeto. Pero ¿**qué ocurre cuando hay más de un archivo de audio asociado a un solo objeto?** Una manera de resolverlo es tener un script que acepte como parámetros cada archivo de audio que necesitemos. Luego, para comenzar o parar la reproducción del audio utilizaremos los métodos *Play* y *Stop*.

```
1  private AudioSource audio_m;
2  public AudioClip sound1;
3  public AudioClip sound2;
4
5  void Start()
6  {
7    audio_m = GetComponent<AudioSource>();
8    audio_m.clip = sound1;
9    audio_m.Play();
10   audio_m.Stop()
11   audio_m.clip = sound2;
12   audio_m.Play();
13 }
```

Como se puede comprobar esta solución no nos vale, ya que los audios no se reproducen completamente. *Play* es una función asíncrona, o lo que es lo mismo el script sigue ejecutando instrucciones sin esperar a que *Play* haya terminado. Una posible solución es reproducir cada audio de forma individual utilizando *PlayOneShot*.

```
private AudioSource audio_m;
public AudioClip sound1;
public AudioClip sound2;

void Start()
{
  audio_m = GetComponent<AudioSource>();
  audio_m.PlayOneShot(sound1);
  audio_m.PlayOneShot(sound2);
}
```

Ejercicio

Utiliza la función **PlayOneShot** para reproducir el audio *catch-ball* cuando se crea un nuevo **Balón** y el audio *bounce-ball* cuando se produce una colisión del **Balón**.

5.8.2 Audio Manager

Si empezamos a añadir componentes de audio a cada objeto de la escena ocurrirá en algún momento que tengamos demasiados y sea realmente difícil gestionarlos, por no decir que incrementará el peso de nuestra aplicación. Además, hay situaciones en las que deja de ser válida esta configuración, por ejemplo cuando queremos reproducir un sonido por la destrucción de un objeto.

Probamos a reproducir el sonido *explode-ball* cuando colisionan dos **Balones**. Unity nos dirá que no puede reproducir un audio desde un objeto que no existe. Esto ocurre porque hemos destruido el objeto antes de terminar la reproducción del audio, pero si esperamos a que finalice el efecto no es el deseado.

La solución a estas situaciones es la creación de un objeto *Audio Manager* para todo el proyecto. Este objeto se encargará de recibir del resto para reproducir los distintos audios, y como nunca se destruye (hasta el final de la ejecución) siempre podrá hacerlo. Vamos a ver como crear este tipo de objeto.

En primer lugar vamos a cambiarle el nombre a **SonidoAmbiente** por **AudioManager** y eliminamos el clip de *forest* del componente. En nuestra carpeta *Scripts* creamos uno nuevo que se llame *AudioManager* que asociamos a nuestro objeto vacio.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class AudioManager : MonoBehaviour
{
 public AudioSource EffectsSource;
 public AudioSource MusicSource;
```

```
public AudioClip Ambiance;
// Singleton
public static AudioManager Instance = null;

// inicializa la instancia Singleton
private void Awake()
{.
 if (Instance == null)
 {
  Instance = this;
 }
 else if (Instance != this)
 {
  Destroy(gameObject);
 }
 // Evitamos que el objeto AudioManager se destruya cuando se recarga la escena
 DontDestroyOnLoad(gameObject);
}

void Start()
{
 PlayAmbiance(Ambiance);
}

// Reproduce una sola vez un archivo de audio
public void PlayClip(AudioClip clip)
{
 EffectsSource.PlayOneShot(clip);
}

// Reproduce un archivo en bucle como ambiente.
public void PlayAmbiance(AudioClip clip)
{
 MusicSource.clip = clip;
 MusicSource.Play();
 MusicSource.loop = true; // Reproduce en bucle
}
}
```

El script requiere de dos componentes *Audio Source* por lo que los creamos y los pasamos por parámetros junto con el clip de audio que queremos como fondo, en nuestro caso *forest*. Esta clase es única para toda la aplicación y se puede acceder a sus métodos de la siguiente manera.

```
AudioManager.Instance.PlayClip(_clip_);
AudioManager.Instance.PlayAmbiance();
```

Ejercicio

Cambiar el script asociado al prefab **Balón** para que utilice el AudioManager para reproducir todos los sonidos propuestos anteriormente junto con el audio *explode-ball* cuando colisionan dos balones.

5.9 Conclusiones

Los scripts de Unity son una herramienta muy potente de cara al desarrollo de aplicaciones. En esta práctica hemos visto los fundamentos para la programación de scripts que realicen las tareas más significativas de cara a la realización de nuestro proyecto. Existen más opciones de cara a comportamientos más complejos que se invita a explorar a través de los recursos vistos en la primera práctica.

6. Interfaz de Usuario

6.1 Objetivo de la práctica

En esta práctica vamos a aprender a incluir componentes en nuestro proyecto que nos permitan interaccionar con él de forma más intuitiva. Menús, *widgets*, *labels* con información, etc son algunos de los componentes que vamos a tratar en esta práctica.

6.2 Requisitos previos

El alumno debe conocer los fundamentos de *Scripting* de Unity, así como el funcionamiento de los componentes de los objetos y los eventos que se pueden producir en una escena en Unity.

6.3 Elementos de la Interfaz

Antes de empezar a crear nuestra interfaz de usuario debemos conocer algunos conceptos elementales.

En primer lugar tenemos el objeto ***Canvas***. El *Canvas* es el área donde todos los elementos de interfaz deben estar. Es un objeto con una serie de particularidades, donde todos los elementos de interfaz deben ser hijos suyos. Al crear un nuevo elemento en la interfaz, tal como una imagen o un texto, automáticamente crea un *Canvas* si ya no había uno en la escena. El elemento es creado como un hijo de este *Canvas*. El área *Canvas* es mostrado como un rectángulo en la escena haciendo más sencillo posicionar los elementos.

Todos los elementos de la interfaz son objetos 2D aunque nuestro proyecto sea 3D.

Los objetos 2D se llaman ***Sprites***. En realidad se pueden identificar también con las texturas de los los objetos en cualquier proyecto, solo que en 2D corresponden al objeto en sí. Por ejemplo, un botón de nuestra interfaz es un objeto 2D asociado a una imagen de un botón.

Como parte del aprendizaje, vamos a crear un menú principal para nuestro juego o aplicación, con la salvedad de que va a ser una escena diferente a la que veníamos utilizando.

En nuestra carpeta *Scenes* creamos una nueva escena que llamamos *MainMenu*. En esta nueva escena configuramos la cámara cambiando las siguientes propiedades:

- *Clear Flags* ->*Solid Color* para establecer el color de fondo de la pantalla del menú. Más tarde veremos como poner una imagen de fondo.
- *Background* ->Elegimos un color
- *Culling Mask* ->UI. Esta propiedad nos permite elegir qué capas se mostrarán en la vista de la cámara. De momento dejaremos solo la capa UI.

Ahora creamos en la ventana *Hierarchy* un objeto *Canvas* por defecto vacio. En sus propiedades buscamos *Render Mode* y seleccionamos *Screen Space-Camera*. Aparecerá la propiedad *render Camera* a la cual asociamos a nuestra cámara de la escena. Esto nos permitirá enfocar automáticamente la cámara seleccionada al objeto *Canvas*

Si añadimos un elemento UI al canvas, por ejemplo un objeto tipo *Text-TextMeshPro*, la vista de la cámara deberá ser similar a la Figura 6.1.

Figura 6.1: interfaz de usuario que contiene una sola etiqueta sobre un fondo de color básico.

A partir de aquí podemos empezar a crear la interfaz a nuestro gusto con los distintos elementos disponibles:

- *Image y RawImage*. La imagen es solo para mostrar Sprites. *RawImage* es para mostrar cualquier tipo de textura.
- *Text*. para mostrar un *Label*.
- *Panel*. Es un objeto rectangular que contiene los componentes *Sprite Renderer* e *Image*

Además, podemos utilizar los típicos componentes de interfaz: *Toggle*, *Slider*, *Button*, *Scroll*, *Scrollbar*, *Dropdown* e *InputField*.

Ejercicio

Junto con el *Label* recién creado, vamos a añadir a nuestro *Canvas* dos botones. Cambiaremos el texto del *Label* a **MENÚ JUEGO** y los botones tendrán el texto **JUEGO** y **SALIR**. Para modificar el texto asociado a un botón debemos modificar el objeto *Text-TextMeshPro*, que está asociado al botón por defecto.

6.4 Asociar funciones a elementos de la interfaz

Cuando hemos añadido botones a nuestro menú normalmente querremos que hagan algo cuando son presionados. Para añadir funcionalidad a estos elementos debemos en primer lugar saber qué queremos que hagan. En nuestro ejemplo el botón **JUEGO** hará que cambiemos de escena a la que tenemos nuestro juego de balones. **SALIR** evidentemente hará que abandonemos la aplicación.

Estas dos funciones son muy simples pero a la vez muy útiles de cara a aplicaciones más complejas. Lo primero es crear un nuevo objeto vacío en la escena que va a ser nuestro *Manager* y le vamos a asociar un nuevo script. Empecemos por cerrar la aplicación, añadiendo al script una nueva función *Exit*.

```
1 public void Exit()
2 {
3   Application.Quit();
4 }
```

La función es bastante simple pero tenemos que aprender a asociarla a un botón. Buscamos en la ventana *Inspector* del botón que elijamos la propiedad *OnClick* que es un acceso directo al evento.

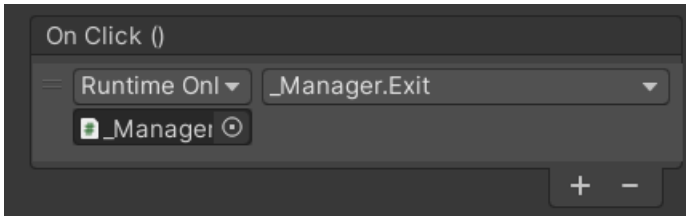


Figura 6.2: Elemento controlador del evento *OnClick* de un objeto *Button* al que se asocia la función para cerrar la aplicación.

Asociando **Manager** a esta propiedad nos permite acceder a las funciones públicas relacionadas con este objeto. Solo hay que seleccionar la función *Exit* para que sea la que se ejecute.

De igual forma vamos a implementar la funcionalidad de cambiar de escena.

```
1 public void ChangeScene(string sceneToChangeTo)
2 {
3   SceneManager.LoadScene(sceneToChangeTo);
4 }
```

Esta función requiere que incluyamos el paquete *Unity.SceneManagement* al inicio del script.

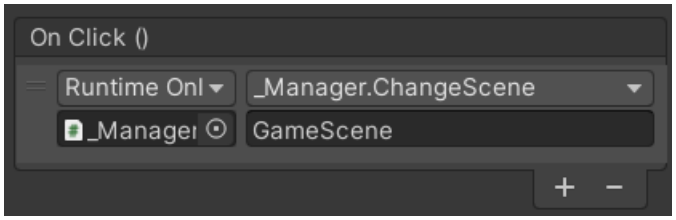


Figura 6.3: Elemento controlador del evento *OnClick* de un objeto *Button* al que se asocia la función para el cambio de escena.

En este caso deberemos pasar por parámetros el nombre de la escena a la que queremos cambiar.

6.5 Personalización

El menú que hemos creado es funcional pero bastante simple. Normalmente, querremos que la interfaz sea adecuada a la aplicación que estamos manejando mediante un estilo propio o mediante imágenes relacionadas.

Para personalizar nuestros *sprites* (recordar que los sprites en 2D son los objetos, por ejemplo los botones) podemos variar entre las opciones por defecto del proyecto navegando en la propiedad *Source Image* del componente *Image* asociado al botón y ajustando el tamaño para adecuarlo a lo que necesitamos. Siempre podemos utilizar recursos adicionales como son los que se ofrecen en `https://assetstore.unity.com/`.

Para añadir una imagen de fondo es todavía más sencillo. Importamos la imagen que queramos a nuestro proyecto arrastrándola a nuestra carpeta *Assets*. Antes de poder utilizarla hay que configurarla en la ventana *Inspector* indicando en la propiedad *Texture Type* que va a ser un *Sprite(2D and UI)* y el modo va a ser *Single*. Hay que tener cuidado que la imagen tenga un tamaño adecuado. Para evitar problemas nuestra imagen la escalaremos (antes de importarla) al tamaño 1920×1080 que corresponde con la resolución 4K.

Una vez configurada aparecerá un símbolo de reproducción en el *thumbnail* asociado a la imagen.

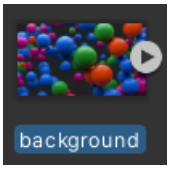


Figura 6.4: Ejemplo de imagen configurada como *thumbnail*.

Arrastrando este elemento a nuestro objeto *Canvas* crearemos un objeto con un componente *Sprite Renderer* con la imagen asociada. Mediante los controles de posición

y escala, ajustaremos la imagen en la escena al tamaño del *Canvas*. Aun así, es posible que no veamos nuestra imagen como fondo del menú cuando ejecutemos la escena. Eso se debe a que la cámara solo muestra la capa UI y nuestro objeto está por defecto en la capa *Default*. Cambiando la capa donde se encuentra el objeto veremos como cambia nuestro menú.

Ejercicio

Buscar en `https://assetstore.unity.com/` sprites para los botones y una imagen para colocar de fondo. En la tienda no podremos descargarlo, en cambio, podremos asociarlo a nuestro proyecto mediante nuestra cuenta personal. Al hacerlo nos dará la opción de abrir el Editor de Unity, para a continuación, abrir el *Package Manager* desde donde podremos descargarlo e importarlo a nuestro proyecto.
También podemos conseguir *Widgets* para nuestro juego. Los *Widgets* son microaplicaciones ya definidas que se despliegan en la escena como objetos. Estos objetos tienen propiedades y funciones particulares. Algunos ejemplos de *Widgets* son: un reloj analógico, un selector de colores, un ecualizador de audio, etc.

6.6 Interacción con el Juego

Crear un menú nos ha servido para aprender las bases de como crear elementos UI en nuestro juego. Otra función importante es aprender como mostrar información en tiempo real de lo que ocurre en el juego. Para ejemplificarlo, vamos a implementar un indicador que nos muestre el valor de la rotación de la estructura en cada momento.

En nuestra escena principal vamos a repetir el proceso de añadir un objeto *Canvas* con la diferencia que esta vez, la propiedad *Culling Mask* de la cámara tendrá el valor *Everything*. De esta manera la interfaz se verá sobre la reproducción de nuestro juego.

Si recordamos la práctica anterior, aprendimos a crear un objeto *AudioManager* que controlaba todos los aspectos del audio del juego. Vamos a crear un objeto *GameManager* de la misma manera para controlar los aspectos del juego. Si no nos acordamos, basta con duplicar el objeto *AudioManager* y eliminar los elementos relacionados con el audio en su script asociado.

Ejercicio

Como ejercicio final se trata de incluir en *GameManager* todas las funciones repartidas por distintos scripts y componer un interfaz de juego con los siguientes elementos.

- Un botón que cada vez que se presione, cree un balón en el juego.
- Un botón que inicie la rotación de la estructura de forma aleatoria.
- Un botón que pare la rotación de la estructura y reinicie la posición de la estructura.
- Un botón que aumente la velocidad de rotación hasta un valor máximo de 5.
- Un botón que reduzca la velocidad de rotación hasta un valor mínimo de 0.
- Incluir un botón que bien cierre la aplicación o bien vuelva al menú principal.
- Incluir un *Label* que indique la velocidad actual de rotación de la estructura.
- Incluir un *Label* que muestre los valores de rotación de la estructura en cada momento.

Para mostrar un valor en un objeto *Label* debemos usar el paquete *TMPro* en nuestro script, para poder utilizar las propiedades de los objetos *Text-TextMeshPro*. A continuación vemos el código necesario para resolver los dos últimos elementos del ejercicio.

```
1  using TMPro;
2
3  public class GameManager : MonoBehaviour
4  {
5    public TextMeshProUGUI speedText;
6    public TextMeshProUGUI rotationText;
7
8   void Update()
9   {
10    speedText.text = "Speed: " + speed.ToString();
11    rotationText.text = "Rotation: " + Mathf.Round(rotationChange.x * 100.0f) / 100.0f + " - " +
          Mathf.Round(rotationChange.y * 100.0f) / 100.0f + " - " + Mathf.Round(rotationChange.z *
          100.0f) / 100.0f;
12  }
13 }
```

6.7 Conclusiones

Los elementos de interfaz nos permiten incluir funcionalidades para modificar el comportamiento del juego durante su ejecución, así como, visualizar la información generada. También hemos aprendido a utilizar los componentes para movernos a través de distintas escenas de ser necesario y a personalizar nuestra interfaz.

Con esta práctica ya conocemos todos los elementos necesarios para realizar nuestro proyecto para la asignatura.