UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INFORMÁTICA

GRADO EN INGENIERÍA INFORMÁTICA

Realidad Virtual y Aumentada

Programación e interacción. Lenguaje C# en Unity 3D, control de elementos multimedia y 3D.

Autor: Javier Córdoba Romero y Juan José Corroto Martin

Profesor: Javier Alonso Albusac Jiménez

Diciembre, 2019

Índice

1. C#

C# nació en el 2000 por parte de Microsoft enmarcado dentro del .NET Framework, un framework que permite reutilizar código escrito para otros lenguajes que también compilen para .NET.

.NET Framework se fue expandiendo con los años hasta llegar al 2016 con el desarrollo de .NET Core, una alternativa de código abierto y usable tanto en Windows, Linux y macOS, haciendo que los programas escritos para este framework pudiesen ser ejecutados en la mayoría de sistemas operativos.

C# es un lenguaje de programación orientado a objetos con tipado fuerte y con recolector de basura ya integrado. La sintaxis de C# está influenciada por la de C, C++ y Java.

1.1. Tipos de Variable

En C# existen 4 categorías de variables:

- Valor
- Referencia
- Parámetros genéricos
- Punteros

Tipos de variables por valor Los `structs` y la mayoría de tipos incluidos por defecto pertenecen a esta categoría, exceptuando al tipo `string`, lo mismo ocurre en Java.

Un ejemplo de tipo pasado por valor está en el Listado 1, donde se ve que aunque el valor de `x` sea asignado a `y` y posteriormente el valor de `x` cambie, el valor de `y` no lo hace.

Tipos de variables por referencia Mientras que los tipos de variables por valor están compuestos por sólo una parte, el valor, los tipos por referencia están compuestos por dos partes: un objeto y la referencia a ese objeto. El valor que se almacena en la variable es la referencia a ese objeto, el objeto se creará y almacenará independientemente de la referencia cuando se utilice el operator `new`.

Un ejemplo de tipo de variable por referencia es la clase `Point2D`, que almacena el valor de

```
1  public static void Main()
2  {
3          int x = 4;
4          int y = x;            // Se copia el valor
5
6          Console.WriteLine(x); // 4
7          Console.WriteLine(y); // 4
8
9          y = 7;
10
11         Console.WriteLine(x); // 4
12         Console.WriteLine(y); // 7
13 }
```

Listado 1: Ejemplo de tipo de variable por valor

un punto en el espacio 2D, el Listado 2 ilustra un ejemplo de el comportamiento de un tipo de variable por referencia.

```
1  public class Point2D
2  {
3          public int x;
4          public int y;
5  }
6
7  public static void Main()
8  {
9          Point2D p1 = new Point2D();
10         p1.x = 4;
11         Point2D p2 = p1;          // Se copia la referencia
12
13         Console.WriteLine(p1.x); // 4
14         Console.WriteLine(p2.y); // 4
15
16         p1.x = 7;
17
18         Console.WriteLine(x);    // 7
19         Console.WriteLine(y);    -// 7
20 }
```

Listado 2: Ejemplo de tipo de variable por referencia

1.1.1. Arrays

En C# los arrays están completamente soportados y gestionados por el lenguaje, sin que el programador tenga que preocuparse por la gestión de la memoria o de cómo se guardan en memoria.

Los arrays siempre se guardan en un bloque contiguo de memoria haciendo que los accesos a

sucesivos elementos de un array sean muy eficientes.

Los elementos de un array siempre se inicializarán con un valor por defecto, este valor depende del tipo de variable del array.

Si el tipo de variable del array es por valor, el valor por defecto es cero en el caso de los tipos numéricos, `false` para el tipo booleano, en el caso de los structs, se inicializan con el valor predeterminado de cada elemento, como se puede ver en el Listado 3.

```
1  public static void Main()
2  {
3          int[] x = new int[5];     // Se crea un array de 5 enteros
4          x[1] = 2;                                  // Se asigna el valor 2
               al segundo elemento del array
5
6          Console.WriteLine(x[0]);     // 0
7          Console.WriteLine(x[1]);     // 2
8          Console.WriteLine(x[2]);     // 0
9          Console.WriteLine(x[3]);     // 0
10         Console.WriteLine(x[4]);     // 0
11 }
```

Listado 3: Ejemplo de array con tipo de variable por valor

Sin embargo, si el tipo de variable del array es por referencia, el valor por defecto es `null` ya que , como se ha comentado anteriormente, lo que se almacena en la variable es la referencia al objeto.

La consecuencia de esto es que un array de tipo de variable por referencia necesita ser inicializado después de declarar el array, como se puede observar en el Listado 4.

1.2. Clases

En C#, una clase es un tipo de variable por referencia, para declarar una clase basta con escribir la palabra clave `class` seguido por el nombre de la clase y finalizando por los corchetes de apertura y cierre.

Una clase puede contener uno o varios campos, estos campos son variables que pertenecen a la clase, estos campos pueden ser inicializados incluso antes de que el constructor sea ejecutado, un ejemplo de clase con un campo inicializado está en el Listado 5.

Una clase también puede contener métodos, un método es una función que recibe cero o más parámetros y que puede devolver un parámetro. Un método se define de la siguiente manera:

```
public class Point2D
{
        public int x;
        public int y;
}

public static void Main()
{
        Point2D[] x = new Point2D[5];    // Se crea un array de 5 Point2D
        x[1] = new Point2D();                    // Se asigna el valor 2
            al segundo elemento del array

        Console.WriteLine(x[0] == null);    // true
        Console.WriteLine(x[1] == null);    // false
        Console.WriteLine(x[2] == null);    // true
        Console.WriteLine(x[3] == null);    // true
        Console.WriteLine(x[4] == null);    // true
}
```

Listado 4: Ejemplo de array con tipo de variable por referencia

```
public class Point2D
{
        public int x = 10;  // El valor inicial es 10
        public int y;       // El valor inicial es el de por defecto, 0
}
```

Listado 5: Ejemplo de clase con un campo inicializado

```
tipo-de-retorno nombre-del-metodo(lista-de-parametros)
{
        lista-de-sentencias
}
```

Listado 6: Estructura de un método

La definición de un método en concreto se llama *firma*, pueden cooexistir dos métodos con el mismo nombre mientras no tengan la misma firma, a esto se le llama *sobrecarga* de métodos.

Dentro de cada clase hay un método especial, el constructor, este método es llamado justo después de crear un objeto de una clase con el operator `new` y una vez los campos de la clase se han inicializado, el constructor no tiene tipo de retorno y acepta tantos parámetros como sea necesario.

Un ejemplo de clase con campos, métodos sobrecargados y un constructor está en el Listado 7

```
1  public class Point2D
2  {
3          public int x = 10;       // El valor inicial es 10
4          public int y;            // El valor inicial es el de por defecto, 0
5
6          public Point2D(int x, int y)    // Constructor de la clase
7          {
8                  this.x = x;                          // Asignar el
                      parametro x al campo x de la clase
9                  this.y = y;                          // Asignar el
                      parametro y al campo y de la clase
10         }
11
12         // Metodo que devuelve un Point2D y que acepta un Point2D
13         public Point2D calcularVector(Point2D p)
14         {
15                 int vx = p.x - this.x;
16                 int vy = p.y - this.y;
17
18                 return new Point2D(vx, vy); // Devolver un nuevo Point2D
19         }
20
21         // Metodo sobrecargado que devuelve un Point2D y que acepta dos
              enteros
22         public Point2D calcularVector(int px, int py)
23         {
24                 int vx = px - this.x;
25                 int vy = py - this.y;
26
27                 return new Point2D(vx, vy); // Devolver un nuevo Point2D
28         }
29 }
```

Listado 7: Ejemplo de clase con campos, sobrecarga de métodos y constructor

1.2.1. Herencia

Una clase puede heredar de otra clase para ampliar o personalizar la clase base. La ventaja de la herencia es la capacidad de reutilizar la funcionalidad de la clase base en lugar de construirla

desde cero. Una clase sólo puede heredar de una clase, pero puede ser heredada por muchas clases, lo que forma una jerarquía de clases.

Las referencias a una clase de una jerarquía es polimórfica. Esto significa que una variable de tipo X puede referirse a un objeto que heredó de X, como se puede ver en el Listado 8, la variable `Animal` en el método `main` es polimórfica porque puede almacenar tanto una referencia a la clase `Gato` como a la clase `Pajaro`.

Respecto al uso de la herencia en Unity, la clase base en Unity es `MonoBehaviour`, es decir, todas las clases base deben heredar de ella. Como nota, en Unity se favorece la composición frente a la herencia, en parte por el sistema de componentes de Unity. Lo que significa que la herencia no es tan usada como en otros entornos de desarrollo de videojuegos.

1.3. Modificadores

Los modificadores son palabras clave que acompañan a variables, clases y parámetros que modifican la forma de acceder a ellos u otras características como la forma en la que los métodos se comportan una vez son heredados, hay tres tipos principales de modificadores:

Modificadores de acceso , que son los que controlan cómo y quién puede acceder a las variables, estos modificadores son:

- `public`: No hay restricciones al acceso.
- `protected`: El acceso está limitado a la clase contenedora y a los tipos derivados de la clase contenedora.
- `internal`: El acceso está limitado al ensamblado actual, este es el modificador por defecto si no se especifica ninguno.
- `protected internal`: No hay restricciones al acceso.
- `private`: No hay restricciones al acceso.

El concepto de *ensamblado* se asemeja al concepto de unidades de compilación de C, sin embargo, en C#, los ensamblados también pueden contener recursos como iconos, imágenes, texto traducido...

Modificadores de clases , que son los que controlan las propiedades de una clase, estos modificadores son:

- `abstract`: Indica que la clase no tiene una implementación completa y por lo tanto no

```
1  public class Animal
2  {
3          public int numeroDePatas;
4
5          public Animal(int nPatas)
6          {
7                  this.numeroDePatas = nPatas;
8          }
9  }
10
11 public class Gato : Animal
12 {
13         public string color;
14
15         public Gato(int nPatas, string color) :
16                 base(nPatas) // Llamar al constructor de la clase base
17         {
18                 this.color = color;
19         }
20 }
21
22 public class Pajaro : Animal
23 {
24         public string color;
25         public int numeroDeAlas;
26
27         public Pajaro(int nPatas, string color, int nAlas) :
28                 base(nPatas) // Llamar al constructor de la clase base
29         {
30                 this.color = color;
31                 this.numeroDeAlas = nAlas;
32         }
33 }
34
35 public static void Main()
36 {
37         Animal gato = new Gato(4, "blanco");        // Ejemplo de
               polimorfismo
38         Animal pajaro = new Pajaro(2, "negro", 2);  // Ejemplo de
               polimorfismo
39
40         Console.WriteLine("El gato tiene " + gato.numeroDePatas + "
               patas");
41         Console.WriteLine("El pajaro tiene " + ((Pajaro)
               pajaro).numeroDeAlas + " alas");
42         Console.WriteLine("El gato es de color " + ((Gato) gato).color);
43         Console.WriteLine("El pajaro es de color " + ((Pajaro)
               pajaro).color);
44 }
```

Listado 8: Ejemplo de clase con campos, sobrecarga de métodos y constructor

se podrán crear instancias de esa clase. La implementación debe de ser completada por clases derivadas no abstractas.

- `sealed`: Impide que la clase sea heredada por otras clases.
- `partial`: La definición de la clase está dividida en dos o más archivos, es útil para distinguir la parte de inicialización de la UI con la de lógica.

Modificadores de métodos , que son los que controlan cómo se heredan los métodos de una clase a otra, estos modificadores son:

- `abstract`: Indica que el método no tiene implementación y por lo tanto no se podrán crear instancias de esa clase. La implementación debe de ser completada por clases derivadas no abstractas.
- `sealed`: Impide que el método sea sobrescrito por otras clases derivadas.
- `virtual`: Indica que el método modificado podrá ser sobrescrito en una clase derivada, ya que por defecto, esto no está permitido.
- `override`: Permite modificar la definición de un método en una clase derivada, siempre que esto esté permitido.

1.4. Delegados – Programación basada en eventos

C# también tiene facilidades para una programación basada en eventos, este elemento se llama `delegado`.

Un delegado es un objeto que contiene una referencia a un método al que llamará cuando se le indique.

Al definir un delegado también se definirá que tipo de métodos podrá llamar, un delegado se define de la siguiente manera:

```
1 delegate tipo-de-retorno nombre-del-delegado(lista-de-parametros);
```

Listado 9: Estructura de un delegado

Para añadir un método a un delegado bastará con usar el operador de asignación (=).

Los delegados de C# también son multicast, es decir que un delegado no sólo almacena una única referencia a un método, si no que puede referenciar a múltiples métodos.

Un ejemplo del uso de delegados está en el Listado 10

```
1  delegate void Saludos(string nombre);  // Declarar un delegado que no
       devuelve nada y que acepta un parametro string
2
3  static void saludarEspanol(string nombre)
4  {
5          Console.WriteLine("Hola " + nombre + "!");
6  }
7
8  public static void Main()
9  {
10         Saludos sal = saludarEspanol;  // Crear un delegado y asignarle el
               metodo saludarEspanol
11         sal("Javier");                 // Llamar a los metodos asignados
               al delegado
12 }
```

Listado 10: Ejemplo de uso de delegados unicast

1.4.1. Eventos en Unity

Todos los objetos en un mundo de Unity contienen uno o varios componentes, esto aporta mucha flexibilidad ya que se puede reutilizar el manejador de colisiones de un personaje en otro sin tener que modificar el código.

Los componentes son tan diversos como colisionadores, *rigidbodies*, que permiten que los colisiones interactúen con el motor de físicas, un animador o un sistema de partículas.

Dentro de estos componentes, los scripts que se programan en C# también son componentes lo que aumenta la flexibilidad del código escrito.

Dentro de estos scripts tenemos *eventos* en forma de métodos que Unity llamará cuando suceda algún evento, estos eventos son:

- `Awake()`: Este evento es llamado una sola vez cuando el script es cargado, **aunque el script no esté activado**
- `Start()`: Este evento podrá ser llamado múltiples veces cuando el script se active.
- `Start()`: Este evento es llamado una sola vez cuando el script es cargado y **activado**
- `FixedUpdate()`: Llamado antes de que el motor de físicas vuelva a recalcular las físicas del mundo, llamado con la frecuencia del sistema de físicas e independientemente del *frame-rate*, útil para aplicar fuerzas para que están sean lo más precisas posible.
- `Update()`: Llamado cada *frame*, útil para implementar lógica del juego.
- `LateUpdate()`: Llamado cada *frame*, después de que todas las funciones `Update` sean ejecutadas, útil cuando se debe realizar alguna operación dependiente en el movi-

miento realizado en `Update`.

- `OnGUI()`: Usado para implementar la interfaz de usuario, puede ser llamado múltiples veces en un solo frame si hay varios eventos que procesar.
- `OnApplicationPause()`: Llamado cada vez que el juego se pone en pausa.

Otro problema resuelto por Unity es el siguiente supuesto.

A la hora de modificar los parámetros de los *scripts* consumiría mucho tiempo ya que habría que abrir el *script* en el editor de código y buscar el parámetro a modificar de entre todas las variables declaradas, Unity, para hacer más sencillo este proceso, hace que todas las variables públicas de un *script* aparezcan en el inspector de componentes, lo que simplifica este proceso y hace que pueda ser realizado por gente sin conocimientos técnicos.

2. Uso de C# en Unity

La primera versión del motor de juegos Unity fue lanzada en el 2005, desde entonces mucho ha cambiado en el motor, pasando por el motor de scripting, nuevas tecnologías gráficas y el modelo de licencias.

El lenguaje de scripting de Unity pasó por varias etapas, la primera de ellas fue *Boo*, un lenguaje orientado a objetos usando el *Common Language Runtime* (CLR) de .NET Framework, tecnología similar a la usada por *Java* en su máquina virtual. La siguiente etapa pasó por usar una variante de *Javascript* como lenguaje de scripting y, por último, se pasó a usar C# como lenguaje de scripting con *Visual Studio* como uno de los IDEs soportados.

C# es un lenguaje de programación orientado a objetos y ejecutado sobre el *Common Language Runtime*, usando principalmente con tipos estáticos aunque con soporte para tipos dinámicos, también se puede enfocar a una programación basada en eventos gracias a sus *delegados*.

Ha conseguido obtener la condición de estándar ISO, su última revisión fue en 2018, con referencia ISO 23270:2018[1]

Actualmente Unity soporta más de 20 plataformas[2], entre las que más destacan podemos encontrar: Windows, Linux, Mac, Playstation 4, Xbox One, Nintendo 3Ds, Oculus Rift, Android, iOS, WebGL, ARKit y ARCore.

C# en Unity también permite interaccionar con los diferentes componentes del motor, como por

[1] https://www.iso.org/standard/75178.html
[2] https://unity3d.com/es/unity/features/multiplatform

ejemplo *Mecanim*, su sistema de animación, su sistema de físicas o su sistema de componentes, lo que proporciona un control total sobre el motor y no sólo tareas de scripting in-game.

2.1. Manejo de objetos de Unity desde C#

En esta sección se va a tratar cómo manipular objetos del mundo virtual desde un *script* en *Unity*. Como ya se ha dicho, los *scripts* son componentes de los objetos. Desde cualquier *script* podemos manipular cualquier otro objeto del mundo, o el mismo objeto. Esto último es especialmente fácil si queremos aplicar una transformación, pues tenemos acceso al componente *Transform* como variable de clase. En [11] podemos ver como se puede acceder al componente *transform* del objeto e invocar sus funciones para aplicarle una traslación. La variable *transform* de ese listado no es necesario que se defina en ningún sitio del *script*, sino que se tiene acceso a ella directamente. Esto es porque el componente *Transform* es inherente a todos los objetos que están en la escena, puesto que todos los objetos deben de tener una transformación para saber dónde hay que dibujarlos. El resto de componentes (que deben ser añadidos manualmente), pueden ser accedidos simplemente con la función *GetComponent<Tipo>()*. De esta forma podemos fácilmente aplicar transformaciones o fuerzas (en el caso de rigidBody) sobre el mismo objeto.

```
1 Vector3 direction = new Vector3(mainCamera.transform.forward.x, 0,
     mainCamera.transform.forward.z).normalized * speed * Time.deltaTime;
2 Quaternion rotation = Quaternion.Euler(new Vector3(0,
     -transform.rotation.eulerAngles.y, 0));
3 transform.Translate(rotation * direction);
```

Listado 11: Acceso al componente transform para modificar el propio objeto

En el caso de querer modificar otro objeto, por ejemplo, crear un enemigo a cierta distancia de otro, simplemente tenemos que encontrar la referencia al objeto. Esto se puede hacer muy fácilmente haciendo una variable pública y asignándola desde el editor. Sin embargo, si queremos acceder al objeto de forma dinámica o únicamente usando *C#*, se puede hacer de varias formas, todas usando funciones estáticas de la clase *GameObject*:

1. Por nombre: usando la función *GameObject.Find()*. Se puede buscar de forma directa un objeto en la escena virtual mediante su nombre, simplemente es necesario conocerlo de antemano. Este último matiz puede que sea la mayor inconveniencia, pues es probable que no sepamos de antemano el nombre del objeto, sobre todo si se ha generado dinámicamente. Esta función tratará los caracteres ”/” no como parte del nombre, sino como

parte de una jerarquía de objetos. La documentación oficial de *Unity* [3] [1] desaconseja usar esta función cada frame.

2. Por tipo: usando la función *GameObject.FindObjectOfType(Type type)*. En este caso, buscamos por tipo de objeto, teniendo un ejemplo en el listado 12. Esta función retorna el **primer** objeto cargado del tipo especificado, o *null* si no existe ninguno de dicho tipo. Para conseguir un iterador de todos los objetos de ese tipo, también existe la función *GameObject.FindObjectsOfType(Type type)*. La documentación oficial desaconseja utilizar estas funciones por ser más lentas que el resto.

3. Por etiqueta o *tag*: usando la función *GameObject.FindGameObjectWithTag(String tag)*. Esta función, al igual que la anterior, devuelve el **primer** objeto cargado con dicha tag, o *null* si no encuentra ninguno. Además, lanzará una excepción si la tag no existe. Esta función es muy fácil de usar, pues *Unity* tiene un sistema nativo de tags para los objetos muy simple de usar, a parte de tener tags predefinidas. El hecho de que sean simples cadenas de caracteres también lo hace más sencillo de utilizar.

```
1  using UnityEngine;
2  using System.Collections;
3
4  // Search for any object of Type GUITexture,
5  // if found print its name, else print a message
6  // that says that it was not found.
7  public class ExampleClass : MonoBehaviour
8  {
9      void Start()
10     {
11         GUITexture texture =
               (GUITexture)FindObjectOfType(typeof(GUITexture));
12         if (texture)
13             Debug.Log("GUITexture object found: " + texture.name);
14         else
15             Debug.Log("No GUITexture object could be found");
16     }
17 }
```

Listado 12: Búsqueda de objetos por tipo

Una vez hemos conseguido la referencia al objeto, podemos modificar cualquiera de sus cualidades accediendo a sus componentes, de la misma forma que se explicó anteriormente con el objeto al que pertenece el *script*.

De forma similar se pueden controlar contenidos multimedia, como sonidos o vídeos. En Unity estos contenidos también son componentes. Se pueden modificar fácilmente usando el mismo

[3] https://docs.unity3d.com/ScriptReference/GameObject.html

proceso descrito anteriormente. Simplemente hay que encontrar la referencia al componente de sonido o vídeo y utilizar las funciones disponibles por parte de dicho componente.

Todo lo descrito anteriormente es igual de válido para desarrollar en realidad virtual. En *Unity* no es necesario modificar ningún punto del desarrollo de la escena, por lo que la forma de modificar objetos 3D o multimedia es la misma. La única diferencia es el componente del controlador del jugador, que tendrá que tener la lógica asociada al control con el *hardware* de realidad virtual.

3. Ejemplos

Para terminar y concretar el trabajo, hemos realizado varios *scripts* sencillos que demuestran de forma práctica cómo modificar elementos del juego mediante el lenguaje *C#*. Se ha usado el escenario de ejemplo que incluye el *plugin* de realidad virtual de Google (el cuál usamos para poder ejecutar el juego en el móvil) 1. Dentro de este escenario se han desarrollado 3 *scripts*:

1. Movimiento automático del jugador, para poder desplazarse por el escenario sin pulsar botones.
2. Aplicación de fuerzas a un objeto con físicas.
3. Reproducción de un vídeo interactuando con un objeto del escenario.

La parte más interesante del primer *script* se puede ver en el listado 13. Cada frame, el *script* calcula si el jugador debería moverse o no. Hemos decidido controlarlo mediante el ángulo de visión. Si el jugador baja la vista por debajo de un cierto ángulo, empezará a moverse hasta que suba la vista por encima de otro ángulo. Todos los parámetros pueden ser elegidos desde el editor.

```
1 if (isWalking)
2         {
3             Vector3 direction = new
                  Vector3(mainCamera.transform.forward.x, 0,
                  mainCamera.transform.forward.z).normalized * speed *
                  Time.deltaTime;
4             Quaternion rotation = Quaternion.Euler(new Vector3(0,
                  -transform.rotation.eulerAngles.y, 0));
5             transform.Translate(rotation * direction);
6         }
```

Listado 13: Translación y rotación del jugador en el movimiento automático

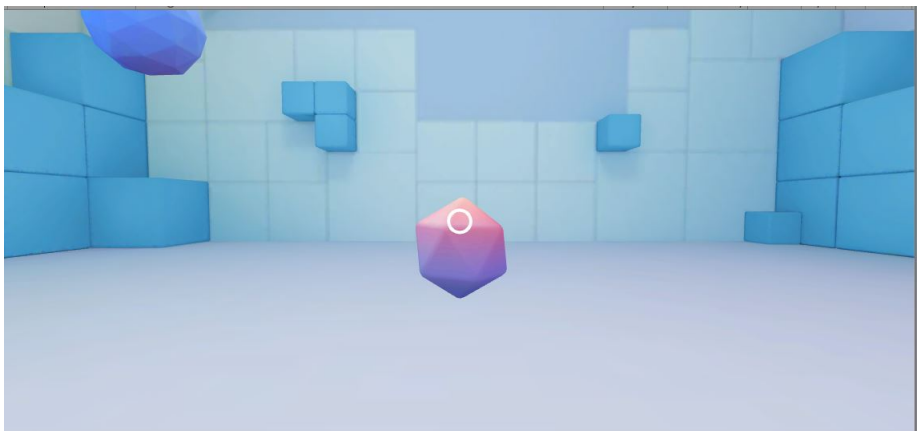

Figura 1: Escena para los ejemplos.

El segundo *script* [14] consiste en una función que se ejecutará cuando el jugador interactúe con la figura 3D que se puede ver en la figura 1. Para ello, encontramos el componente *RigidBody*, que es el que nos dejará aplicar una fuerza en los 3 ejes, iterando sobre todos los componentes del objeto. También buscamos la referencia al jugador mediante la función discutida anteriormente *GameObject.FindGameObjectWithTag("Player")*, usando como tag "Player". Obtenemos el componente *RigidBody* del jugador para poder calcular el vector entre los dos objetos, y aplicamos una fuerza en esa dirección, aunque en el eje Y (el que apunta hacia .ªrriba") se cambia para que la fuerza sea siempre en sentido ascendente. Este ejemplo muestra muy bien cómo se pueden obtener componentes y otros objetos mediante el uso de un par de funciones.

El último *script* consiste en la reproducción y pausa de un vídeo interactuando con otro objeto dentro de la escena. En la figura 2 se pueden ver el objeto con el que hay que interactuar y el plano donde se puede visualizar el vídeo. Este último tiene un componente *VideoPlayer* que es el que se encarga de reproducir el vídeo como si fuera una textura dinámica del plano. En el listado 15 se puede ver cómo el proceso es prácticamente el mismo: Encontramos el objeto dentro de la escena y obtenemos el componente deseado. En este ejemplo aparece cómo encontrar un componente en concreto sin necesidad de usar un iterador, símplemente estableciendo el tipo específico de componente en la función *GetComponentInChildren()*. Sin embargo, esto sólo funcionará con los componentes de tipos ya definidos en *Unity* (no con otros *scripts*).

4. Conclusión

La conclusión de este estudio es que Unity es muy flexible tanto en la interacción como en la inclusión de elementos multimedia.

```
// Iterate over all components to get RigidBody
Component[] components = GetComponentsInChildren<Component>();
foreach (Component component in components)
{
        if(component is Rigidbody)
        {
                Rigidbody rb = (Rigidbody)component;

                // Calculate vector between player and object
                GameObject player =
                    GameObject.FindGameObjectWithTag("Player");

                Component[] player_components =
                    player.GetComponentsInChildren<Component>();

                Rigidbody player_rb = null;

                foreach (Component player_component in player_components)
                {
                        if (player_component is Rigidbody)
                        {
                                player_rb = (Rigidbody)player_component;
                        }
                }

                Vector3 dir =  rb.position - player_rb.position;\\
                Vector3 unit = dir / dir.magnitude;
                unit.y = 0.5f;
                Vector3 force = unit * 250;
                rb.AddForceAtPosition(force, rb.position);
        }
}
```

Listado 14: Aplicación de una fuerza a un objeto con físicas

```
GameObject video = GameObject.FindGameObjectWithTag("Video");

var videoplayer =
    video.GetComponentInChildren<UnityEngine.Video.VideoPlayer>();

if (!videoplaying)
{
        videoplayer.Play();
        videoplaying = true;
}
else
{
        videoplayer.Pause();
        videoplaying = false;
}
```

Listado 15: Inicio y pausa del vídeo desde el objeto con el que se interactúa

Figura 2: Escena para los ejemplos, segunda perspectiva.

Algunas de las facilidades que Unity provee para la interacción es la inclusión por defecto de los eventos más usados en el desarrollo de juegos.

Respecto a las facilidades con los elementos multimedia destaca el amplio soporte a formatos tanto de audio como de video y la manera unificada de controlar estos aspectos, sin que sea necesario tener en cuenta el formato del elemento.

Referencias

[1] Unity. *Unity Scripting Reference*. URL: https://docs.unity3d.com/ScriptReference/.

[2] Joseph Albahari y Ben Albahari. *C# 7.0 in a Nutshell*. O' Reilly, 2017.

[3] Microsoft. *Documentación C#*. URL: https://docs.microsoft.com/es-es/dotnet/csharp/.

[0] Unity. *Unity Events Reference*. URL: https://docs.unity3d.com/Manual/ExecutionOrder.html.