Universidad Central de Venezuela
Facultad de Ciencias
Escuela de Computación

Laboratorio 2

Videojuegos, Programación, Diseño

Cámaras Virtuales

1. Pre-Laboratorio

- Investigar:
  1. Videojuegos en primera persona.
  2. Videojuegos en tercera persona.
  3. Side-scrollers.
  4. Perspectiva.
  5. Proyección Ortogonal.
  6. Géneros de videojuegos.
- Investigar según su framework, biblioteca o motor de juego escogido.
  1. Implementación de cámaras en el juego.
  2. Manejo de escenas.
  3. Scripting o programación de la lógica de juego.

2. Definición

En los videojuegos se utilizan cámaras virtuales para mostrar un mundo 3D o 2D en algún dispositivo visual. Un sistema virtual de cámaras se encarga de controlar una o mas cámaras en un escenario de juego. Usualmente la cámara en muchos frameworks y bibliotecas para desarrollo de videojuegos no es mas que otro objeto en escena [10].

3. Conceptos

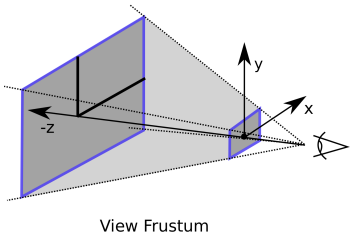


Figura 1: *Viewing Frustum* de una cámara virtual donde se muestra el far y near clipping plane [6].

Algunos conceptos necesarios para entender el uso de cámaras en la gran mayoría de los frameworks, bibliotecas y motores para el desarrollo de videojuegos [9].

3.1. Tipos de Proyección

Existen dos tipos proyección en la mayoría de los frameworks y bibliotecas para el desarrollo de videojuegos estos son proyección perspectiva y proyección ortogonal.

3.1.1. Perspectiva

Es la forma natural en que el ojo humano percibe una escena, en la proyección perspectiva los objetos distantes se ven mas pequeños que los objetos cercanos dando profundidad a distintos objetos en una escena. Este tipo de proyección es utilizada usualmente en juegos 3D.

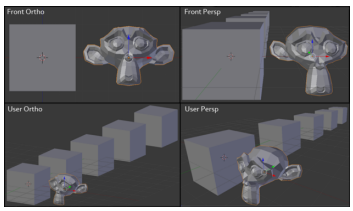


Figura 2: Proyección ortogonal y perspectiva.

3.1.2. Ortogonal

En una proyección perspectiva un objeto lejano es mas pequeño que un objeto cercano, en proyección ortogonal se ignora este efecto, eliminando la profundidad de escena. Este tipo de perspectiva es utilizada en muchos juegos 2D.

3.2. Field Of View (FoV) o Campo de Vista

Es el ancho del angulo de vista de la cámara, indica la extension de lo puede ver la cámara en cualquier momento, usualmente se mide en ángulos, este angulo puede ser el field of view vertical o el field of view horizontal dependiendo del framework, biblioteca o motor de juego usado (ver 3) [1].

3.3. Clipping Planes o Planos de Clipping

En computación se manejan términos discretos por lo tanto una cámara no puede ver hacia el infinito, para esto están el far y near clipping planes los cuales indican donde termina el renderizado de escena y donde empieza según la posición de la cámara respectivamente (ver 1 y 3).

3.3.1. Near Clipping Plane o Plano de Clipping Cercano

Es donde empieza a dibujarse los objetos de escena en display, los objetos antes de este punto son ignorados por el motor gráfico.

3.3.2. Far Clipping Plane o Plano de Clipping Lejano

Es donde termina de dibujarse los objetos de escena en display, los objetos mas lejanos a este punto son ignorados por el motor gráfico.

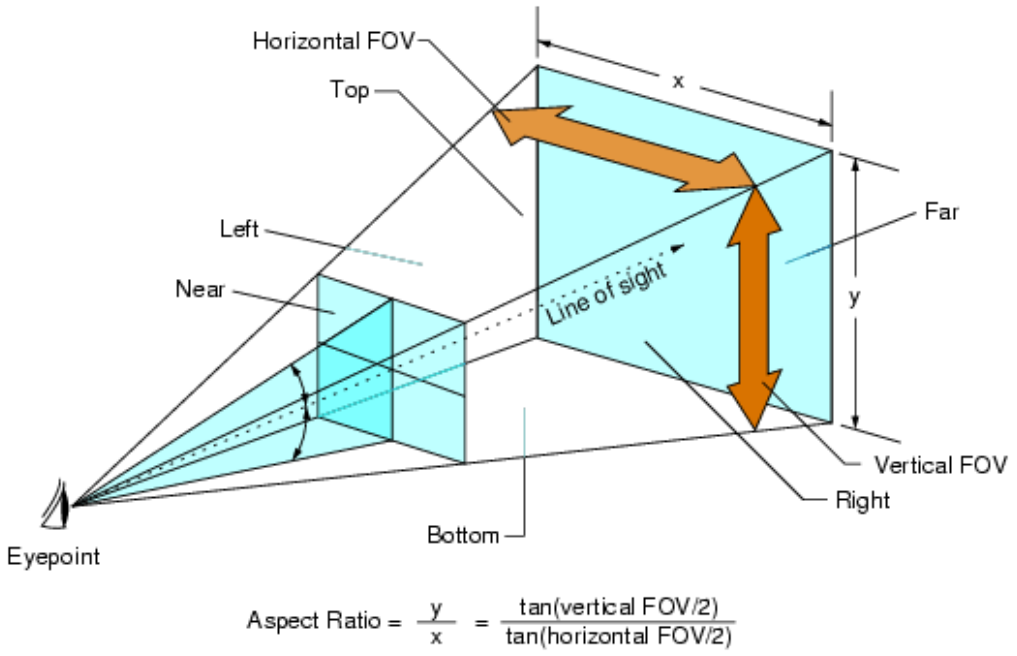


Figura 3: Parámetros de una Cámara Virtual

4. Tipos de Sistemas de Cámara en Videojuegos

Existen principalmente tres tipos de sistemas de cámara en videojuegos. En un sistema *fixed* la cámara no cambia sus parámetros originales, un sistema *tracking* sigue algún objeto en juego, y un sistema *interactive* la cámara es parcialmente autónoma y cambia sus parámetros según distintas situaciones. Para implementar distintos sistemas de cámara los desarrolladores de videojuegos usan técnicas como programación con restricciones o inteligencia artificial.

4.1. Estáticas (*Fixed*)

Figura 4: Ejemplo de cámaras estáticas en *Resident Evil 2* [5] [2].

En este tipo de cámara las propiedades de la cámara como su posición, orientación y campo visual (*field of view*) son colocadas durante el desarrollo del juego y estas no cambian durante el *gameplay*. Algunos ejemplos de juegos con este tipo de camara son los primeros Resident Evil, y Alone In The Dark, usualmente es utilizada para crear tension [3][5].

4.2. Seguidoras (*Tracking*)

Este tipo de cámara sigue a algún objeto en el juego usualmente el personaje principal u otro objeto de considerable importancia. Este sistema presenta varios problemas sobretodo en ambientes tridimensionales y tercera persona donde la cámara podría quedar detrás de una objeto que ocluye totalmente la vista o no deja ver algún objeto de interés al jugador [10]. Su uso es muy común en los primeros juegos 3D en tercera persona como *Crash Bandicoot* [7] o *Tomb Raider* [4, p. 39], los juegos primera persona también utilizan una cámara seguidora, a diferencia de en tercera persona donde la cámara esta detrás del personaje en un juego primera persona la cámara esta como visión del personaje principal, en los juegos 2D esta cámara esta presente en todos los juegos tipo side-scroller.

4.3. Interactivas (*Interactive*)

Las cámaras interactivas son cámaras que cambian su posición, campo visual u orientación según distintas situaciones, lugares o intereses del juego y sus desarrolladores, usualmente estas cámaras poseen alguna forma de inteligencia artificial. En su mayoría las cámaras interactivas son cámaras tracking mejoradas, estas siguen al personaje como lo hace una cámara tracking pero su posición y orientación puede cambiar si estas se ven obstruidas por algún objeto o cambian sus parámetros para mostrar objetos de interés claramente evitando de esta forma las desventajas principales de una cámara tracking. Algunos ejemplos de juegos con este tipo de camara son *Super Mario 64*, *Super Mario Sunshine*, *The Legend of Zelda: The Wind Waker.*

Figura 5: En *Super Mario 64* la cámara rota de forma inteligente para mostrar el camino [8].

5. Actividad

- Durante esta actividad se creara la base del proyecto y se implementara inicialmente la cámara según el diseño de su juego. Los objetivos de la actividad son:

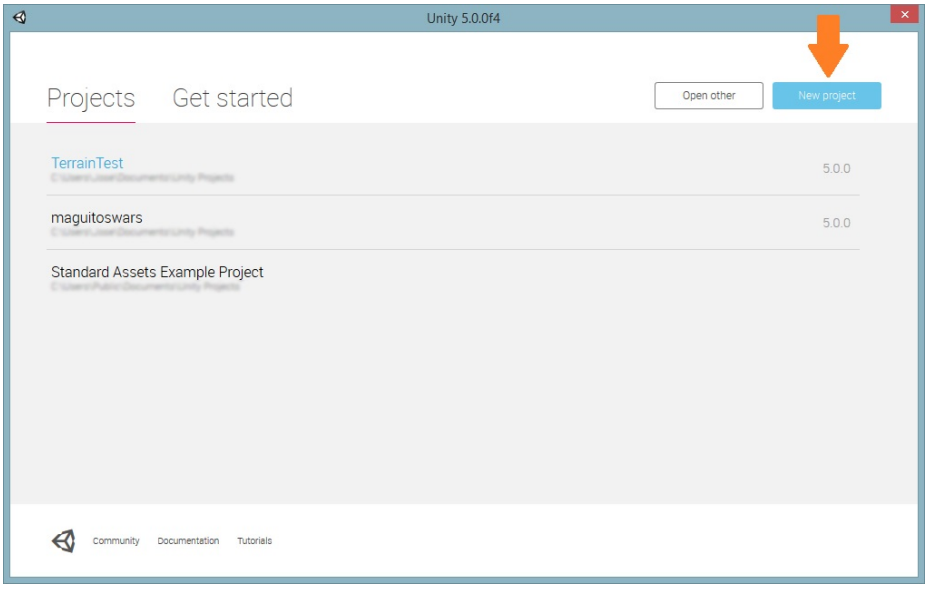
  1. Crear una pequeña escena con *placeholders* [1].
  2. Implementar la cámara principal de juego según el diseño del juego.
  3. Debe crear una estructura de datos (ejemplo una clase) que defina a los actores principales en su mundo de juego, incluyendo el jugador principal. Esto depende directamente del diseño del juego.
  4. Debe asociar esta estructura a un *placeholder* que representa cada actor, incluyendo el jugador principal.
  5. Agrege en la lógica de juego (esto se realiza usualmente a través de scripts u otro medio que provea las herramientas que eligió) comportamientos a uno o varios actores en escena, este comportamiento debe ser observable por la cámara.

[1] Un *placeholder* no es mas que un procurador de un objeto, de manera que el *placeholder* es remplazado en un futuro por el objeto real.

5.1. Ejemplo en Unity3D

En esta sección se muestra como ejemplo la realización de la actividad del laboratorio en Unity3D, según su herramienta de trabajo algunas cosas puede ser muy parecidas o muy diferentes por esto es recomendado en el pre-laboratorio investigar sobre su herramienta de trabajo igualmente en la actividad del laboratorio 1 se sugieren algunos parámetros para escoger su herramienta. Para esto usaremos el motor de juego Unity v5.0.

Luego de descargar, instalar el motor de juego Unity3D (en el laboratorio 1 se encuentran los distintos links para cada motor y framework recomendado) para empezar un juego primero tenemos que crearlo, para esto ejecutamos *Unity.exe* normalmente encontrado en *[Carpeta de Instalacion]/Editor/Unity.exe* y creamos un nuevo proyecto dando click sobre *New Project*.



Luego se debe colocar el nombre del proyecto de juego (1), seleccionar entre Unity 3D o 2D (2) en nuestro caso seleccionaremos 2D ya que el juego a realizar durante este laboratorio general debe ser 2D, sin embargo se puede crear también juegos 2D en modo Unity 3D utilizando proyección ortogonal para la camara y finalmente debemos crear el proyecto dando click sobre *Create Project* (3).

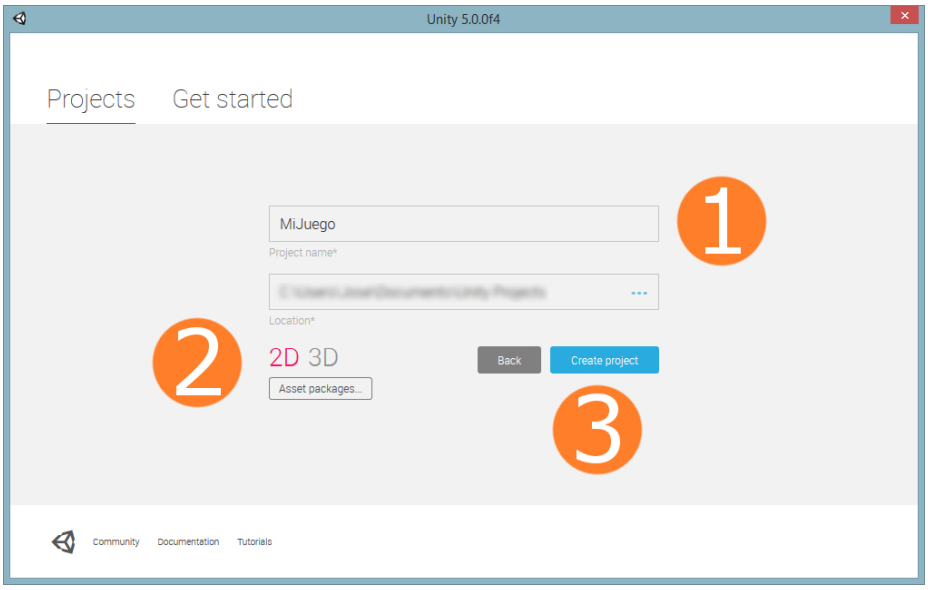
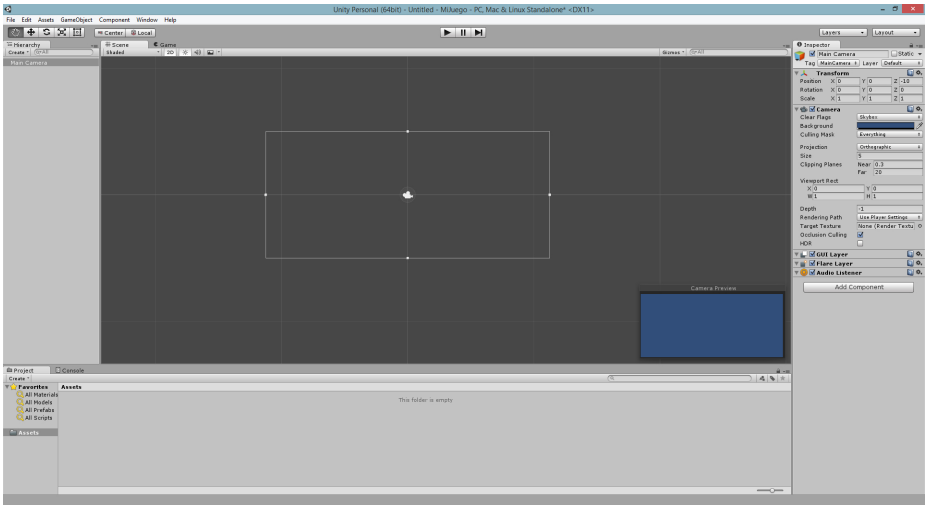


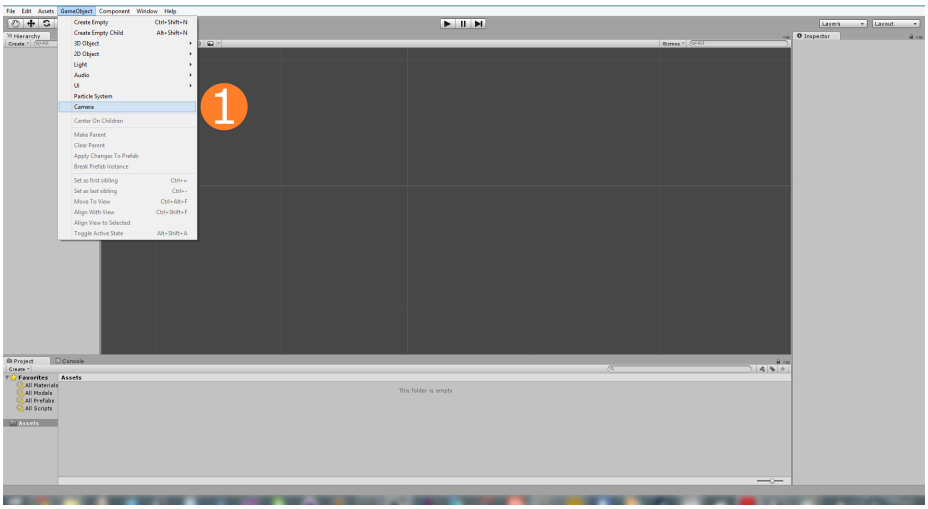
Figura 6: Nombre del juego (1), Modo 2D o 3D (2), Crear Proyecto (3)

Esto nos lleva directamente al editor de escena en Unity, la terminología utilizada para describir las distintas partes de la interfaz del motor de juego suele cambiar entre motores. En el caso de Unity al iniciar por primera vez nos muestra la interfaz por defecto donde podemos ver la jerarquía de los objetos de escena en el panel izquierdo, las propiedades del objeto actualmente seleccionado en el panel derecho, el explorador de archivos del proyecto en el panel inferior y la escena en el centro como foco principal. Para desplazarse en escena puede usar click derecho y central sostenido y las teclas direccionales.

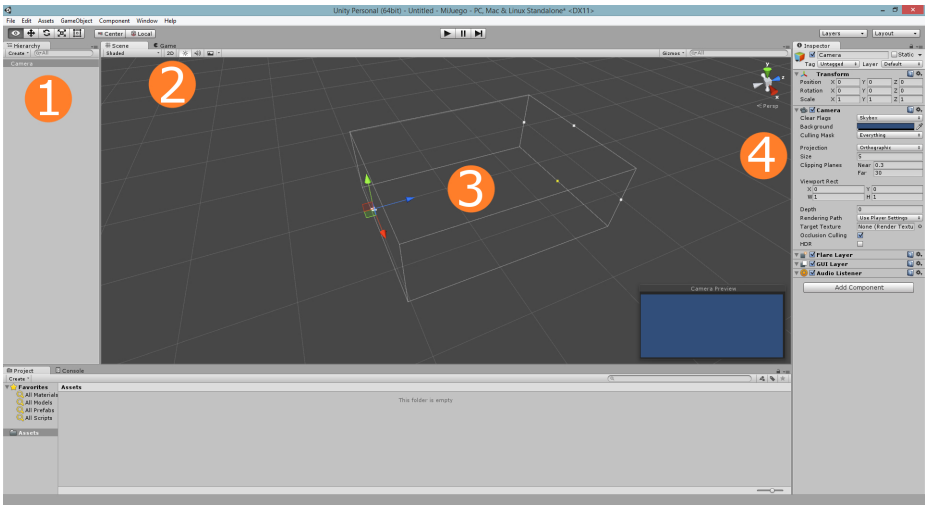


Por defecto Unity nos crea una cámara como objeto de escena al iniciar un proyecto, esta con proyección ortogonal para modo 2D y perspectiva para modo 3D, vamos a eliminar este objeto de escena y crear una cámara por nuestra cuenta. Para eliminar la cámara damos click derecho sobre *Main Camera* en la jerarquía de objetos (panel izquierdo) y click sobre *Delete*.

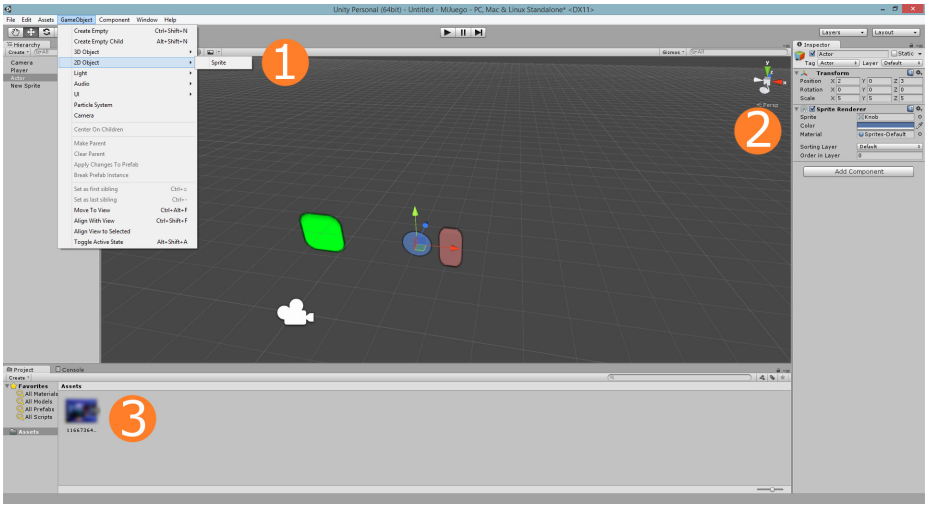
Para crear un objeto en escena de tipo cámara en Unity debemos ir a Menu de Herramientas (arriba), dar click sobre *Game Object* y luego en *Camera* (1).



Por defecto la cámara es creada con proyección perspectiva, podemos cambiar esto a conveniencia, dando click sobre el nuevo objeto *Camera* creado (1) en las propiedades de objeto (panel izquierdo) podemos cambiar el tipo de proyección a perspectiva u ortogonal (perspective y orthographic) entre otras cosas como la distancia de los clipping planes (Sección 3.3), posición de la cámara, rotación de la cámara, tamaño de la ventana de vista (Viewport Rect), tamaño para proyección ortogonal, campo de visión (Sección 3.2) para proyección perspectiva, etc (4). Si damos click sobre el botón *2D* (2) en el editor de escena intercambiar entre ver la escena en 2D o 3D, en modo 2D con la cámara seleccionada es fácil observar el volumen de la vista de la cámara (3).

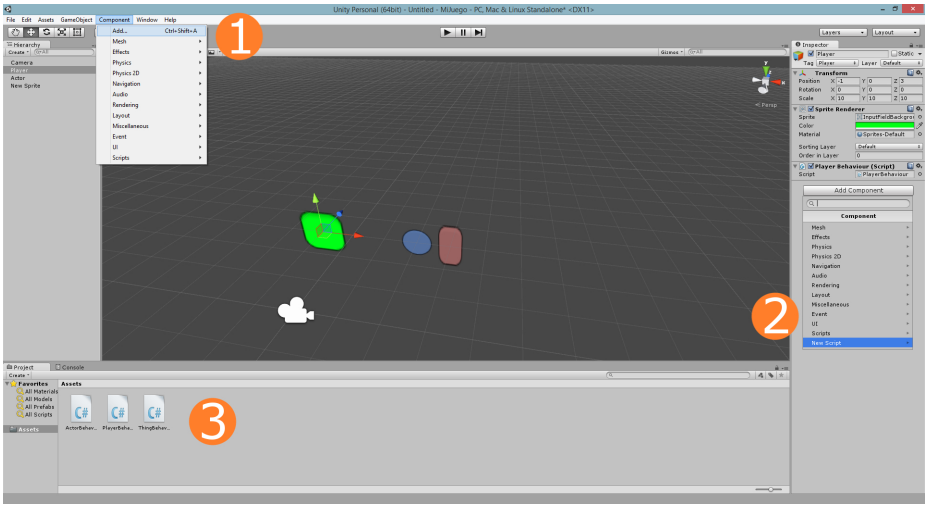


Igualmente como creamos la cámara vamos crear objetos placeholder para representar el jugador principal, actores y ambiente en distintas posiciones y tamaños, solamente que esta vez en vez de cámaras vamos a crear *2D Object* → *Sprite* y *3D Object* (1). Cada objeto es recomendado nombrarlos adecuadamente (2) y categorizarlos (en Unity esto se hace con el uso de *Tags*). Para los sprite usaremos en propiedades *Sprite* alguno por defecto o se puede usar alguno importado (3) por *Assets* → *Import New Asset* (estos deben ser imágenes en este caso) y en *Color* alguno de preferencia para diferenciar, recordemos que estos son placeholders.



Vamos a crear ahora scripts asociados a cada uno de estos objetos, estos scripts en

un futuro contendrán distintas propiedades, comportamientos y recursos asociados a estos objeto, en Unity un objeto puede tener asociado un numero indefinido de scripts. Para asociar un script a un objeto seleccionamos este objeto, en la barra de herramientas (arriba) damos click sobre *Component* → *Add...* (1) esto abre un menú desplegable sobre las propiedades del objeto, seleccionamos *New Script* (2) y colocamos un nombre apropiado, Unity agrega estos scripts a los *Assets* del proyecto (3).



Para editar estos scripts Unity trae un editor por defecto llamado MonoDevelop, sin embargo se puede usar otros editores externos para modificar los scripts, una de las ventajas de MonoDevelop es que se puede compilar el proyecto desde el mismo editor, sin embargo existen plugins para otros editores como Visual Studio que permiten esto. El formato por defecto de los scripts en C# en Unity es el siguiente.

```
using UnityEngine;
using System.Collections;
public class PlayerBehaviour : MonoBehaviour {
        // Use this for initialization
        void Start () {}
        // Update is called once per frame
        void Update () {}
}
```

Unity importa por defecto *UnityEngine* y las estructuras en *System.Collections*, el

template presenta dos procesos *Start* y *Update*. *Start* se ejecuta cuando se crea las instancia del objeto asociado al script. *Update* se ejecuta por cada frame. Agreguemos propiedades al objeto *Player* a través del script, en este caso agreguemos un parámetro de vida (*Health*) y velocidad (*Speed*).

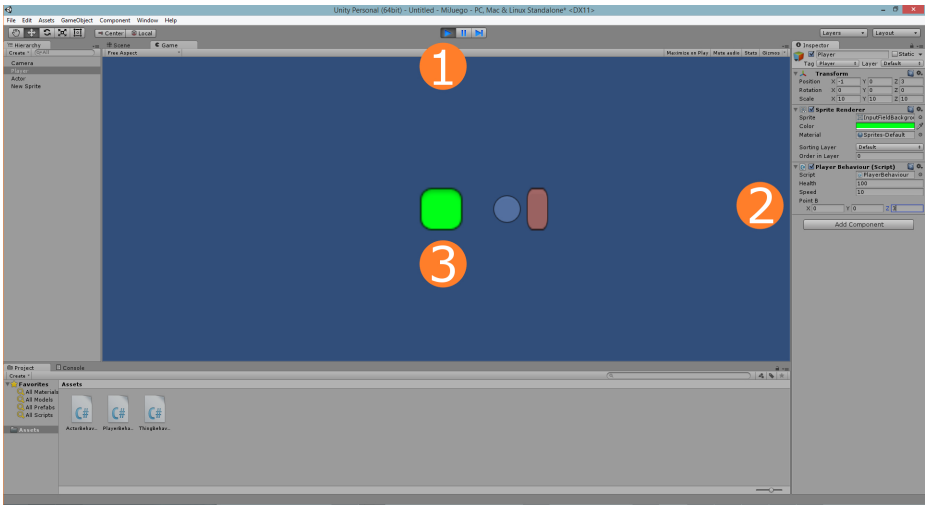
```
public class PlayerBehaviour : MonoBehaviour {
        public float Health = 100.0f;
        public float Speed = 10.0f;
        // Use this for initialization
        void Start () {}
        // Update is called once per frame
        void Update () {}
}
```

Esta propiedad debe aparecer en el componente *Script* asociado al objeto *Player* ya que estos son públicos. Uno de los objetivos de la actividad es realizar ciertas acciones sobre algún objeto de manera que esto sea notable por la cámara. Hagamos entonces en el script que el jugador se mueva entre dos puntos constantemente, para esto debemos modificar la transformada del objeto, la transformada posee información de la posición, rotación y escala del objeto asociado al script.

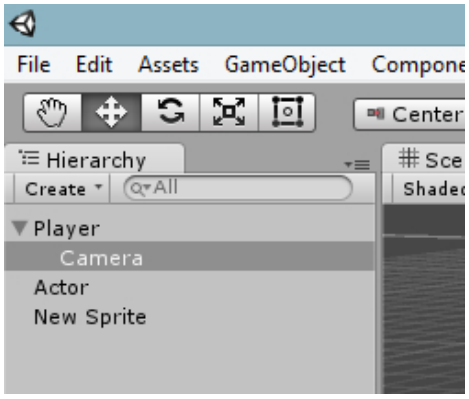
```
public class PlayerBehaviour : MonoBehaviour {
        public float Health = 100.0f;
        public float Speed = 10.0f;
        // Movement control vars
        private Vector3 _pointA;
        public Vector3 PointB;
        // Use this for initialization
        void Start () {
                _pointA = this.transform.position;
        }
        // Update is called once per frame
        void Update () {
                this.transform.position = Vector3.Lerp(_pointA,
```

```
                PointB, Mathf.Cos(Time.time * Speed));
        }
}
```

Lo que hacemos acá es modificar la posición del objeto *Player* según una operación de interpolación lineal (*lerp*) entre un dos puntos 3-dimensionales *_pointA y PointB* (2) con un parámetro en -1 y 1 a una velocidad *Speed*. Con esto ahora al dar click sobre el botón *Play* (1) podemos ver con la ejecución de los scripts en la cámara al objeto *Player* (3) moviéndose.



Otro requerimiento en la actividad consiste en implementar la cámara según el diseño del juego.



En este ejemplo se asume una cámara seguidora tipo side-scroller. Esto es sumamente sencillo en Unity utilizando la jerarquía de objetos, para hacer a la transformada de *Camera* dependiente de la transformada del *Player* simplemente colocamos a *Camera* como hijo del objeto *Player*, para hacer esto realizamos click sostenido sobre el objeto *Camera* en la jerarquía y lo colocamos sobre el objeto *Player*. Ahora la cámara se mueve junto con el jugador.

Referencias

[1] Feng Zhu School of Design: FoV in Video Games. http://artsygamer.com/fov-in-games/, 2014.

[2] Capcom. Resident Evil 2. http://www.giantbomb.com/resident-evil-2/3030-9418/, 1998.

[3] Casamassina, M. Resident Evil Review. http://uk.ign.com/articles/2002/04/26/resident-evil-3?page=3, 2002.

[4] Gates, B. *Game Design (2nd Edition)*. Thomson Course Technology, 2004.

[5] GiantBomb Community. Fixed Camera: Concept. http://www.giantbomb.com/fixed-camera/3015-1715/, 2014.

[6] Microsoft. What Is a View Frustum? https://msdn.microsoft.com/en-us/library/ff634570.aspx, 2014.

[7] Naughty Dog Software. Crash Bandicoot. http://www.ign.com/games/crash-bandicoot/ps-603, 1996.

[8] Nintendo. Super Mario 64. http://www.ign.com/games/super-mario-64/n64-606, 1996.

[9] Unity Technologies. Unity Manual: Camera. http://docs.unity3d.com/Manual/class-Camera.html, 2014.

[10] Rollings Andrew, E. A. *Fundamentals of Game Design*. Prentice Hall, 2006.