Aspectos tecnológicos en Proyectos de Esports

Clase 2

Clase 2

Visualización 2D

01

Motores de juego
Elección del motor de juego

Creación de un videojuego 2D

02

Unity: entorno de trabajo
Desarrollo de un videojuego 2D

Aspectos tecnológicos en Proyectos de Esports

Clase 2

Visualización 2D

01

Motores de juego
Elección del motor de juego

Creación de un videojuego 2D

02

Unity: entorno de trabajo
Desarrollo de un videojuego 2D

Aspectos tecnológicos en Proyectos de Esports

Motores de juego

Existen una gran diversidad de motores de juego que permiten el desarrollo de videojuegos en **dos y tres dimensiones**. La **elección del motor de juegos adecuado** es muy importante para el futuro proceso de desarrollo. Algunos factores a considerar puede ser los siguientes:

- Tipo de juego a desarrollar
- Desarrollo 2D o 3D
- Experiencia con la que se cuenta
- Comunidad del motor de juegos
- Recursos disponibles para el motor de juegos

Motores de juego

Las herramientas que proporciona cada motor de juego pueden variar, pero en todos los casos están diseñadas para **eliminar parte de la complejidad del desarrollo** de un videojuego.

Antiguamente, el solo hecho de intentar mostrar un objeto 3D en pantalla era muy complejo, aún sin tener en cuenta la cantidad de características relacionadas a esto, como por ejemplo la iluminación de la escena, la animación del objeto, entre otras.

Todos **estos puntos llevaban mucho tiempo de desarrollo**, y en muchos casos ni siquiera era una parte importante de la mecánica del juego, sino que eran solo partes de una visualización de una pantalla.

Motores de juego

Los motores de juego proporcionan **características preconstruidas** que hacen mucho más sencillo el desarrollo de un videojuego. Algunas de estas características son:

- Representación de gráficos 2D y 3D.
- Cálculo de la física.
- Manejo de detección de colisiones.
- Manejo de interfaces de usuario
- Soporte de animación, audio y video.
- Scripting
- Soporte multijugador y de herramientas de red.

Motores de juego

Abstraerse de cómo están resueltas estas características, hace que el desarrollador pueda **enfocarse en la lógica y la estética** del futuro videojuego.

Además, los motores de juego generalmente permiten **importar librerías y frameworks externos**, que contienen funcionalidad específica que resuelve un tema en particular.

Estos componentes externos son generalmente fáciles de integrar en los proyectos y son una buena alternativa para la creación de juegos que necesitan de **características especiales**, como lo pueden ser la Realidad Virtual (RV) y la Realidad Aumentada (RA).

Motores de juego

Entre los motores de juego más conocidos y utilizados podemos nombrar:

- Unity
- Unreal Engine
- Godot
- CryEngine
- Phaser

Motores de juego

- **Unity**

Desarrollado desde 2005, Unity es uno de los motores gráficos más **conocidos y utilizados**, sobre todo en la **industria de los juegos independientes**.

El motor es **adecuado para juegos 2D, 3D** y también es una opción popular para la creación de juegos con **RV** o **RA**.

Unity es **gratis** para usuarios principiantes y ofrece más de **25 plataformas de publicación**, lo que permite desarrollar la aplicación una única vez y luego exportarla para su uso en diferentes plataformas.

Motores de juego

Entre las características más importantes se puede nombrar la gran cantidad de **documentación** disponible, una enorme y muy activa **comunidad** de usuarios, una amplia variedad de **componentes pre-desarrollados** (assets) y **plugins** que facilitan la integración con otras herramientas.

Unity permite programar en **C#** y, haciendo uso de los **manuales y tutoriales** existentes, su aprendizaje no es nada complejo.

Unity **se actualiza con bastante frecuencia**, pero mantiene ciertas versiones con **soporte a largo plazo** (LTS, del inglés Long Term Support)

Motores de juego

- **Unreal Engine**

Creado por **Epic Games** en 1998, es uno de los **motores gráficos más destacados** dentro de la industria del videojuego.

Debido a sus **sólidas capacidades gráficas**, iluminación dinámica, sombreadores y otras características, Unreal es uno de los motores más elegidos para el **desarrollo de videojuegos hechos por grandes compañías** de la industria, con altos costos tanto de desarrollo como de marketing.

Unreal no es el más adecuado para la creación de juegos en 2D.

Motores de juego

UNREAL ENGINE

Al igual que Unity, Unreal tiene también un **Marketplace** en donde se pueden obtener assets gratuitos y pagos.

Unreal es más adecuado para **proyectos grandes** que involucran un equipo de desarrollo considerable.

El motor de juegos es **pesado en el aspecto gráfico** y requiere una computadora más potente, en comparación con otros motores de juego.

Unreal también permite exportar los videojuegos a variadas plataformas: PC, Mac, iOS, Android, Xbox, PlayStation, entre otras.

Motores de juego

UNREAL ENGINE

Como lenguaje de programación utiliza **C++** en combinación con un lenguaje visual denominado **Blueprints**.

La utilización de **C++** brinda a los desarrolladores un **gran control sobre las acciones de todo el sistema**, pero a su vez vuelve el proceso de codificación mucho **más complejo**.

Los **Blueprints** son un tipo de programación basada en nodos que sirve para implementar la lógica completa de una aplicación de una **forma sencilla**.

Utilizar C++ puro es más eficiente, pero para proyectos sencillos **no existe diferencia apreciable** entre ambas alternativas de codificación.

Motores de juego

- **Godot**

Godot existe desde 2014 pero ha ganado popularidad en los últimos años. Es un motor de videojuegos multiplataforma, **libre y de código abierto** para sistemas Windows, OS X, Linux y BSD.

Permite exportar los videojuegos creados a PC (Windows, OS X y Linux), teléfonos móviles (Android, iOS), y HTML5.

Godot se ha consolidado como una **fuerte alternativa** a los otros motores vistos, sobre todo para proyectos llevados a cabo por **equipos pequeños** o desarrolladores que trabajan en solitario.

Motores de juego

GODOT
Game engine

Los videojuegos creados con Godot pueden ser programados con **C++, C#** o bien **GDScript**, un lenguaje de script propio, de alto nivel y tipado de forma dinámica muy sencillo de utilizar gracias a su simple sintaxis.

Godot también soporta **Visual Scripting**, una buena característica para personas que no tengan grandes conocimientos de programación y prefieran crear la lógica utilizando nodos y conexiones.

Al no ser tan popular como los anteriores motores vistos, en comparación **no hay tantos recursos disponibles**.

Motores de juego

- **CryEngine**

CryEngine es un motor gráfico muy potente desarrollado en 2002 por Crytek. El motor está diseñado para usarse en juegos de PC y consolas, incluyendo PlayStation y Xbox.

A pesar de no ser tan popular, las **capacidades gráficas y potencia** de CryEngine pueden llegar a ser mayores que en Unity, y asemejarse a las de Unreal.

CryEngine permite programar en **C++** y **Lua**.

Motores de juego

CRYENGINE®

Es un motor gráfico que se destaca por su **calidad y potencia**, pero no cuenta con demasiadas facilidades para crear diferentes tipos de aplicaciones, sino que esta más orientado a crear videojuegos de tipo "**shooter**".

Debido a esto último, sumado a que **es necesario invertir bastantes horas para aprender a utilizarlo correctamente**, no se puede considerar la mejor opción para iniciarse en la creación de juegos propios.

Motores de juego

- **Otros motores de juego 2D**

  - GameMaker
  - Phaser (HTML5)
  - Construct
  - GDevelop
  - Ren'Py

Motores de juego

- **Estadísticas de uso**

GGJ 2023 total	7625	
Engine	**Games**	**Percentage**
Unity	4669	61,23 %
Unreal Engine	715	9,38 %
Godot Engine	500	6,56 %
Game Maker	167	2,19 %
Construct 3	93	1,22 %
YAHAHA Studio	86	1,13 %
Ren'Py	48	0,63 %

GGJ 2024 total	9857	
Engine	**Games**	**Percentage**
Unity	3549	36,00 %
Godot Engine	905	9,18 %
Unreal Engine	757	7,68 %
Game Maker	140	1,42 %
Construct	99	1,00 %
Ren'Py	54	0,55 %
Gdevelop	42	0,43 %

Elección de un motor de juego

No existe un motor de juego que sea **óptimo en todos los aspectos**, pero **Unity** es uno de los más recomendables para cuando se está **iniciando en el desarrollo** de videojuegos. Entre los factores que generan un escenario ideal para **aprender rápidamente** a usar Unity podemos encontrar:

- **Tutoriales**: hay una gran cantidad de tutoriales y ejemplos que guían al nuevo usuario en el proceso de aprendizaje. Estos tutoriales están categorizados según el tipo de desarrollo, son audiovisuales y están provistos de todos los elementos necesarios para realizarlos (objetos, audios, imágenes, scripts, entre otros). Esto hace que la

Elección de un motor de juego

- **Documentación**: el manual de usuario es amplio, de fácil comprensión y está subdividido en distintas categorías. Contiene un buscador que facilita la búsqueda de un tema o funcionalidad en particular.

- **Componentes disponibles**: cuenta con un repositorio (Asset Store) en donde es posible encontrar una gran variedad de componentes que pueden acelerar el desarrollo de las aplicaciones. El buscador de componentes permite realizar búsquedas complejas mediante distintos tipos de filtro.

Elección de un motor de juego

- **Comunidad**: la gran popularidad de Unity es un factor preponderante. Al ser el motor de juegos más utilizado, su comunidad está compuesta por millones de usuarios. Contiene un foro subdividido en categorías en donde es posible plantear las situaciones o problemas que pueden surgir durante el desarrollo de una aplicación.

- **Requerimientos de software/hardware**: los requisitos de sistema son considerablemente más bajos en comparación con otros motores de juego. Su instalación es simple, lo que incentiva al desarrollador a comenzar rápidamente con la utilización del motor de juegos.

Clase 2

Visualización 2D

Motores de juego
Elección del motor de juego

01

Creación de un videojuego 2D

Unity: entorno de trabajo
Desarrollo de un videojuego 2D

02

Aspectos tecnológicos en Proyectos de Esports

Unity: entorno de trabajo

La interfaz de usuario de Unity contiene la barra de herramientas y una serie de **vistas**. Las más importantes son:

- **Hierarchy**. Muestra la escena actual en forma de árbol. Nos permite ver la estructura de la escena con todos sus componentes.
- **Scene**. Muestra el área de construcción de la escena de juego en forma gráfica.
- **Game**. Contiene una vista previa del juego, tal y como se verá en la pantalla del jugador.

Unity: entorno de trabajo

- **Asset Store**. Es la tienda de componentes. Generalmente se organizan en paquetes que se pueden importar en el proyecto. Algunos paquetes son gratis y otros son pagos.
- **Inspector**. Permite ver los detalles del objeto/componente que este seleccionado, pudiendo cambiar sus propiedades y configuración.
- **Console**. Permite visualizar los mensajes de consola. En esta sección se indica de cualquier error que pueda haber en la aplicación. Otro uso importante de la consola es cuando le enviamos información para ir chequeando el funcionamiento de la aplicación.

Unity: entorno de trabajo

- **Proyect**. Aquí se almacenan todos los componentes necesarios para el proyecto. Cuenta con dos subcarpetas principales:
  - **Assets**. Generalmente, los componentes de un proyecto son muchos y es útil organizarlos en subcarpetas. Por ejemplo, en una subcarpeta “Scenes” se guardarán las diferentes escenas del proyecto; en otra subcarpeta todos los scripts, en otra todas las imágenes, en otra todos los sonidos, etc.
  - **Packages**. Son los paquetes de componentes que se importan en un proyecto y se usan en el desarrollo de la aplicación.

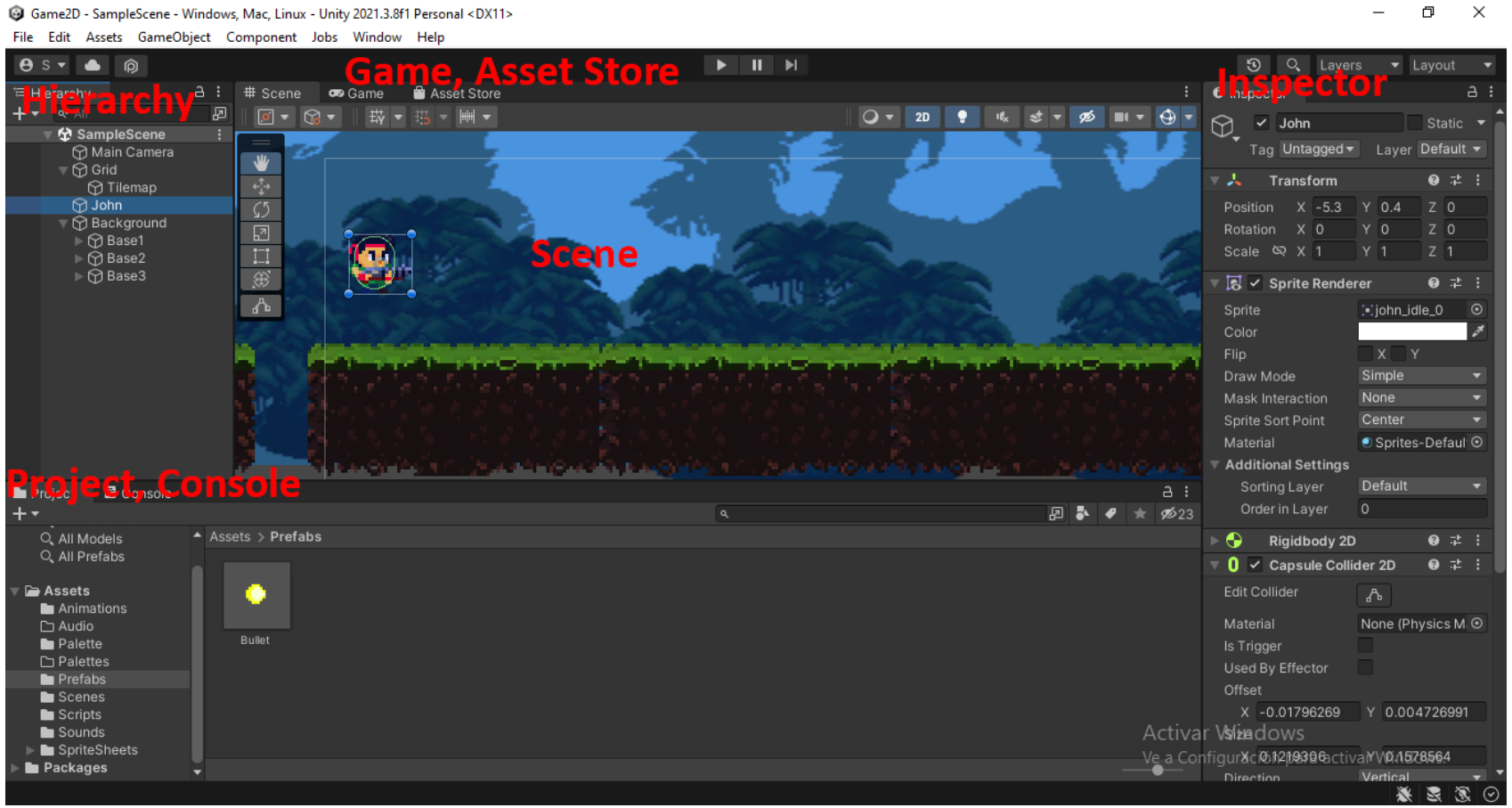
Unity: entorno de trabajo

El **MonoDevelop** fue originalmente el **editor de scripts** predeterminado en Unity y se instalaba por defecto con el motor de juegos. Actualmente, **Visual Studio Code** es uno de los editores más utilizados. Una vez dentro de Unity, **es posible elegir el editor** que se desea utilizar.

Cuando se crea un nuevo script en Unity, por defecto se añaden algunas líneas de código importando algunas **librerías**.

Estas librerías permiten emplear **clases y métodos propios de Unity**, como pueden ser las clases GameObject, Camera, Sprite y los métodos Start() y Update().

Unity: entorno de trabajo



Desarrollo de un videojuego 2D

Algunos conceptos básicos por conocer al momento de desarrollar un videojuego 2D son:

- **Sprite**. Imagen de mapa de bits presente en pantalla, que generalmente se refiere a los personajes del juego o a objetos interactivos.
- **Tileset**. Ayuda a construir escenarios 2D mediante un proceso sencillo y rápido.
- **Tilemap**. Un mapa o escenario construido a partir de un tileset.

Desarrollo de un videojuego 2D

- **Collider**. Define la forma de un objeto para sus colisiones físicas. Es invisible y no necesita tener exactamente la misma forma que el objeto (de hecho, una aproximación es más eficiente).

- **Script**. Permite dar el comportamiento deseado a los objetos. A través de diferentes eventos es posible modificar propiedades de los componentes en el tiempo y responder al input del usuario.

Desarrollo de un videojuego 2D

Para aprender a utilizar Unity y adquirir los **conocimientos básicos** para la creación de un videojuego 2D, **desarrollaremos un prototipo sencillo** de videojuego.

Usaremos un **package externo** bajado desde un sitio web para acelerar algunos pasos del proceso de desarrollo, como puede ser el diseño de los personajes y otros elementos multimedia. Lo pueden bajar desde: https://didigameboy.itch.io/jambo-jungle-free-sprites-asset-pack

La **versión de Unity** que se usará es la **2021.3.8f1**.

Desarrollo de un videojuego 2D

- **Creación del proyecto**

Inicialmente instalamos **Unity Hub**. Esto nos permitirá manejar las diferentes versiones de Unity y nuestros diferentes proyectos.

Desde Unity Hub, creamos un nuevo proyecto 2D:

- **Project → New Project → 2D**

Elegimos un nombre para el proyecto y la ruta de nuestro **workspace**, que es en donde se guardará nuestro nuevo proyecto.

Desarrollo de un videojuego 2D

- **Incorporación de assets predesarrollados al proyecto**

Para avanzar rápidamente en el desarrollo del prototipo de juego 2D, vamos a utilizar algunos **assets predesarrollados** del package externo que descargamos. Copiamos la carpetas "**Sounds**" y "**SpriteSheets**" dentro de la carpeta "**Assets**".

En el caso de estar desarrollando un juego desde cero, deberíamos crear cada uno de los personajes, paisajes, sonidos, música, etc.

También es posible incorporar componentes desde un paquete del **Asset Store**. Estos componentes quedan almacenados en la carpeta "**Packages**".

Desarrollo de un videojuego 2D

- **Creación de una paleta**

Creamos una paleta en donde tendremos todas las imágenes para crear el mundo de nuestro juego, es decir, el ambiente por donde se moverá el personaje.

Abrimos la ventana de manejo de paletas: **Window → 2D → Tile Palette**, y si lo deseamos la anexamos a alguna área de nuestra vista, para tenerla cerca y hacer mas cómodo su uso.

Desarrollo de un videojuego 2D

Creamos una nueva paleta, le damos un nombre y la guardamos en una nueva carpeta, por ejemplo, “**Palettes**”.

Configuramos la imagen “**tile_spridesheet**” de la carpeta “SpriteSheets” que habíamos agregado, para que sea interpretada como un set de sprites. Para eso elegimos el modo “**Multiple**”.

Además, seleccionamos el modo de filtro **Point** en lugar de **Bilinear**, ya que este último difumina los puntos y en este caso queremos un estilo de juego pixelado.

Desarrollo de un videojuego 2D

Entramos a “**Sprite Editor**”, y separamos los pequeños sprites para que Unity pueda interpretarlas correctamente: en el menú **Slice**, elegimos **Automatic** y **Smart**, y presionamos **“Slice”**

Como se ajusta al tamaño de cada sprite, algunos quedan de menor tamaño, podemos editarlos si nos es más cómodo que queden todos iguales.

Seleccionamos solo los sprites que vamos a usar, y finalmente presionamos “**Apply**” y arrastramos la imagen a nuestra paleta (seleccionamos de vuelta la carpeta “Palettes” como destino).

Desarrollo de un videojuego 2D

- **Creación del terreno**

Creamos nuestro Tilemap y al Grid le agregamos pequeños sprites formando el terreno (se puede hacer con varios sprites a la vez).

- **2D Objects → TileMap → Rectangular**

Se deberá ajustar el **tamaño de las celdas del grid** teniendo en cuenta el tamaño de los sprites utilizados (para este ej. 0.08, porque los sprites tienen 8 pixeles).

También es posible configurar el **size** y **posición** de la cámara, para ajustar el terreno a la pantalla.

Desarrollo de un videojuego 2D

- **Creación del personaje**

Tomamos la imagen "**player/john_idle**" y separamos un **único sprite** del personaje de forma similar a lo hecho con el terreno. Luego creamos nuestro personaje arrastrando el sprite a la escena.

Agregamos el componente de física al personaje:

- **Add component → Rigidbody 2D**.

La propiedad "**Gravity**" está en 1, para que el personaje caiga. Podemos bajarla o subirla, dependiendo de si cae muy rápido o lento.

Desarrollo de un videojuego 2D

- **Incorporación de colliders para manejo de colisiones**

Para que el personaje pueda estar sobre el terreno, debemos ir al TileMap que creamos y agregarle el "**Tilemap Collider 2D**". Esto agrega una "caja de colisiones" por cada imagen.

En este caso no tiene sentido tener cajas independientes, ya que nuestro terreno esta todo unido, y la cantidad de colliders bajará la performance final de la app. Por esto, agregamos además un "**Composite Collider 2D**", para que junte todas las cajas de colisión. Para que funcione, en el Tilemap activamos "**Used by Composite**".

Desarrollo de un videojuego 2D

Al agregar el “Composite Collider 2D” automáticamente se agrega un **Rigidbody 2D**, que debemos setear como **estático**, ya que el terreno no se va a mover.

Por último, le agregamos el collider al personaje. En este caso usamos “**Capsule collider 2D**”. Es posible modificar su forma usando “**Edit collider**” si así lo quisiéramos.

Al terminar de agregar la física (rigidbody y colliders), ahora el personaje se mantiene sobre el terreno.

Desarrollo de un videojuego 2D

- **Control del personaje**

Una vez creados el terreno y el personaje, es momento de agregar los scripts necesarios para **controlar el personaje**.

Creamos una carpeta “**Scripts**” en assets y creamos un nuevo script (**C#**) llamado “PlayerScript”.

Agregamos el script al personaje, para entonces poder acceder a las características de dicho personaje.

Desarrollo de un videojuego 2D

Cuando creamos un script, Unity automáticamente genera **dos funciones**:

- **Start**: el código que pongamos acá se ejecutará **una sola vez**, al inicio de la app.
- **Update**: el código que pongamos acá se ejecutará **una vez por frame** (cuadro). Si el juego va a una velocidad de 60 FPS, se ejecutara 60 veces por segundo.

Desarrollo de un videojuego 2D

Agregamos las variables privadas y publicas que luego vamos a necesitar:

```
private Rigidbody2D rb2D; // ref. al rigidbody del personaje
private float horizontal;
public float speed = 2;
public float jumpForce = 150;
```

En **Start**, obtenemos el componente rigidbody y lo guardamos:

```
rb2D = GetComponent<Rigidbody2D>();
```

En **Update**, obtenemos la entrada del usuario:

```
horizontal = Input.GetAxisRaw("Horizontal");
```

Desarrollo de un videojuego 2D

Agregamos una función **FixedUpdate**, que es más constante que **Update**, así la frecuencia no depende del dispositivo en el que se ejecuta la app.

En esta función modificamos la posición del personaje en función de las teclas presionadas por el usuario y una variable publica **speed** que podremos configurar con el valor deseado:

```
private void FixedUpdate() {
   rb2D.velocity = new Vector2(horizontal * speed, rb2D.velocity.y); }
```

Para que el personaje no se mueva "rotando", seteamos una restricción en el RigidBody2D: **Constraints → Freeze Rotation (Z)**.

Desarrollo de un videojuego 2D

Si queremos que el personaje pueda saltar, creamos una nueva función "**Jump**", la cual invocaremos desde **Update**:

```
if (Input.GetKeyDown(KeyCode.UpArrow)) {
        Jump();
}

void Jump() {
        rb2D.AddForce(Vector2.up * jumpForce);
}
```

También agregamos la variable publica **jumpForce** para setear la potencia de salto (usar un valor alto mayor a 100).

Desarrollo de un videojuego 2D

Por último, agregamos un breve código para que el personaje mire hacia el lado correcto al moverse.

Para lograr esto, se debe modificar el **localScale** del personaje, dependiendo de su orientación. Agregamos en **Update**:

```
if (horizontal < 0.0f)
        transform.localScale = new Vector3 (-1.0f, 1.0f, 1.0f);
else if (horizontal > 0.0f)
        transform.localScale = new Vector3 (1.0f, 1.0f, 1.0f);
```

Desarrollo de un videojuego 2D

Generación de disparos

Si deseamos que el personaje pueda disparar su arma, inicialmente debemos preparar el **Sprite** de la bala usando el sprite que esta en "**SpriteSheet/Player/weapon_bullet**".

Seteamos **Point**, **Multiple** y en el **Sprite Editor** usamos el tipo de corte por tamaño de celda (**8x8**) para realizar la separación de los sprites.

Arrastramos a la escena **la primera imagen de la bala** y le damos un nombre, por ejemplo "Bullet".

Desarrollo de un videojuego 2D

Ahora debemos crear una **ANIMACION** para la bala.

Seleccionamos todos los sprites de la bala, click derecho **Create** → **Animation**. Le damos un nombre y la movemos a una nueva carpeta "**Animations**".

Activamos LoopTime y luego arrastramos la animación al objeto "Bullet". Si ejecutamos el juego veremos la bala "animada".

Agregamos a "Bullet" un RigidBody2D (con Gravity Scale en 0, para que no caiga) y CircleCollider2D.

Desarrollo de un videojuego 2D

Por último, agregamos un **script para mover la bala**. En carpeta scripts: new C# script: BulletScript y lo agregamos al objeto Bullet.

En este script vamos a mover la bala hacia donde el personaje esta mirando y en la velocidad seteada en la variable publica **speed**.

```
public float speed = 3.0f;
private Rigidbody2D rb2D;
private Vector3 direction;
```

En **Start**, guardamos el rigidbody:

```
rb2D = GetComponent<Rigidbody2D>();
direction = new Vector3(1, 0); // borrar luego, cuando ya instanciemos prefabs
```

Desarrollo de un videojuego 2D

Agregamos una función para setear la orientación de la bala y la función **FixedUpdate**, en donde modificamos la posición de la bala:

```
private void FixedUpdate()
{
    rb2D.velocity = direction * speed;
}

public void SetDirection(Vector3 aDirection)
{
    direction = aDirection;
}
```

Desarrollo de un videojuego 2D

Si ejecutamos el juego veremos la bala en movimiento. Pero no queremos que la bala aparezca en escena hasta que no se realice un tiro. Entonces creamos un **prefab**.

Para esto creamos una carpeta "**Prefabs**" y arrastramos directamente el objeto hacia ahí.

Al script del personaje, le pasamos ahora el prefab de la bala, para que pueda usarlo cuando se realice un tiro. Para eso creamos una nueva variable publica y arrastramos el prefab visualmente.

```
public GameObject bulletPrefab;
```

Desarrollo de un videojuego 2D

Falta agregar la lógica para que al tocar la **barra espaciadora** salga el tiro. Una nueva función **Shoot** deberá instanciar la bala y setear la dirección según la orientación del personaje.

Las balas que se van generando quedan eternamente como objetos en la escena. Se debería **destruir las balas que ya no se ven** en pantalla.

En **Update**, agregamos:

```
if(Input.GetKeyDown(KeyCode.Space)) {
        Shoot(); }
```

Desarrollo de un videojuego 2D

Agregamos la función **Shoot**:

```
void Shoot()  {
  // seteamos direccion del tiro según la orientacion del personaje
  Vector3 direction;
  if (transform.localScale.x == 1.0f)
        direction = new Vector3(1, 0); // derecha
  else
        direction = new Vector3(-1, 0); // izquierda
  GameObject bullet = Instantiate (bulletPrefab, transform.position + direction * 0.1f,
                                   Quaternion.identity);
  // obtenemos el componente de la bala y le seteamos la direccion del tiro
  bullet.GetComponent<BulletScript>().SetDirection(direction);
}
```

Desarrollo de un videojuego 2D

- **Mejora de la escena y jugabilidad**

La cámara se debe ir moviendo a medida que el personaje lo hace. Creamos un script llamado **CameraScript**, que tendrá una variable pública con el personaje, para poder copiar su posición:

```
public GameObject player; // variable publica
```

En **Update** agregamos el siguiente código y asignamos el script a la cámara, pasándole la instancia del personaje.

```
Vector3 position = transform.position;
position.x = player.transform.position.x;
position.y = player.transform.position.y;
transform.position = position;
```

Desarrollo de un videojuego 2D

También podemos mejorar un poco la estética del juego, creando un background o paisaje de fondo.

Vamos agregando imágenes de fondo desde "**SpriteSheet/bakground**", y las ordenamos seteando la propiedad "**Order in Layer**".

Los componentes con un **valor más grande se verán por delante** de los que tienen un valor menor. Por esto, el personaje principal, el terreno y la bala deben tener un valor más grande que todo lo relativo al background.

Desarrollo de un videojuego 2D

- **Incorporación de música y efectos de sonido**

En la cámara existe un “Audio Listener”, es decir que la cámara escucha el sonido.

Para incorporar música al juego, agregamos a la cámara un “Audio Source” y le asignamos el archivo “**snd_main_theme**”. Como por default esta “**Play on Awake**” empezara a sonar desde el inicio.

Activamos “**Loop**” para que al terminar comience nuevamente.

Desarrollo de un videojuego 2D

Incorporaremos como ejemplo un **efecto de sonido a la bala**.

Al script de la bala agregamos una variable pública Audioclip: public AudioClip sound;

En **Start** accedemos a la cámara para obtener la fuente de audio, y reproducir una vez el audio del tiro:

```
Camera.main.GetComponent<AudioSource>().PlayOneShot(sound);
```

Este sonido **se reproducirá al inicio de cada tiro**, al instanciar el prefab de la bala. Seleccionamos el prefab, y le asignamos el archivo de audio a utilizar (por ejemplo, **snd_rifle_shoot**).

Desarrollo de un videojuego 2D

Y así finaliza este sencillo ejemplo de desarrollo de un juego 2D.

Por supuesto quedan muchas funcionalidades por agregar y analizar, pero al menos se pudieron ver las características más básicas a manejar.