



Santo Domingo Savio (Salesianos)

DESARROLLO DE APLICACIONES MULTIPLATAFORMA

ANIME NO TATAKAI

Autor:

José Luis Carballo Flores



Mayo 2022

Índice general

Índice general	2
Índice de figuras	3
1 Introducción	5
1.1. Resumen	5
1.2. Motivación	6
1.3. Objetivos del Proyecto	6
1.4. Metodología	6
2 Desarrollo	9
2.1. Entornos de desarrollo	9
2.2. <i>Gameplay</i>	16
2.3. Proceso de desarrollo del juego	18
3 Conclusiones	31
3.1. Mejoras y líneas futuras	31
3.2. Reflexión	31
4 Anexos	33
4.1. Diagramas UML	33
4.2. Código / Clases	39
Bibliografía	49

Índice de figuras

1.1.	Videojuegos tipo defensa de torres.	5
2.1.	Unity.	9
2.2.	Ejemplo de un BoxCollider2D.	11
2.3.	Ventana Animation en Unity.	11
2.4.	Ejemplo de la ventana Animator.	12
2.5.	Ejemplo variables públicas.	12
2.6.	Clase MonoBehaviour.	13
2.7.	Ejemplo función OnCollision.	15
2.8.	Ejemplo función OnTriggerEnter.	15
2.9.	GitLab.	16
2.10.	GitKraken.	16
2.11.	TeamDeku.	17
2.12.	TeamNaruto.	17
2.13.	Controles.	18
2.14.	Botones en el campo de juego.	18
2.15.	Función para el movimiento.	19
2.16.	Sprites del mapa y los botones.	19
2.17.	Sprites de los personajes.	20
2.18.	Función para referenciar el Animator de los personajes.	20
2.19.	Sprites de movimiento.	20
2.20.	Separación de calles.	21
2.21.	Temporizador UI.	21
2.22.	Script del Temporizador.	21
2.23.	Cargar escena deseada.	22
2.24.	Salir de la aplicación.	22
2.25.	Menú Principal.	22
2.26.	Función para los combates script de TeamDeku.	23
2.27.	Ejemplo pelea.	23
2.28.	Script Barras de Vida.	23
2.29.	Cambiar animaciones de combate.	24
2.30.	Sprites de ataque.	24
2.31.	Clonación Personajes.	24
2.32.	Instanciar los personajes.	25
2.33.	Instanciar ataque especial.	25
2.34.	Cambios de escena en el menú principal.	26
2.35.	Escena Controls.	26
2.36.	Escena Info.	26
2.37.	Escena Naruto.	27
2.38.	Escena Deku.	27
2.39.	Escena Empate.	27
2.40.	Slider para bloquear el botón selección de personaje.	28
2.41.	Dar valores a los sliders de bloqueo de los personajes.	28
2.42.	Mejora de Velocidad Final de Partida.	29
2.43.	Forma del mapa o terreno de juego.	29

4.1.	Diagrama de Clases General	33
4.2.	Clases Botones	34
4.3.	Diagrama de Clases Team Naruto	34
4.4.	Clases Team Deku	35
4.5.	Clases Temporizador, Spawns, Nexos y Funciones	35
4.6.	Diagrama de Componentes	36
4.7.	Diagrama de Caso de uso	37
4.8.	Diagrama de secuencia-sacar personaje	37
4.9.	Primer Diagrama de Gantt	38
4.10.	Diagrama de Gantt Actualizado	38
4.11.	Clase Botones	39
4.12.	Clase Controlador de Botones	40
4.13.	ClasePersonaje1	41
4.14.	ClasePersonaje2	42
4.15.	Clase Nexo	43
4.16.	Clase Temporizador1	44
4.17.	Clase Temporizador2	45
4.18.	Clase Spawn	46
4.19.	Clase Personaje Especial	47
4.20.	Clase Habilidad Especial	47

Capítulo 1

Introducción

En la actualidad el sector de los videojuegos se halla en continuo auge en nuestra sociedad, tanto a nivel artístico, como económico. Esto es posible gracias al amplio abanico de posibilidades que ofrecen a la hora de entretenerte, pues los juegos están diseñados bajo diferentes temáticas, dificultades escalonadas por edad y diversas modalidades (*shooter*, *arcade*, *PvP*, etc).

1.1. Resumen

Este proyecto tiene como objetivo la implementación de un videojuego en el entorno de desarrollo Unity. El proyecto, con título *Anime No Tatakai*, es un juego digital de estrategia -desarrollado para ser jugado en PC- para dos usuarios que comparten el mismo hardware (teclado), con el fin de que su equipo (Team Naruto o Team Deku) consiga llegar al final del tablero para hacer daño al nexo enemigo (objetivo que se debe destruir para ganar la partida). La batalla llega a su fin, bien cuando la vida de uno de los rivales llega a 0, o bien, cuando el cronómetro marque el final. En este último caso, vence el que menos daño haya recibido.

En efecto, el jugador debe decidir con habilidad qué personajes pone en juego, teniendo en consideración que cada uno de ellos tiene un determinado coste de tiempo, daño, vida y velocidad. Ahí reside la estrategia.

Arcade, defensa de torres es la idea de este proyecto y tomamos como inspiración juegos como *Plants vs Zombies*, desarrollado por la empresa PopCap Games en 2009 y fue elegido para ser uno de los mejores juegos del año [9]. Otro juego en el que nos inspiramos, es *Warlords: Call to Arms*, creado por Ben Olding Games, este fue un aclamado minijuego que tuvo muy buenas críticas [7] y, finalmente el famoso *Space Invaders* diseñado por Toshihiro Nishikado y lanzado al mercado en 1978 [15].



Figura 1.1: Videojuegos tipo defensa de torres.

1.2. Motivación

Para nosotros, los videojuegos siempre han estado en nuestra vida desde una edad muy temprana, por ello nuestra principal motivación residía en experimentar el proceso de desarrollo de un videojuego, ya que esta vez no íbamos a ser meros jugadores, sino los creadores de este.

Otra de las motivaciones principales de este proyecto era asentar los conocimientos que hemos ido adquiriendo a lo largo de dos años del curso Desarrollo de Aplicaciones Multiplataforma, en concreto el modelo de programación informática que organiza el diseño de software en torno a datos u objetos (POO). La programación orientada a objetos es el paradigma dominante en el desarrollo de videojuegos. Llegar a adquirir un conocimiento profundo de un campo tan abstracto como este ha sido posible, precisamente, gracias al presente proyecto.

Por último debemos tener en consideración el creciente auge del mundo del anime, no hay más que ver el apoyo recibido por plataformas digitales de visionado de cine y series online como el popular “Netflix”. Esta situación justifica la selección de la temática que guía nuestro videojuego, y es que pensamos que la animación japonesa es un punto importante para captar consumidores.

1.3. Objetivos del Proyecto

Para realizar el presente trabajo, se han establecido como fin los siguientes objetivos concretos:

- (a) Introducción al entorno de diseño y programación: el primer objetivo consiste en aprender a usar las herramientas que se han seleccionado para el desarrollo (Unity, lenguaje C#...).
- (b) Concreción de la temática y ambientación del juego: en este momento, debíamos seleccionar la temática en la que se iba a ambientar y el tipo de diseños que se iban a incluir.
- (c) Validar funcionalidad del Software con respecto a los objetivos del principio.
- (d) Desarrollo de las mecánicas de los personajes, diseño del nivel y comportamientos en los combates: aquí simplemente debíamos desarrollar el código idóneo para que el juego funcionase.
- (e) Diseño y animación de los personajes: este objetivo está centrado en la interfaz gráfica.
- (f) Elección de la banda sonora: en este caso teníamos que prestar especial atención a que el acompañamiento musical concordase con la ambientación elegida.
- (g) Puesta en común de todos los elementos y subsanación de errores.

1.4. Metodología

En cuanto a la metodología empleada, la decisión consistió en seguir el método iterativo, dividiendo el proyecto en grupos de tareas (iteraciones) que se van ejecutando y probando sucesivamente. El hecho de realizar las pruebas pertinentes tras cada etapa permite controlar errores en una fase temprana del trabajo y facilitar así su depuración.

En este apartado vamos a dar una visión general de las fases seguidas a lo largo de la realización del proyecto, que son las correspondientes a la mayoría de los desarrollos de softwares.

- (a) **Estudio:** en esta parte hemos indagado sobre videojuegos con una finalidad parecida con el objeto de extraer ideas sobre organización e ideas que se llevan a cabo en videojuegos similares.
- (b) **Análisis:** en esta fase teníamos que tener en cuenta los requisitos o restricciones que debía cumplir antes de pasar a la programación. Estos ya venían definidos en el anteproyecto, (las unidades tienen un coste basado en tiempo, cada unidad tiene su velocidad, vida y daño correspondientes a su tiempo de reaparición, los personajes hacen daño al nexo enemigo, no se pueden hacer daño personajes del mismo equipo,etc).

- (c) **Diseño e implementación:** tras finalizar las fases de estudio y análisis y con el listado de requisitos bien definidos, procedemos a diseñar la arquitectura y funcionalidad del videojuego.
- (d) **Pruebas:** como dijimos en la sección de metodología, nuestra estrategia era realizar las pruebas pertinentes en las fases más tempranas para ir comprobando progresivamente la efectividad del videojuego, e implementarlas con otras partes del videojuego y arreglar las pruebas a medida que vamos avanzando con el desarrollo.

Capítulo 2

Desarrollo

2.1. Entornos de desarrollo

Unity Hub



Figura 2.1: Unity.

Unity Hub es una aplicación independiente que agiliza la forma en la que navegas, descargas y administras tus proyectos e instalaciones de Unity.

Puedes usar Unity Hub [16] para: administrar, descargar e instalar las versiones LTS (soporte a largo plazo) o Tech Stream (prelanzamiento) de Unity Editor. Crear y administrar proyectos de Unity con distintas versiones. Buscar plantillas, proyectos de muestra y materiales de aprendizaje para diferentes niveles de habilidad. Administrar tu perfil, preferencias y licencias de Unity.

Versiones de Unity Editor

Las versiones TECH Stream del Unity Editor [20], son Editores con acceso anticipado a nuevas características para futuras versiones del Editor. Se publican dos veces al año, con soporte hasta el próximo lanzamiento, añadiendo actualizaciones y correcciones de errores a la versiones actuales de TECH Stream.

Las Versiones con Soporte a largo plazo se publican después de las versiones TECH Stream, con mejoras en la estabilidad del editor, tienen un soporte de 2 años con actualizaciones y parches cada 15 días, estas actualizaciones no modifican ni añaden nuevas herramientas al Editor, las versiones LTS (Long Time Support) actualmente son:

- Unity Editor 2017.4
- Unity Editor 2018.4
- Unity Editor 2019.4
- Unity Editor 2020.3
- Unity Editor 2021.3

Unity Editor 2020.3.32f1

Empezamos con la versión de Unity Editor 2020.3.26 el 31 de enero 2022, seleccionamos esta versión porque es la versión LTS (soporte a largo plazo) con mayor estabilidad, sin ninguna modificación en las herramientas, solo actualizaciones y parches. Dejamos de actualizar Unity Editor hasta la versión 2020.3.32f1, porque la siguiente versión de Unity Editor 2020.3.33f1 [19], actualizaban la mínima versión de Unity 2D a la versión 2020.3.26f1 y nos daría futuros problemas en animar los personajes o crear los *Prefabs* de los personajes.

Estas son las herramientas y componentes que contiene Unity Editor [17].

- **Frames:**

Unity trabaja tan rápido como le sea posible, entre unos 50 a 60 cuadros por segundo dependiendo de la plataforma.

- **GameObjects:**

Es el concepto más importante del Editor de Unity, a cualquier objeto de Unity se le llama *GameObject* (botones, personajes, imágenes, etc.).

- **Script:**

Los Script son un tipo de componente que agregamos al game object a través del inspector. Por dentro son fragmentos de código que determinan el comportamiento de nuestros objetos.

Todo objeto en un juego es un *GameObject*, desde los personajes y los objetos, luces, cámaras y efectos especiales. Sin embargo, un *GameObject* no puede hacer nada por sí mismo, su funcionalidad se la dan los Scripts que lleve asociados.

- **Transforms:**

Se usa para guardar la posición, rotación y escala, con respecto a los ejes X, Y, Z. Un *GameObject* siempre tendrá el componente Transform, no es posible eliminar el componente o crear un *GameObject* sin este componente.

- **Escenas:**

Las escenas en Unity contienen los objetos (*GameObject*) de nuestro juego. Pueden ser usadas para crear distintos niveles, un menú, una pantalla de Game Over, etc.

- **Gizmos:**

Se utilizan para proporcionar ayudas visuales de depuración o configuración en la vista de escena, por ejemplo una etiqueta visual, o que un personaje tenga un contorno de un color y su enemigo de otro color...

- **Main Camera:**

Es un componente de Unity, el cual nos permite ver el funcionamiento de nuestro juego. Puede haber más de una cámara en la misma escena, y muchas cámaras en nuestro juego, las cuales podemos personalizar y manipular para hacer que la presentación del juego sea realmente única. Tienen unas características principales:

- Fondo. El color aplicado a la pantalla.
- Proyección. Activa o desactiva la capacidad de la cámara para simular perspectiva.
- Tamaño y campo de visión. Lo que queremos ver dentro de nuestro juego.
- Profundidad. Uso de forma organizativa en caso de juego 2D, y forma de diferenciar capas del juego.

- **Canvas:**

Es un tipo de *GameObject*, el cual nos ayuda a posicionar los elementos UI (imágenes, botones, textos...). Utiliza el objeto EventSystem por defecto que se encarga de enviar eventos a objetos, basado en input, sea teclado, ratón, tacto o un input personalizado. Si en nuestro juego creamos una imagen, por defecto si no lo hemos creado antes, se nos creará un canvas automáticamente.

- **Canvas Scaler:**

Es un componente dentro del canvas, el cual controla la escala de los elementos que se encuentran dentro de dicho canvas, y la resolución de la pantalla.

- **RigidBody 2D:**

Hace que un game object esté bajo el control del motor de física, es decir, hace que los objetos se muevan de una manera más realista. Si un GameObject contiene un Rigidbody, este podría ser influenciado por la gravedad, se puede modificar su movimiento, y podría interactuar con otros GameObject que a su vez tengan RigidBody o BoxCollider.

- **Sprite Renderer:**

Este componente genera un imagen y administra cómo aparece visualmente en la escena, los sprites son renderizados para una mejor visión de ellos.

- **Prefabs:**

Un Prefab es un game object con componentes, valores de propiedad, y otros game objects hijos como un asset común. El prefab actúa como plantilla para instanciar en la Escena. Esto se hace cuando queremos instanciar a un game object en la escena más de una vez.

- **Sprites:**

Los Sprites son objetos gráficos 2D, en general son un conjunto de imágenes, con los cuales podemos poner un mapa de fondo para nuestro videojuego, crear el movimiento, ataque, muerte de nuestros personajes, etc.

- **Box Colliders 2D:**

Es un tipo de colisionador que interactua con el administrador de física en 2D. Es una figura cuadrada con una posición establecida, anchura y altura en el espacio del Sprite.

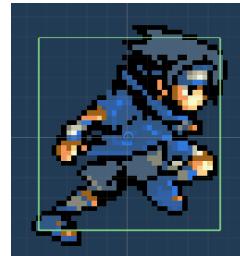


Figura 2.2: Ejemplo de un BoxCollider2D.

- **Animation:**

En este componente se crean los *clips* que vamos a utilizar en nuestro videojuego como ,por ejemplo, el movimiento, la acción de atacar, morir, etc. Cada clip puede ser pensado como una sola grabación lineal. Se deben seleccionar los Sprites que queremos usar y arrastrarlos a esta ventana para darles el efecto deseado.

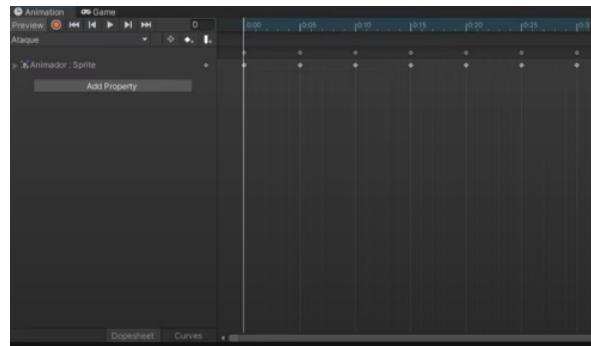


Figura 2.3: Ventana Animation en Unity.

■ **Animator y Animator Controller:**

Este componente es utilizado para asignar una animación a un game object en su escena. Este requiere una referencia a un Animator Controller que define qué clips de animación utilizar, y hacer transiciones entre ellos (esto se hace en los scripts). Dentro de la ventana de Animator en Unity definimos qué clips vamos a utilizar y sus transiciones. Para crear transiciones entre ellos, están los parámetros, que, por ejemplo, podría ser un booleano, con el cual vamos a decidir cuándo pasar de un clip a otro desde el código.

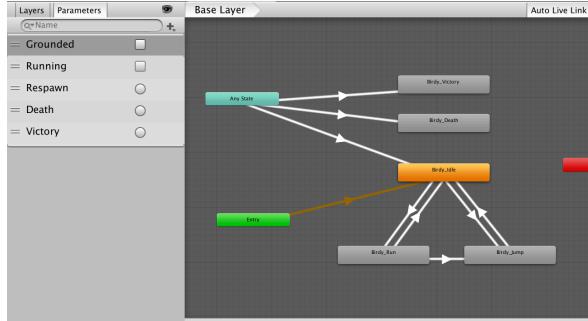


Figura 2.4: Ejemplo de la ventana Animator.

■ **Tags:**

Los tags nos sirven para referenciar uno o mas GameObject desde dentro de los Scripts, con la función *FindGameObjectsWithTag*.

■ **Variables públicas:**

Si una variable es pública, podremos acceder a esta y darle valores desde el inspector de Unity, y no tener que acceder al Script. Si hacemos referencia a algún tipo de GameObject y lo hacemos público, tendremos que arrastrar dicho GameObject al Script para que sepa a qué objeto se refiere.

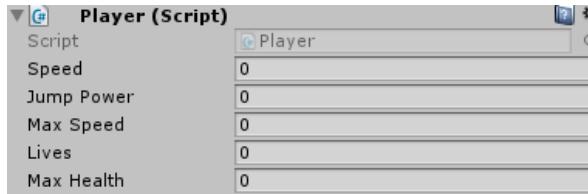


Figura 2.5: Ejemplo variables públicas.

Visual Studio / C#

Librerías utilizadas [18]

■ **UnityEngine:** Dentro de esta librería utilizamos *Time.deltaTime* que nos devuelve cuántos segundos ha tardado en renderizar el último frame, y algunas otras funciones referidas al *Time*.

Utilizamos *Time.timeSinceLevelLoad* para que el tiempo empiece a correr cuando se abra la escena donde está el Script que contenga esta función.

Otras funciones de las que más hemos utilizado han sido *FindObjectOfType* esta función nos devuelve el primer objeto que contenga el Script al que hacemos referencia, y *GetComponents* que nos permite acceder a funciones o variables de un objeto en concreto.

Otra función muy importante utilizada en nuestro proyecto ha sido *Instantiate*, esta función recibe como parámetros el game object que queremos instanciar y la posición donde queremos instanciarlo.

Por último utilizamos la función *FindGameObjectsWithTag*, esta función nos sirve para hacer referencia a un objeto en concreto dentro de nuestros Scripts por su tag añadido previamente desde el inspector.

- **UnityEngine.UI**: Esta librería nos sirve para utilizar elementos UI (Interfaz de Usuario) como pueden ser los botones, Slider, imágenes, y acceder a sus propiedades.

- **UnityEngine.SceneManagement**: Gracias a esta librería podemos acceder a las propiedades de las escenas de nuestro juego, como puede ser cambiar entre escenas, empezar un cronómetro justo cuando cargue esa escena, buscar una escena por su tag, etc.

Clase MonoBehaviour

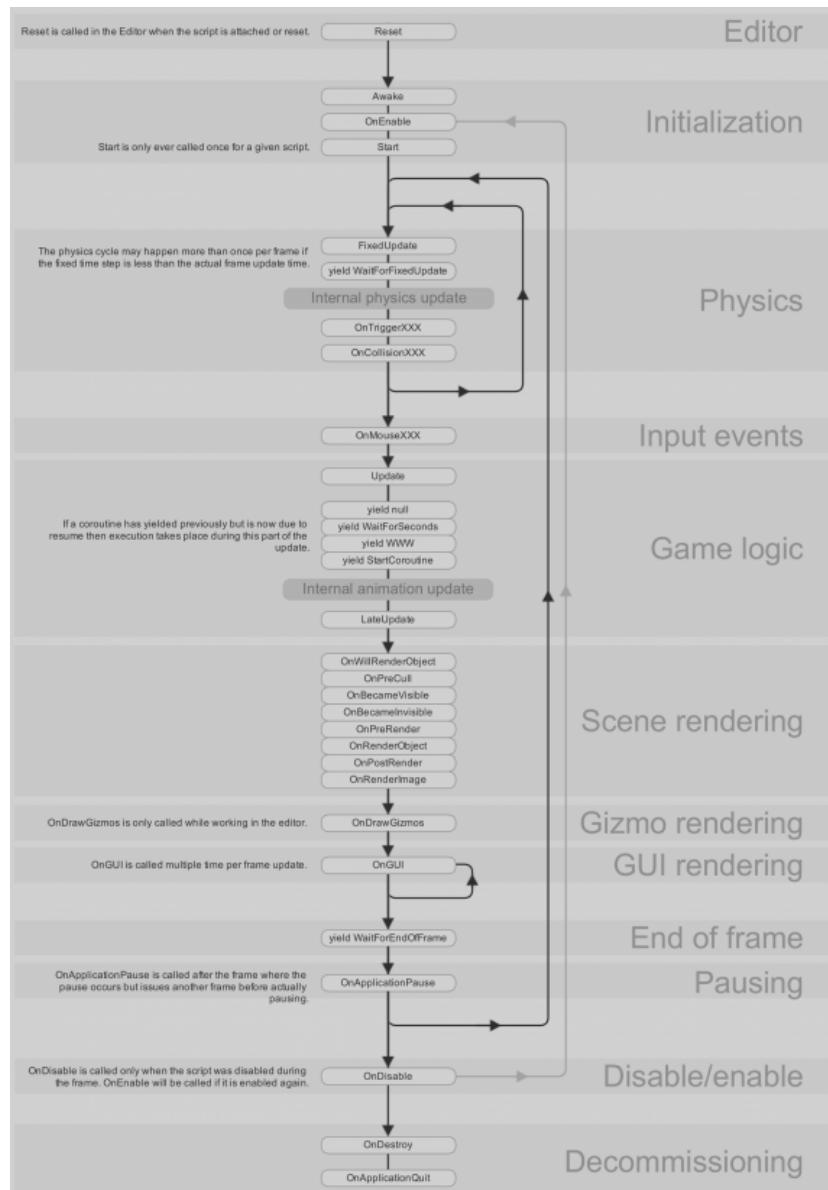


Figura 2.6: Clase MonoBehaviour.

En Unity, las clases heredan de la clase padre *MonoBehaviour*, y heredan de esta todas sus propiedades, esto se hace porque la clase *MonoBehaviour* posee variables, objetos y métodos que están definidos por Unity y que muchas de ellas son necesarias para la programación de videojuegos en Unity.

Algunos de los métodos más usados son:

- **Awake:** Se usa para inicializar variables o el estado del juego antes de que este comience, es utilizado para obtener todas las referencias que se hacen a otros objetos desde ese Script.
- **Start:** Se llama justo después del Awake, para diferenciar entre *Awake* y *Start*, pondremos el ejemplo en el que estamos realizando un juego en el que nuestro personaje tiene un arma, en la función *Awake* pondremos la munición que contiene el arma, y en la función *Start* habilitaremos que pueda disparar. Se utiliza para inicializar algunas variables.
- **OnEnable:** Esta función se llama cuando el objeto se habilita y se activa.
- **OnDisable:** Se llama a esta función cuando el objeto deja de estar disponible o inactivo.
- **Update:** Se ejecuta una vez por cada frame, pero si nuestro dispositivo en el que está corriendo nuestro videojuego, no puede con tanta carga y los frames empiezan a bajar, el update se llama con mucha menos frecuencia.
- **FixedUpdate:** Se ejecuta constantemente y puede ejecutarse más de una vez por cada frame, exactamente se llama a esta función cada 0.02 segundos, normalmente es usado cuando queremos usar físicas en nuestro juego, para que haya más fluidez.
- **OnTriggerXXX:** El método *OnTriggerXXX* nos permite detectar cuándo dos game objects solapan sus Colliders, de esa forma podremos aplicar todas las acciones que sean necesarias.

Para que ocurra la ejecución de *OnTriggerXXX* deben darse varias condiciones:

La primera es que deben haber dos *GameObjects* con Colliders involucrados, uno de los Colliders debe estar en modo Trigger, para esto se debe pulsar la casilla «Is Trigger» en el inspector.

Además al menos uno de los *GameObjects* involucrados debe tener asignado un componente RigidBody (que también se asigna en el inspector), porque *OnTrigger* está relacionado a la parte física del motor.

Hay tres tipos de funciones Trigger:

1. **OnTriggerEnter** Se usa cuando queremos que nuestros objetos interactúen nada más tocarse, por ejemplo si una bala impacta en otro personaje, esta le infligirá daño al momento.
 2. **OnTriggerStay** Se usa cuando queremos que nuestros objetos estén interactuando en cada momento, por ejemplo, si caemos en unas zarzas, estas nos harán daño constantemente.
 3. **OnTriggerExit** Se usa cuando queremos que una funcionalidad en concreto se dé al dejar de tocarse dichos objetos, por ejemplo, queremos que al atravesar una puerta aparezca un monstruo, pues esto no ocurrirá hasta que los dos objetos dejen de tocarse.
- **OnCollisionXXX:** Es quizá uno de los métodos predefinidos más útiles. A este método se le llama cada vez que el objeto que lleva este script colisiona con otro objeto de la escena, este se usa cuando el Collider o Rigidbody entra en contacto con otro Collider o Rigidbody.

Al igual que los Trigger, uno de los dos objetos tiene que tener un componente Rigibody.

Hay tres tipos de funciones Collision:

1. **OnCollisionEnter** Se usa cuando queremos que nuestros objetos interactúen nada más tocarse, por ejemplo, si en el juego tenemos una valla electrificada que hace daño, esta nos infligirá daño nada más tocarla, y la collision cumplirá su función y no nos dejará pasar.
2. **OnCollisionStay** Se llama una vez por frame para cada colisionador/cuerpo rígido que está en contacto con el cuerpo rígido/collisionador.
3. **OnCollisionExit** Se usa cuando este cuerpo rígido/collisionador ha dejado de tocar otro cuerpo rígido/collisionador.

- **Diferencias entre *OnCollisionXXX* y *OnTriggerXXX*:** Una diferencia entre estas dos funciones reside en los parámetros que reciben. Si la función es *OnCollision*, esta recibirá un objeto *Collision*, este tiene la información sobre el impacto, como la velocidad y los puntos de impacto, si es la función *OnTrigger*, esta recibirá por parámetros el objeto *Collider*, son áreas que se le añaden a un *GameObject* que es la forma en la que Unity detecta las colisiones.

```
void OnCollisionEnter(Collision collision)
{
    GameObject objeto1 = GameObject.Find("objeto1");
    if (collision.gameObject.name == "objeto2")
    {
        Debug.Log("objeto1 ha colisionado con objeto2");
    }
}
```

Figura 2.7: Ejemplo función OnCollision.

```
void OnTriggerEnter(Collider collider)
{
    GameObject objeto1 = GameObject.Find("objeto1");
    if (collider.gameObject.name == "objeto2")
    {
        Debug.Log("objeto1 ha entrado en área de objeto2");
    }
}
```

Figura 2.8: Ejemplo función OnTrigger.

Si queremos lograr el efecto de colisión real de dos cuerpos rígidos, usaremos *OnCollisionEnter* y el motor Unity manejará automáticamente el efecto de las colisiones de cuerpos rígidos.

La gran diferencia entre un Trigger y un Collision la encontramos, concretamente, en el inspector: si el campo de Trigger está marcado, entonces el objeto no tendrá colisión, en caso contrario, los objetos tendrán cuerpo físico y por tanto podrán chocarse entre sí. Para mayor claridad de esta cuestión, pensemos en un juego de automóviles donde es posible la colisión, en este caso concreto, la casilla Trigger no debería estar seleccionada, ya que si esto ocurriese se atravesarían.

- ***OnMouseXXX*:** Esta función nos permite manejar operaciones simples o lógicas basadas en las entradas del mouse. Estas se ejecutan en un objeto de juego con Collider y un elemento de interfaz de usuario, que tiene el script adjunto con los métodos de eventos del mouse.

Hay varios tipos de funciones *OnMouse*:

1. ***OnMouseEnter***: a esta función se la llama tan pronto como el puntero del mouse ingresa al área delimitada de un *GameObject*.
2. ***OnMouseExit***: se la llama tan pronto como el puntero del mouse sale del área delimitada de un *GameObject*.
3. ***OnMouseDown***: cumple su función tan pronto como se presiona el botón izquierdo del mouse en el área delimitada de un *GameObject*.
4. ***OnMouseUp***: ejecuta la acción tan pronto como se suelta el botón izquierdo del mouse en el mismo game object o en cualquier otro lugar de la pantalla.

- ***Coroutines*:** Una corutina es un método que puede pausar la ejecución durante un tiempo, y luego continuar esta ejecución desde donde acabó. Dentro de las corrutinas está la función *Yield WaitForSeconds* que sirve para esperar tantos segundos como le digamos antes de seguir la corutina.

- ***OnDestroy*:** Se utilizará para destruir el objeto al que se haga referencia dentro de esta función.

Gitlab



Figura 2.9: GitLab.

Es una plataforma online pensada para el desarrollo colaborativo que hace posible la creación de repositorios para el control de versiones de un proyecto [10]. Varias personas pueden trabajar sobre el mismo proyecto sin sobrescribir el trabajo de otros y mantener versiones anteriores del código.

Este es el repositorio donde se encuentra nuestro proyecto *Anime no Tatakai*. <https://gitlab.com/animetatakai/general-proyecto/>

Gitkraken



Figura 2.10: GitKraken.

Es una herramienta de escritorio que administra distintos clientes que usan Git de forma visual o por linea de comandos [3], accesible y con una interfaz intuitiva, con flexibilidad entre distintos Sistemas operativos.

Esta herramienta hace integraciones con GitHub, GitLab, Bitbucket, y Azure DevOps.

Piskel

Piskel, es un editor Online muy completo para crear Sprites en 2D.

Alferd Spritesheet Unpacker (ASU)

Es una aplicación de Windows, cuya función es separar imágenes dentro de una hoja de Sprites de juegos, ya que normalmente en estas vienen muchas imágenes de distintas acciones como correr, pelear, saltar... Si queremos alguna en concreto esta aplicación es la idónea para separarlas.

2.2. *Gameplay*

El juego que hemos desarrollado se adscribe a la tipología llamada *Tower Defense*, esto es, defensa de torres o videojuegos de defensa, ya que el objetivo es defender “el territorio” de un jugador frente a los atacantes enemigos. La protección de las posesiones se logra gracias a la colocación de estructuras o contrincantes en la trayectoria de ataque del enemigo. La selección y la ubicación de los elementos defensivos es una estrategia esencial del género. Pues bien, nuestro proyecto, titulado *Anime No Tatakai*,

sigue esta senda: en él existen dos equipos *Team Naruto* y *Team Deku*, que defienden sus territorios, que son las columnas desde donde parten los personajes. En este trayecto se encontrarán con olas de enemigos las cuales buscan destruir o dañar a fin de llegar al campamento del oponente para hacer daño al nexo. El juego comienza con 100 puntos de vida para cada jugador.



Figura 2.11: TeamDeku.



Figura 2.12: TeamNaruto.

En cuanto al campo de juego, este estará constituido por un tablero de 5 filas, donde situaremos (a nuestra elección) las unidades que hayamos seleccionado previamente. Estas filas constituyen 5 posibles caminos para llegar a territorio enemigo.

Con respecto a los personajes, hay que tener en cuenta que poseen diferentes daños, vidas, velocidades de movimiento y costo, que es equivalente a un porcentaje de tiempo necesario para que salgan a combatir. Ahí reside la estrategia: una unidad puede tener un daño grande, pero el tiempo de espera hasta poder jugar otro personaje es muy dilatado, lo que podría afectar negativamente a la vida de tu nexo, si tu oponente pone en juego personajes de menos costo y menos daño, pero con mayor velocidad de movimiento. Tenemos que tener en cuenta que el tipo de peleas desarrolladas son cuerpo a cuerpo, a excepción de Deku y Naruto que lanzan sus habilidades a distancia a través de un ataque definitivo que consigue vencer a todos los oponentes de su misma fila y hace daño al nexo enemigo. Algunos ejemplos de combate serían los siguientes:

- Sasuke que tiene 1 segundo de cose, 2 de daño y 8 de vida, se enfrenta a Tokoyami que tiene 2 segundos de coste, 4 de daño y 10 de vida, en esta batalla saldrá victorioso Tokoyami, y proseguirá su camino hacia el nexo enemigo, pero con 6 de vida puesto que habrá recibido dos ataques de Sasuke antes de que este muera (ya que los personajes reciben daño cada segundo). Si nadie más se interpone en su camino hará 4 de daño al nexo enemigo.
- Deku que tiene un ataque definitivo y un coste de tiempo de 5 segundos, se enfrenta en su misma línea a tres Kakashi (6 de daño, 15 de vida y 3 segundos de coste). En este combate saldría victorioso Deku porque su ataque definitivo vence a todas las unidades enemigas independientemente de su número, sin embargo, Deku moriría inmediatamente, pero su ataque proseguiría su camino hasta chocar con el nexo enemigo e infiigirle su daño correspondiente (5 de daño). La diferencia entre los personajes especiales (Deku y Naruto) y el resto es que su daño siempre llegará al nexo enemigo.

Personaje	Daño	Vida	Coste Segundos	Velocidad de movimiento	Habilidad Especial
Sasuke y AllMight	2	8	1	2.5	Combate cuerpo a cuerpo
Gaara y Tokoyami	4	10	2	2	Combate cuerpo a cuerpo
Kakashi y Bakugo	6	15	3	1.5	Combate cuerpo a cuerpo
Naruto y Deku	X	X	5	3.5	Combate a distancia

Como dijimos anteriormente, este juego digital está pensado para PC y para ser jugado desde el mismo teclado. El jugador situado a la derecha de este hardware manejaría 5 botones (las 4 flechas de dirección y el botón Enter o Intro) y su oponente empleará otros 5 (W, A, S, D y la barra espaciadora).



Figura 2.13: Controles.

2.3. Proceso de desarrollo del juego

1. Botones en el campo de juego

En primer lugar, para crear el tablero en la escena donde se va a desarrollar la partida tenemos que crear un canvas en el inspector de Unity, ya que esto hace más fácil posicionar los elementos UI de nuestro videojuego.

Creamos los botones dentro de dicho canvas y los posicionamos en la escena de Unity con sus respectivos tamaños y posiciones.

Cada botón contiene el mismo Script, cuya función es que cuando el botón no esté seleccionado su color será blanco, y cuando este se seleccione cambiará su color a verde.

A continuación creamos los scripts que vamos a usar para cada agrupación de botones y se los asignamos al canvas. Dicho canvas va a tener 4 scripts:

- ControladorBotonesRightLeft
- ControladorBotonesUpDown
- ControladorLetrasAD
- ControladorLetrasWS

Estos Scripts cumplen la misma función. Los script ControladorBotonesRightLeft y ControladorLetrasAD se mueven por la fila de elección de personaje, mientras que los script ControladorBotonesUpDown y ControladorLetrasWS se desplazan por las columnas laterales donde se posicionarán los personajes seleccionados.

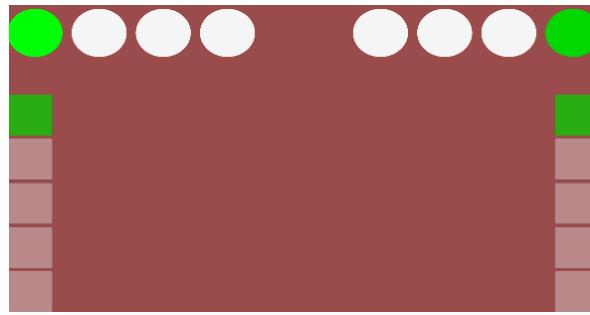


Figura 2.14: Botones en el campo de juego.

2. Movimiento de un *GameObject*

Todos los personajes que conforman el juego se moverán en el eje X [11], pero en direcciones opuestas a fin de que se produzca una colisión y tengan lugar las batallas. Para lograr este propósito hemos creado un game object al que hemos asignado un script que aportará distintas funcionalidades. En cuanto al movimiento le hemos asignado una variable de velocidad, de modo que cuanto mayor sea el valor más rápido se moverá nuestro personaje.

Continuamos accediendo al *Transform* del *GameObject* que contenga este Script, para que sepa a cada momento en qué posición se encuentra con respecto a los ejes X e Y, una vez tengamos el *Trasform*, utilizamos la función *Translate*, para cambiar la posición de nuestro personaje, esta recibe tres parámetros que son los ejes X, Y, Z. Todo esto va a estar dentro de la función *Update* para que cambie dicha posición a cada frame por segundo.

```
gameObject.transform.Translate(-velocidad * Time.deltaTime, 0, 0);
```

Figura 2.15: Función para el movimiento.

Como sólo queremos que nuestro personaje cambie su posición en el eje X, sólo cambiaremos el primer parámetro, y dependiendo de que equipo sea, pues pondremos la velocidad en valor negativo para que vaya hacia la izquierda, o en valor positivo para que vaya hacia la derecha, y dicho valor lo multiplicaremos por *Time.deltaTime*, para que la velocidad de renderización de nuestro ordenador no influya en nuestro juego

3. Sprites para el campo de juego y botones

Para los botones buscamos imágenes de la cara de los personajes que queríamos en nuestro juego, importábamos estas fotos en Unity, y las añadimos en el componente *Image* del botón, y desde el inspector arrastramos las imágenes a dichos botones.

Para el mapa, utilizamos la aplicación online Piskel, con la cual hicimos el mapa desde 0, adecuándonos al tamaño de los botones de nuestro juego.

En nuestra escena creamos un *GameObject*, le añadimos a este el componente *Sprite Renderer*, arrastramos nuestra imagen a este componente, y por último mandamos nuestro mapa al final de las capas, para que los demás *GameObject* se vean y esta imagen no los tape, esto se hace dentro del componente *Sprite Renderer* en el campo *Order in Layer* (Ordenar en capa), y le asignamos el valor -1.



Figura 2.16: Sprites del mapa y los botones.

4. Sprites de los personajes

Para los personajes buscamos Sprites por internet que se adecuaran a lo que estábamos buscando, para separar estas imágenes utilizamos la aplicación de escritorio Alferd Spritesheet Unpacker (ASU) [4]. Una vez separadas las importamos a Unity para poder trabajar con ellas y tener un *GameObject* más visual .

Para asignar los Sprites, lo hacemos desde el componente de *Sprite Renderer* y le añadimos el Sprite que queremos usar para cada personaje, y por último adecuamos su escala para que todos tengan el mismo tamaño.



Figura 2.17: Sprites de los personajes.

5. Animaciones de movimiento de los personajes

Para crear las animaciones de nuestros personajes, necesitamos usar la ventana de *Animation* en Unity, desde ahí creamos el clip de movimiento de cada personaje, añadiendo las imágenes que necesitemos, y haciendo pruebas para que el movimiento de nuestro personaje se adecue a lo que estamos buscando. El primer clip que se hace en esta ventana, será el clip por defecto que tendrá nuestro personaje nada más aparecer en la escena, y como en nuestro videojuego el objetivo es llegar al final de la pantalla para hacer daño al nexo enemigo no tenemos animación de estar quieto (idle). En el código debemos hacer referencia a estas animaciones desde la función *Awake*, para que nada mas empezar el juego ya estén cargadas las animaciones de todos nuestros personajes [13].

```
private Animator animaciones;
// Unity Message | 0 references
private void Awake()
{
    animaciones = GetComponent<Animator>();
}
```

Figura 2.18: Función para referenciar el Animator de los personajes.



Figura 2.19: Sprites de movimiento.

6. Boxcollider 2D y RigidBody a los personajes y a partes del tablero

En este proceso tuvimos que añadir un *BoxCollider2D* y un *Rigidbody* a los personajes de nuestro videojuego, ya que esto era necesario para que se produzca el choque entre ellos. Además fue necesario adecuar los collider de cada personaje para que tuvieran una escala parecida todos y al colisionar permanecieran en la misma línea sin sobreponerse entre ellos. Para que los personajes no chocaran contra los de otras filas añadimos boxColliders como separadores de calles.

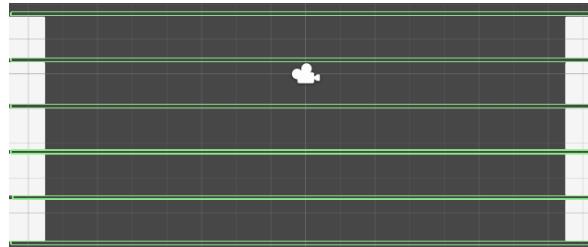


Figura 2.20: Separación de calles.

7. Temporizador de la partida

Decidimos que las partidas tuvieran un tiempo limitado (3 minutos), así que añadimos un temporizador. Inicialmente necesitamos poder ver el tiempo restante con un texto y un *slider* que se pondrá en el *canvas* [8].



Figura 2.21: Temporizador UI.

Para dar los valores al texto y al slider creamos un script en el que el cronómetro comienza la cuenta atrás en el momento en el que se inicia la escena de los combates. Para que el tiempo se empiece a restar cuando se inicia dicha escena utilizamos *Time.timeSinceLevelLoad*. Esta función solucionó la complicación inicial de que el tiempo comenzara nada más abrir el videojuego.

```
private void mostrarTiempo()
{
    tiempo = tiempoJuego - Time.timeSinceLevelLoad;
    int minutos = Mathf.FloorToInt(tiempo / 60);
    int segundos = Mathf.FloorToInt(tiempo % 60);

    string textTime = string.Format("{00:00}:{01:00}", minutos, segundos);
}

if (pararTiempo == false)
{
    textoTiempo.text = textTime;
    temporizador.value = tiempo;
}
```

Figura 2.22: Script del Temporizador.

8. Menú principal (Botones y cambios de escena)

Necesitamos un menú principal que sera la escena inicial desde la que el jugador se encuentre al cargar el juego [6]. Este menú tendrá varios botones que al ser seleccionados que iniciará un evento *onClick()* que ejecutará el script para cambiar a las distintas escenas del juego:

- Controls
- Info
- Play
- Exit

Dentro del script, cambiará la escena a la deseada gracias a la función *SceneManager.LoadScene*, en la que recibe como parámetros el nombre de la escena a la que queremos ir, o para salir con la función *Application.Quit*, dependiendo qué botón seleccione el jugador.

```
SceneManager.LoadScene("SampleScene");
```

Figura 2.23: Cargar escena deseada.

```
Application.Quit();
```

Figura 2.24: Salir de la aplicación.



Figura 2.25: Menú Principal.

9. Combates entre equipos

Para los combates cada personaje tiene asignado un daño y una vida, que son sus atributos para luchar. Estas características vienen definidas en los script de cada una de las unidades. La función *OnCollisionStay* maneja las colisiones, en ella comparamos el Tag del game object el cual lleva un *Collision2D* con los nombres de nuestros enemigos, de este modo no interactuarán con nuestros aliados y sólo harán daño a los rivales. Una vez se haya producido la colisión entre los collider de los dos personajes, utilizaremos la función *GetComponent* para acceder a la función *recibirDaño* de ese personaje contra el que hemos chocado, y le pasaremos como parámetro nuestro daño, de igual manera el enemigo accederá a nuestra función de *recibirDaño*, y recibiremos como parámetros el daño del enemigo para que se reste a nuestra vida. Cuando la vida de estos personajes llegue a 0 morirán por la función *Destroy*.

Todos estos pasos se irán verificando continuamente gracias a la función *FixedUpdate*.

En esta fase del desarrollo nos encontramos con varias dificultades que conseguimos solventar. La primera de ellas se debía a la función *OnCollisionEnter*, que provocaba que los game object al colisionar solo se hicieran daño una vez. Nuestro objetivo era que los personajes recibieran/causaran daño durante toda la franja de tiempo que estuvieran en contacto [5]. Para que esto sucediera sustituimos esta función por *OnCollisionStay*, sin embargo, se hacían daño demasiado rápido ya

que esto al estar dentro de la función *FixedUpdate* se verificaba a cada frame, de modo que no se podía disfrutar de la escena porque duraba demasiado poco. Al final añadimos otro condicional en el que tenía que pasar un segundo entre ataque y ataque para que las batallas duraran más tiempo.

Otra dificultad encontrada fue que los daños y la vida de los personajes se sumaban debido a que había un script general para todos los personajes y lo solucionamos creando un script único para cada personaje.

```
void OnCollisionStay2D(Collision2D other)
{
    if (other.gameObject.CompareTag("Sasuke") && tiempoSiguienteAtaque <= 0)
    {
        animaciones.SetTrigger("Atacar");
        other.collider.GetComponent<Sasuke>().recibirDaño(dañoTeamDeku);
        tiempoSiguienteAtaque = tiempoEntreAtaques;
    }

    if (other.gameObject.CompareTag("Gaara") && tiempoSiguienteAtaque <= 0)
    {
        animaciones.SetTrigger("Atacar");
        other.collider.GetComponent<Gaara>().recibirDaño(dañoTeamDeku);
        tiempoSiguienteAtaque = tiempoEntreAtaques;
    }

    if (other.gameObject.CompareTag("Kakashi") && tiempoSiguienteAtaque <= 0)
    {
        animaciones.SetTrigger("Atacar");
        other.collider.GetComponent<Kakashi>().recibirDaño(dañoTeamDeku);
        tiempoSiguienteAtaque = tiempoEntreAtaques;
    }
}
```

Figura 2.26: Función para los combates script de TeamDeku.



Figura 2.27: Ejemplo pelea.

10. Barras de vida por equipo

En este proceso, añadimos un *boxCollider2D* en representación de un “nexo”, para que cuando los colliders de nuestros personajes chocaran contra él se restara a la vida del nexo el daño recibido de los enemigos. Este proceso es muy similar al de los combates, ya que todo tiene que estar dentro de la función *OnCollisionEnter*, los enemigos cumplirán la función de hacer daño al nexo y morirán gracias a la función *Destroy* [14]. La barra de vida, la creamos con unos Sliders, en los que accedemos por el código de un script asignándoles los valores que deseemos, y restándole el daño de los personajes.

```
void OnCollisionEnter2D(Collision2D other)
{
    if (other.gameObject.CompareTag("AllMight"))
    {
        float dañoAllMight = other.collider.GetComponent<AllMight>().dañoTeamDeku;
        vidas -= dañoAllMight;
    }

    if (other.gameObject.CompareTag("Bakugo"))
    {
        float dañoBakugo = other.collider.GetComponent<Bakugo>().dañoTeamDeku;
        vidas -= dañoBakugo;
    }

    if (other.gameObject.CompareTag("Tokoyami"))
    {
        float dañoTokoyami = other.collider.GetComponent<Tokoyami>().dañoTeamDeku;
        vidas -= dañoTokoyami;
    }

    BarraVida.value = vidas;
}
```

Figura 2.28: Script Barras de Vida.

11. Animaciones de ataque de los personajes

Crear las animaciones de combate de nuestros personajes es un proceso similar al diseño de las animaciones de movimiento que hemos explicado anteriormente. En primer lugar introducimos las imágenes necesarias en nuestro proyecto y desde la ventana *Animation* creamos los clips de ataque de cada personaje. A continuación añadimos las imágenes al clip, haciendo pruebas para que el ataque se adecúe a lo que estamos buscando. En siguiente lugar, accedemos a la ventana *Animator* y en esta unimos las animaciones para que pasen de estar en movimiento a estar en modo ataque desde el código, accediendo a los parámetros definidos en la pestaña de *Animator*, estos pueden ser booleanos, triggers, etc. Finalmente, cambiamos la animación de atacar en el momento de los combates.

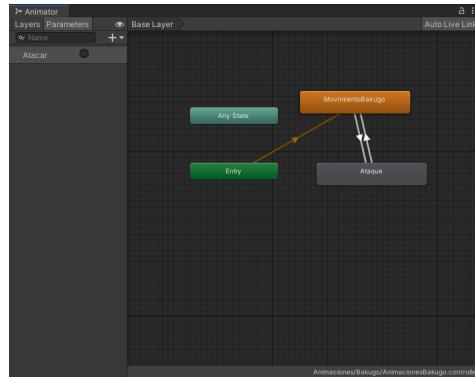


Figura 2.29: Cambiar animaciones de combate.



Figura 2.30: Sprites de ataque.

12. Prefabs de los personajes

Es imprescindible para nuestro juego crear prefabs [1] de cada personaje, ya que estos van a ser instanciados muchas veces y de no hacerlo puede haber errores de ejecución y de rendimiento. Cada clon del personaje tendrá sus características originales, sin que influyan los combates de sus otros clones. Estos se crean desde el inspector de Unity arrastrando nuestro objeto a la parte de los assets donde se encuentran nuestras carpetas (Escenas, Imágenes, etc.).

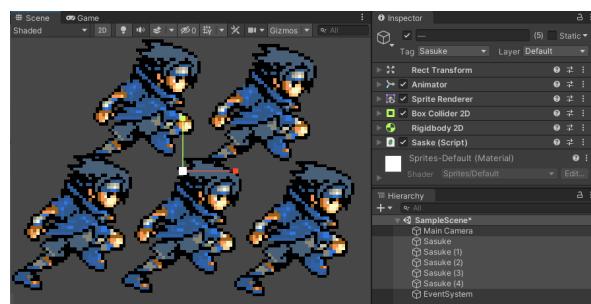


Figura 2.31: Clonación Personajes.

13. Spawn de los personajes en el campo de juego

En el juego se eligen los personajes y en qué carril van a aparecer, para poder hacer la selección de personaje y carril, se van a utilizar los botones que se han posicionado previamente en la escena de Unity 2.15 [2].

Ya que tenemos la parte visual para que los jugadores sepan qué personaje y carril han marcado, creamos dos scripts para cada equipo, a fin de que aparezcan los personajes donde se ha marcado en los botones de los carriles y los personajes seleccionados:

- SpawnNaruto
- SpawnDeku

Para poder instanciar los personajes en el campo de juego, se necesitan dos listas que son definidas en el código de cada script y desde el inspector de Unity le asignamos los valores. Una de las listas contiene los *Prefabs* de los personajes del equipo y la otra lista contiene los *Transforms* de los botones de selección de carril para que aparezcan los *Prefabs* en las coordenadas X, Y, Z de la escena.

La función que se encarga de instanciar los personajes seleccionados en la posición del carril que queremos es *Instantiate*, la cual recibe como parámetros, en primer lugar, la unidad que queremos instanciar y, en segundo lugar, la posición donde queremos que esto ocurra.

```
Instantiate(Prefabs[posicionBotonPersonaje], Carril[posicionBotonCarril]);
```

Figura 2.32: Instanciar los personajes.

14. Personajes especiales Naruto y Deku y sus ataques finales

Para hacer los personajes especiales, creamos los game object de Naruto y Deku y sus ataques especiales [12], y los transformamos en *Prefabs*. Las habilidades tendrán asignado un *Rigidbody* y un *BoxCollider2D* para los combates y hacer daño al nexo enemigo. Para las animaciones de estos es el mismo proceso que en las animaciones de movimiento y de ataque del resto de unidades de nuestro videojuego, a diferencia de que estos solo van a tener la animación de ataque, ya que no se van a desplazar por el tablero, sino que ejecutarán sus movimientos desde la casilla de spawn. Con respecto al código, creamos un script para el personaje, y otro para su ataque especial, el funcionamiento de los scripts de los personajes (Naruto y Deku) es instanciar su ataque especial en la misma posición donde aparezca dicho personaje. Esta función estará dentro de *Start* para que se cumpla nada más instanciarse el personaje, también creamos una corutina de muerte a los 5 segundos, con el objetivo de que este muera nada más lanzar su habilidad. Con respecto a los scripts de los ataques especiales, su funcionamiento es moverse hacia el nexo enemigo para hacerle daño y matar a todos los personajes que se encuentre en su camino. En los scripts de cada personaje que no sean los especiales (Deku o Naruto) se implementa que si el collider con el que se chocan es el del ataque especial mueran por la función *Destroy*.

```
void Start()
{
    Instantiate(rasengan, posicionNaruto);
    StartCoroutine(muerte());
}

// referencias
IEnumerator muerte()
{
    yield return new WaitForSeconds(5);
    Destroy(gameObject);
}
```

Figura 2.33: Instanciar ataque especial.

15. Escenas de juego y cambios de escena

En el punto 8 diseñamos el menú principal, ahora, en esta fase del trabajo dimos funcionalidad a los botones que faltaban (Info, Controls) para mostrar sus escenas pertinentes. Ello lo conseguimos gracias a la función *LoadScene*. En esta sección hemos añadido una nueva función, cuyo objetivo es salir del juego si pulsas 3 veces la tecla F1 desde la escena de las batallas.

```
public void controles()
{
    SceneManager.LoadScene("Controles");
}

0 referencias
public void info()
{
    SceneManager.LoadScene("Información");
}
```

Figura 2.34: Cambios de escena en el menú principal.

■ Controls

En esta escena se encuentran los controles del juego de ambos jugadores, como los personajes de ambos equipos con sus respectivas características.



Figura 2.35: Escena Controls.

■ Info

En esta escena se detalla información sobre el juego.



Figura 2.36: Escena Info.

También creamos tres nuevas escenas que muestran si ha ganado TeamNaruto, TeamDeku, o, por el contrario, hay un empate. Para verificar la victoria de uno de los jugadores accedemos a la vida de los nexos por la función *FindObjectOfType*. Una vez obtenidas las comparamos y aparecerá una escena u otra dependiendo del resultado obtenido.

- Gana TeamNaruto



Figura 2.37: Escena Naruto.

- Gana TeamDeku



Figura 2.38: Escena Deku.

- Empate



Figura 2.39: Escena Empate.

16. Tiempo de Spawn de cada personaje

Para que no haya un problema de tener una cantidad elevada de personajes en el campo de batalla y ralentice el rendimiento de nuestro juego, se necesita poner un tiempo de bloqueo a los personajes y cancelar que el jugador pueda seleccionar otro personaje hasta que no termine el tiempo de bloqueo. Estos serían los tiempos de espera de cada unidad:

Personaje	Coste Segundos
Sasuke y AllMight	1
Gaara y Tokoyami	2
Kakashi y Bakugo	3
Naruto y Deku	5

Para impedir al jugador seleccionar personajes, aparecerán los bloqueos de forma visual, se crea un *slider*, se coloca delante de los botones de selección de personajes y se le dará el valor de tiempo bloqueo del personaje desde el código.



Figura 2.40: Slider para bloquear el boteón selección de personaje.

Anádimos una función al script que instancia a los personajes en el videojuego, esta función da valor al bloqueo de los personajes y dependiendo que personaje se seleccione, se dará un valor específico y ese tiempo de bloqueo impedirá al jugador sacar cualquier otra unidad hasta que el tiempo llegue a 0. Damos valor máximo y valor inicial a todos los slider del equipo.

```
public static void darMaxValue(float tiempo, Slider slider1, Slider slider2, Slider slider3, Slider slider4)
{
    slider1.setMaxValue(tiempo);
    slider2.setMaxValue(tiempo);
    slider3.setMaxValue(tiempo);
    slider4.setMaxValue(tiempo);
}

8 references
public static void darValue(float tiempo, Slider slider1, Slider slider2, Slider slider3, Slider slider4)
{
    slider1.setValue(tiempo);
    slider2.setValue(tiempo);
    slider3.setValue(tiempo);
    slider4.setValue(tiempo);
}
```

Figura 2.41: Dar valores a los sliders de bloqueo de los personajes.

17. Últimas mejoras

Una mejora en el videojuego consiste en aumentar la velocidad de los personajes durante los últimos 20 segundos de la partida. Para conseguir este propósito, creamos una función en cada uno de los scripts de los personajes, en la cual accedemos al tiempo del script de Temporizador y cuando el tiempo sea menor a los 20 segundos comience a aumentarse la velocidad de los personajes. Esta función iría dentro de *FixedUpdate* para que lo vaya verificando a cada momento.

Personaje	Mejora de Velocidad Incremental
Sasuke y AllMight	0.03
Gaara y Tokoyami	0.02
Kakashi y Bakugo	0.01

A grandes rasgos también llevamos a cabo mejoras visuales y mejoramos el código para que sea más comprensible.

```
void mejorarVelocidad()
{
    float tiempoMejoras;
    tiempoMejoras = FindObjectOfType<Temporizador>().tiempo;
    if (tiempoMejoras < 20)
        velocidad += 0.003f;
}
```

Figura 2.42: Mejora de Velocidad Final de Partida.



Figura 2.43: Forma del mapa o terreno de juego.

Capítulo 3

Conclusiones

3.1. Mejoras y líneas futuras

Mejoras posibles del juego:

- La posibilidad de venta al mercado, con personajes personalizados propios, ya que están en uso sprites y música que contienen CopyRight.
- Posibilidad de varias plataformas de juego, sobre todo enfocado para Android. Ya que pensamos que el mercado de videojuegos para móviles es mucho más amplio comparado a la plataforma PC.
- En lo referido a la jugabilidad, mejora de distintos métodos de juego, por ejemplo hacer que salga un punto al azar en el mapa, y el primer personaje tenga una mejora de daño y vida con respecto a los demás.
- Añadir personajes con ataque a distancia (no como los personajes de Naruto/Deku que son especiales) y otros personajes especiales que empujen.
- Mapas que afecten a la jugabilidad (zonas con más daño, con reducción de velocidad, etc.).
- La posibilidad de jugar online.
- Si el juego fuese online, conectar nuestro videojuego con una BBDD para tener un registro de Usuarios, un registro de partidas perdidas y partidas ganadas, etc.

3.2. Reflexión

Una vez terminado el proceso de desarrollo y comprobado que las diferentes funcionalidades implementadas funcionan correctamente podemos afirmar que nos hemos ajustado a las especificaciones originales que no eran otras que elaborar un videojuego multijugador para PC entretenido y con una temática actual como es el anime. En lo referente a los objetivos específicos consideramos que, como equipo los hemos superado; en primer lugar, hemos dominado las herramientas seleccionadas para el desarrollo del proyecto como Unity, en cuanto a la interfaz gráfica, hemos conseguido diseñar unos personajes fácilmente reconocibles y con habilidades muy llamativas en el plano de lo visual (como es el caso del smash de Deku o el rasengan de Naruto), así mismo hemos estado especialmente atentos a las mecánicas del juego para que la victoria no se consiga por una cuestión de azar, sino por una estrategia bien meditada. Para abordar con garantías este proyecto, el trabajo en equipo ha sido un factor clave. Hemos sido capaces de reorganizar las tareas en función de nuestras capacidades y gustos personales, lo que ha generado un ambiente motivador y agradable. Al principio elaboramos un cronograma detallado (Diagrama de Gantt) de los objetivos específicos con sus correspondientes tiempos y aunque, en líneas generales, lo hayamos respetado, lo cierto es que hemos ido modificando este planning inicial en función de las dificultades que hemos ido encontrando en las diferentes etapas de desarrollo. Es importante recalcar que ambos miembros del equipo hemos estado separados físicamente (a excepción de los viernes que teníamos que asistir a clase) y este hecho no ha sido impedimento para una

comunicación fluida, ya que además de las reuniones telemáticas cada semana vía Discord, nos hemos servido de una herramienta de vital importancia para un trabajo compartido como es la herramienta Git Lab. Finalmente, para el desarrollo de este proyecto hemos puesto en práctica conocimientos variados adquiridos a lo largo de todo el grado de DAM (Desarrollo de Aplicaciones Multiplataforma). En nuestra opinión, aplicar el conocimiento teórico-práctico a un proyecto en concreto nos ha servido para confirmar nuestra vocación (de hecho, hemos disfrutado resolviendo los distintos obstáculos) y nos ha permitido conocer más acerca de una las múltiples salidas profesionales de la programación, como es el desarrollo de videojuegos que esperamos proseguir en años venideros.

Capítulo 4

Anexos

4.1. Diagramas UML

Diagrama de Clases

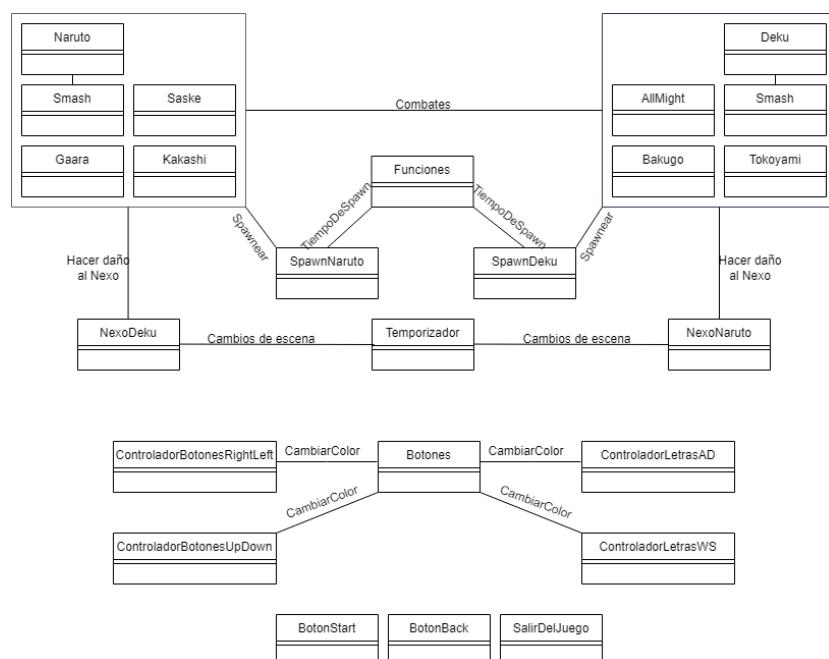


Figura 4.1: Diagrama de Clases General.

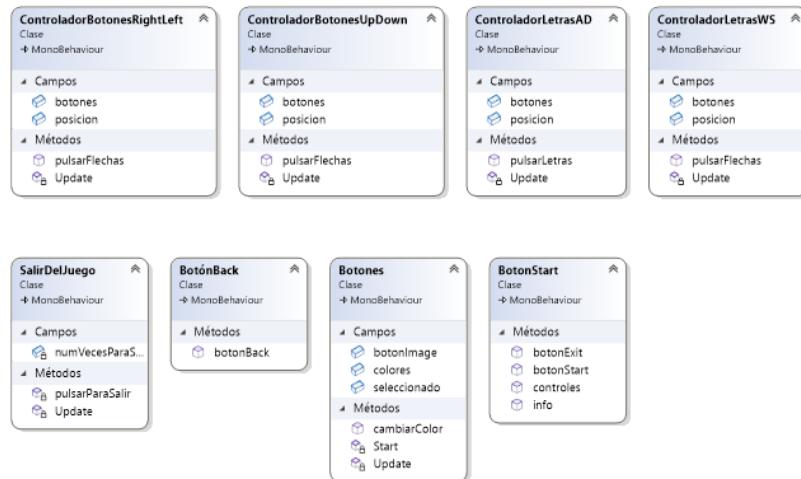


Figura 4.2: Clases Botones.

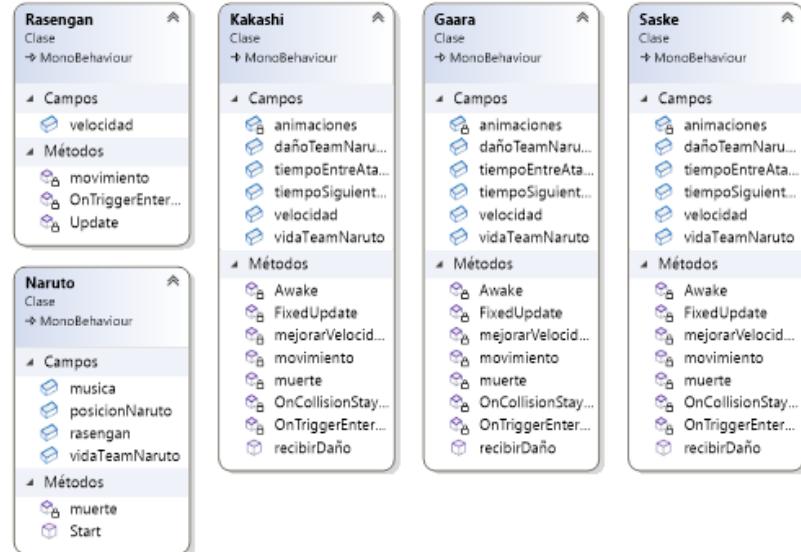


Figura 4.3: Diagrama de Clases Team Naruto.

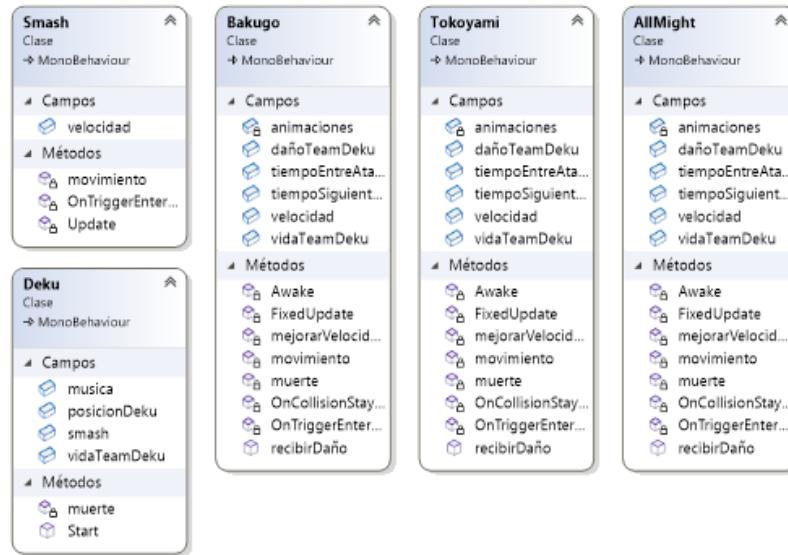


Figura 4.4: Clases Team Deku.

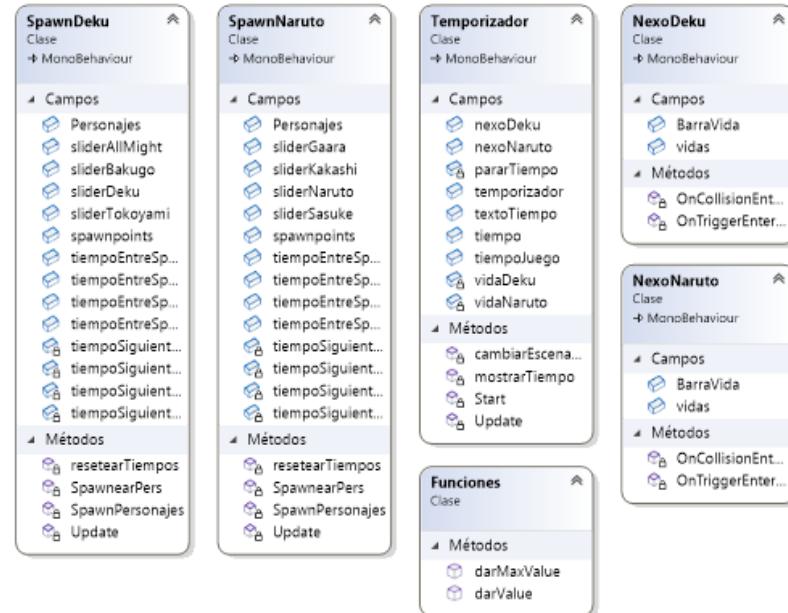


Figura 4.5: Clases Temporizador, Spawns, Nexos y Funciones.

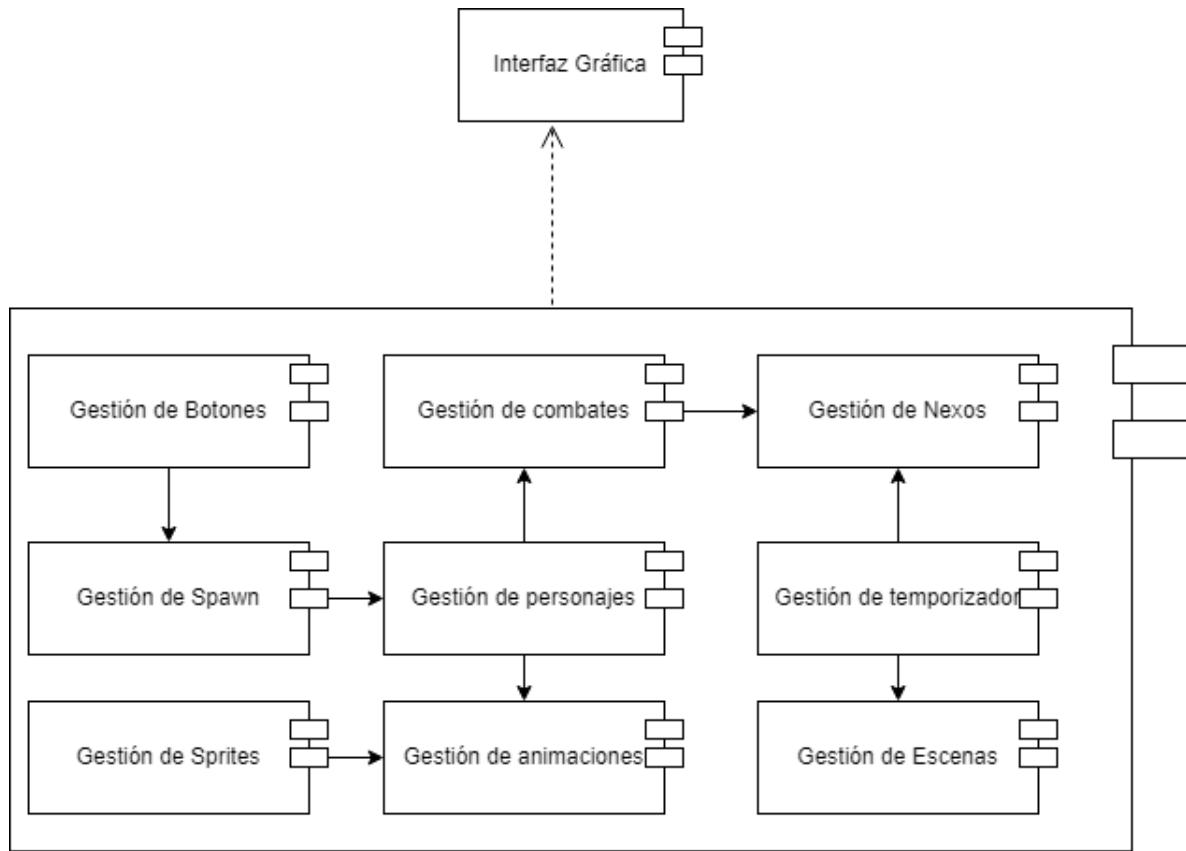
Diagrama de componentes

Figura 4.6: Diagrama de Componentes.

Diagrama de casos de uso

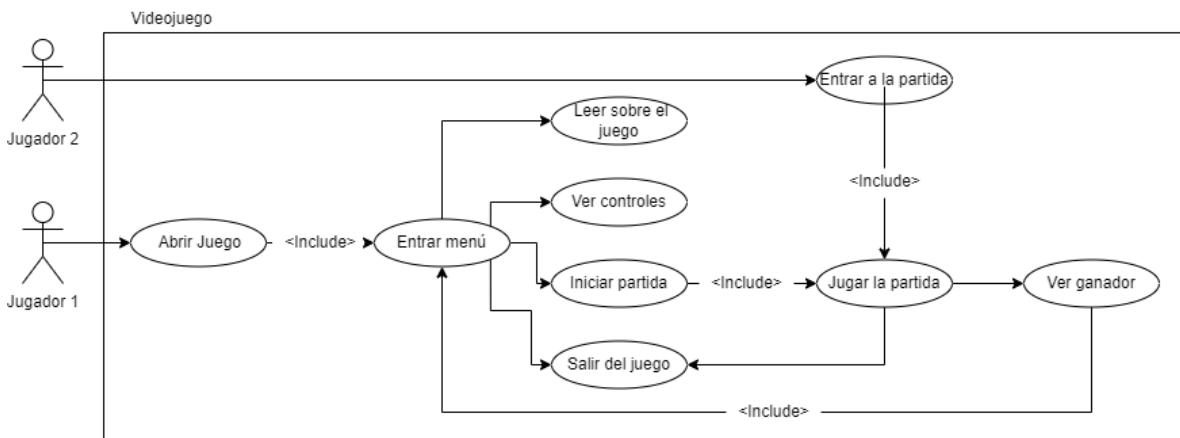


Figura 4.7: Diagrama de Caso de uso.

Diagrama de secuencia

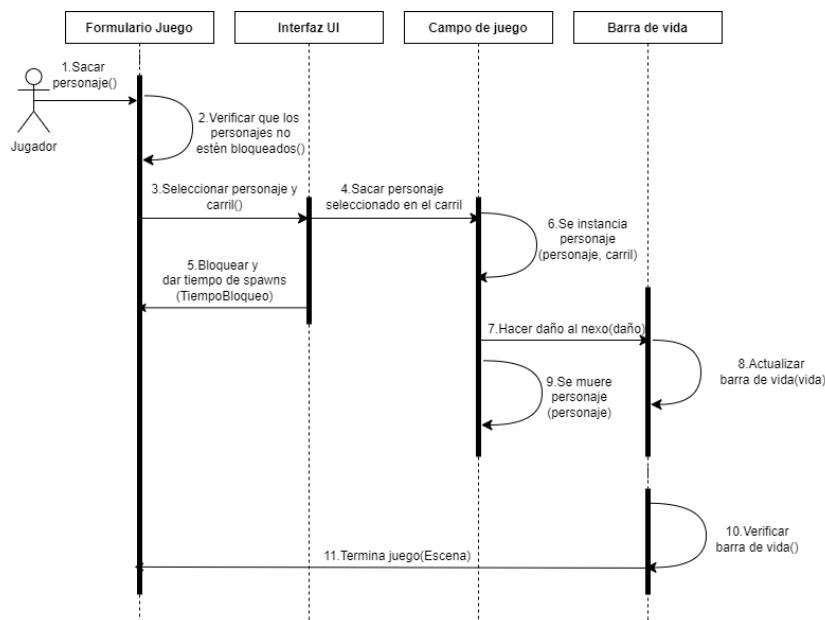


Figura 4.8: Diagrama de secuencia-sacar personaje.

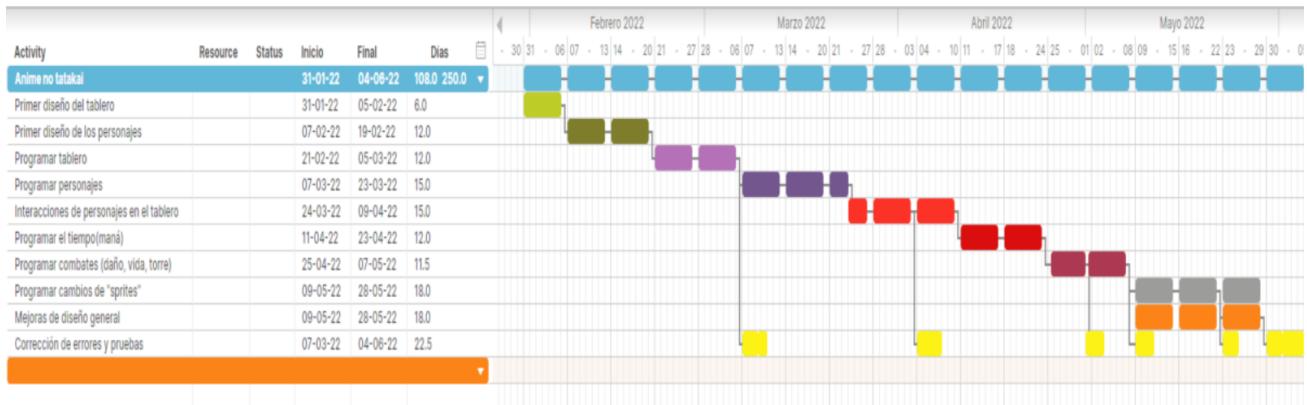


Figura 4.9: Primer Diagrama de Gantt.

Diagrama de Gantt

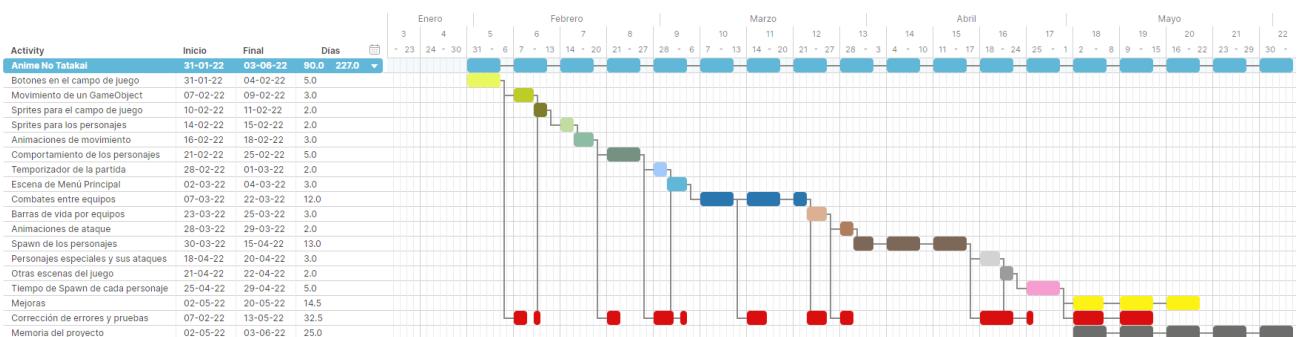


Figura 4.10: Diagrama de Gantt Actualizado.

4.2. Código / Clases

- Clase Botones

```
public Color[] colores;
public Image botonImage;
public bool seleccionado = false;

④ Mensaje de Unity | 0 referencias
void Start()
{
    //Hace referencia a la imagen de los botones
    botonImage = GetComponent<Image>();
    //El color empieza en blanco (no seleccionado)
    botonImage.color = colores[1];
}

④ Mensaje de Unity | 0 referencias
void Update()
{
    cambiarColor();
}

1 referencia
public void cambiarColor()
{
    if (seleccionado)
    {
        botonImage.color = colores[0];
    }
    else
    {
        botonImage.color = colores[1];
    }
}
```

Figura 4.11: Clase Botones.

- Clase Controladores de botones

```
//Creamos un array de objetos "botones"
public Botones[] botones;

//Para marcar la posición en la que nos encontramos
public int posicion = 0;

④ Mensaje de Unity | 0 referencias
void Update()
{
    pulsarFlechas();
}

1 referencia
public void pulsarFlechas()
{
    if (Input.GetKeyDown(KeyCode.RightArrow))
    {
        //El botón anterior se deselecciona
        botones[posicion].seleccionado = false;

        //Pasa a la siguiente posición
        posicion--;

        //Para que no se salga del array y pase al siguiente por el otro lado
        if(posicion < 0)
        {
            posicion = botones.Length - 1;
            botones[posicion].seleccionado = true;
        }
        if(posicion > botones.Length - 1)
        {
            posicion = 0;
            botones[posicion].seleccionado = true;
        }
        botones[posicion].seleccionado = true;
    }
}
```

Figura 4.12: Clase Controlador de Botones.

- Clase Personaje

```

public float velocidad;
public float vidaTeamDeku;
public float dañoTeamDeku;
private Animator animaciones;
public float tiempoEntreAtaques;
public float tiempoSiguienteAtaque;

//Para inicializar variables al iniciar el programa
@ Mensaje de Unity | 0 referencias
private void Awake()
{
    animaciones = GetComponent<Animator>();
}

@ Mensaje de Unity | 0 referencias
void FixedUpdate()
{
    movimiento();
    muerte();
    mejorarVelocidad();

    if (tiempoSiguienteAtaque > 0)
    {
        tiempoSiguienteAtaque -= Time.deltaTime;
    }
}

1 referencia
void movimiento()
{
    gameObject.transform.Translate(-velocidad * Time.deltaTime, 0, 0); //Eje x,y,z
}

3 referencias
public void recibirDaño(float dañoEnemigo)
{
    vidaTeamDeku -= dañoEnemigo;
}

1 referencia
private void muerte()
{
    if (vidaTeamDeku <= 0)
        Destroy(gameObject);
}

```

Figura 4.13: ClasePersonaje1.

```

//Para que cuando se choquen se cambie el animation a atacar
➊ Mensaje de Unity | 0 referencias
void OnCollisionStay2D(Collision2D other)
{
    if (other.gameObject.CompareTag("Sasuke") && tiempoSiguienteAtaque <= 0)
    {
        animaciones.SetTrigger("Atacar");
        other.collider.GetComponent<Sasuke>().recibirDaño(dañoTeamDeku);
        tiempoSiguienteAtaque = tiempoEntreAtaques;
    }

    if (other.gameObject.CompareTag("Gaara") && tiempoSiguienteAtaque <= 0)
    {
        animaciones.SetTrigger("Atacar");
        other.collider.GetComponent<Gaara>().recibirDaño(dañoTeamDeku);
        tiempoSiguienteAtaque = tiempoEntreAtaques;
    }

    if (other.gameObject.CompareTag("Kakashi") && tiempoSiguienteAtaque <= 0)
    {
        animaciones.SetTrigger("Atacar");
        other.collider.GetComponent<Kakashi>().recibirDaño(dañoTeamDeku);
        tiempoSiguienteAtaque = tiempoEntreAtaques;
    }

    if (other.gameObject.CompareTag("NexoNaruto"))
        Destroy(gameObject);
}

➋ Mensaje de Unity | 0 referencias
void OnTriggerEnter2D(Collider2D other)
{
    if (other.gameObject.CompareTag("Rasengan"))
    {
        Destroy(gameObject);
    }
}

➌ 1 referencia
void mejorarVelocidad()
{
    float tiempoMejoras;
    tiempoMejoras = FindObjectOfType<Temporizador>().tiempo;
    if (tiempoMejoras < 20)
        velocidad += 0.003f;
}

```

Figura 4.14: ClasePersonaje2.

- Clase Nexo

```
public float vidas;
public Slider BarraVida;

//Si choca algún objeto con el nexo se baja vida
@ Mensaje de Unity | 0 referencias
void OnCollisionEnter2D(Collision2D other)
{
    if (other.gameObject.CompareTag("Gaara"))
    {
        float dañoGaara = other.collider.GetComponent<Gaara>().dañoTeamNaruto;
        vidas -= dañoGaara;
    }

    if (other.gameObject.CompareTag("Sasuke"))
    {
        float dañoSasuke = other.collider.GetComponent<Sasuke>().dañoTeamNaruto;
        vidas -= dañoSasuke;
    }

    if (other.gameObject.CompareTag("Kakashi"))
    {
        float dañoKakashi = other.collider.GetComponent<Kakashi>().dañoTeamNaruto;
        vidas -= dañoKakashi;
    }

    BarraVida.value = vidas;
}

@ Mensaje de Unity | 0 referencias
void OnTriggerEnter2D(Collider2D other)
{
    if (other.gameObject.CompareTag("Rasengan"))
    {
        vidas -= 5;
    }
    BarraVida.value = vidas;
}
```

Figura 4.15: Clase Nexo.

- Clase Temporizador

```

public Slider temporizador;
public Text textoTiempo;
public float tiempoJuego;
//Para que cuando llegue a 00:00 se pare el temporizador
private bool pararTiempo;
public float tiempo;
private float vidaNaruto;
private float vidaDeku;
public NexoNaruto nexoNaruto;
public NexoDeku nexoDeku;

➊ Mensaje de Unity | 0 referencias
void Start()
{
    pararTiempo = false;
    temporizador.maxValue = tiempoJuego;
    temporizador.value = tiempoJuego;
}

➋ Mensaje de Unity | 0 referencias
void Update()
{
    mostrarTiempo();
    cambiarEscenaGanador();
}

➌ 1 referencia
private void mostrarTiempo()
{
    //Para que se vaya restando segundo a segundo
    tiempo = tiempoJuego - Time.timeSinceLevelLoad;

    //Minutos y segundos
    int minutos = Mathf.FloorToInt(tiempo / 60);
    int segundos = Mathf.FloorToInt(tiempo % 60);
    //Formato en el que queremos el tiempo
    string textTime = string.Format("{00:00}:{01:00}", minutos, segundos);

    if (pararTiempo == false)
    {
        textoTiempo.text = textTime;
        temporizador.value = tiempo;
    }
}

```

Figura 4.16: Clase Temporizador1.

```
private void cambiarEscenaGanador()
{
    vidaNaruto = FindObjectOfType<NexoNaruto>().BarraVida.value;
    vidaDeku = FindObjectOfType<NexoDeku>().BarraVida.value;

    //Si el tiempo llega a 0 se para
    if (tiempo < 1)
    {
        pararTiempo = true;
        if (vidaNaruto < vidaDeku)
        {
            SceneManager.LoadScene("GameOverDeku");
        }
        else
        {
            SceneManager.LoadScene("GameOverNaruto");
        }
    }

    //Gana Deku
    if (vidaNaruto < 1)
    {
        SceneManager.LoadScene("GameOverDeku");
    }

    //Gana Naruto
    if (vidaDeku < 1)
    {
        SceneManager.LoadScene("GameOverNaruto");
    }

    //Empate
    if (tiempo < 1 && vidaDeku == vidaNaruto)
    {
        SceneManager.LoadScene("GameOverEmpate");
    }
}
```

Figura 4.17: Clase Temporizador2.

- Clase Spawn

```
void SpawnPersonajes()
{
    int selPersonajes = FindObjectOfType<ControladorLetrasAD>().posicion;
    int SelSpawns = FindObjectOfType<ControladorLetrasWS>().posicion;
    //Spawnea el personaje elegido, en la posicion que queramos
    GameObject spawnedPers = Instantiate(Personajes[selPersonajes], spawnpoints[SelSpawns]);

    if (spawnedPers.CompareTag("Sasuke"))
    {
        tiempoSiguienteSpawnSasuke = tiempoEntreSpawnsSasuke;
        Funciones.darMaxValue(tiempoSiguienteSpawnSasuke, sliderSasuke, sliderKakashi, sliderGaara, sliderNaruto);
    }

    if (spawnedPers.CompareTag("Kakashi"))
    {
        tiempoSiguienteSpawnKakashi = tiempoEntreSpawnsKakashi;
        Funciones.darMaxValue(tiempoSiguienteSpawnKakashi, sliderSasuke, sliderKakashi, sliderGaara, sliderNaruto);
    }

    if (spawnedPers.CompareTag("Gaara"))
    {
        tiempoSiguienteSpawnGaara = tiempoEntreSpawnsGaara;
        Funciones.darMaxValue(tiempoSiguienteSpawnGaara, sliderSasuke, sliderKakashi, sliderGaara, sliderNaruto);
    }

    if (spawnedPers.CompareTag("Naruto"))
    {
        tiempoSiguienteSpawnNaruto = tiempoEntreSpawnsNaruto;
        Funciones.darMaxValue(tiempoSiguienteSpawnNaruto, sliderSasuke, sliderKakashi, sliderGaara, sliderNaruto);
    }
}
```

Figura 4.18: Clase Spawn.

- Clase Personajes especiales

```

public float vidaTeamNaruto;
public Transform posicionNaruto;
public GameObject rasengan;
public AudioSource sonido;

@ Mensaje de Unity | 0 referencias
void Start()
{
    Instantiate(rasengan, posicionNaruto);
    sonido.Play();
    StartCoroutine(muerte());
}

1 referencia
IEnumerator muerte()
{
    yield return new WaitForSeconds(5);
    Destroy(gameObject);
}

```

Figura 4.19: Clase Personaje Especial.

- Clase Habilidades especiales

```

public float velocidad;

@ Mensaje de Unity | 0 referencias
void Update()
{
    movimiento();
}

1 referencia
void ...movimiento()
{
    gameObject.transform.Translate(velocidad * Time.deltaTime, 0, 0);
}

//Puesto en trigger para que no empuje a sus compañeros
@ Mensaje de Unity | 0 referencias
void OnTriggerEnter2D(Collider2D other)
{
    if (other.gameObject.CompareTag("NexoDeku"))
    {
        Destroy(gameObject);
    }
}

```

Figura 4.20: Clase Habilidad Especial.

Bibliografía

- [1] Antarsoft. *Instanciar personajes con prefabs*. 19 de diciembre de 2020. URL: <https://youtu.be/jY9TGLBAZr8>.
- [2] Antarsoft. *Instanciar personajes enemigos*. 17 de abril de 2021. URL: <https://youtu.be/L2QXWnUZ4e4>.
- [3] LLC DBA GitKraken Axosoft. *Características de GitKraken*. URL: <https://www.gitkraken.com/git-client/features>.
- [4] Mitchell William Cooper. *Separar Sprites Alferd Spritesheet Unpacker*. URL: <https://github.com/ForkandBeard/Alferd-Spritesheet-Unpacker/releases>.
- [5] DédaloLab. *Acciones de pelea de los personajes*. 23 de diciembre de 2021. URL: <https://youtu.be/HzGWrVbsXKg>.
- [6] DédaloLab. *Menú Principal del juego*. 23 de febrero de 2021. URL: <https://youtu.be/3zPgDONZik>.
- [7] Ben Olding Games. *Descripción del juego Warlords CallToArms*. 1 de abril de 2008. URL: <https://archive.org/details/warlords-calltoarms>.
- [8] Turbo Makes Games. *Temporizador en tu partida*. 30 de octubre de 2019. URL: <https://youtu.be/qc7J0iei3BU>.
- [9] Electronic Arts Inc. *Descripción del juego Plants vs Zombies*. 9 de agosto de 2010. URL: <https://www.ea.com/es-es/games/plants-vs-zombies/plants-vs-zombies#description>.
- [10] GitLab Inc. *Manual de los Repositorios GitLab*. URL: <https://docs.gitlab.com/ee/user/project/repository/>.
- [11] Rocket Jam. *Como se mueven los personajes*. 13 de septiembre de 2020. URL: https://youtu.be/P_XnhYSrFM4.
- [12] JoexScript. *Disparando en movimiento y Carga de disparo*. 10 feb 2021. URL: <https://youtu.be/HFhJZ4FsjeU>.
- [13] KMOON. *Como animar a los personajes en Unity 2D*. 20 de mayo de 2020. URL: <https://youtu.be/4vLYoFWV5lk>.
- [14] Code Monkey. *Uso de colissions y Triggers*. 18 de diciembre de 2019. URL: <https://youtu.be/Bc91mHjqLZc>.
- [15] najeraretrrogames. *Blog donde se resume el juego Space Invaders*. El juego es de Toshihiro Nishikado en 1978. 2de octubre de 2016. URL: <https://najeraretrrogames.com/space-invaders-taito-1978/>.
- [16] Unity Technologies. *Manual de Unity Hub*. URL: <https://docs.unity3d.com/hub/manual/index.html>.
- [17] Unity Technologies. *Manual Unity Editor 2021.1*. URL: <https://docs.unity3d.com/es/2021.1/Manual/UnityManual.html>.
- [18] Unity Technologies. *Manual Unity Editor 2021.1 Clases*. URL: <https://docs.unity3d.com/es/2021.1/Manual/ScriptingImportantClasses.html>.
- [19] Unity Technologies. *Versión de Unity Editor 2021.3.3*. URL: <https://unity3d.com/es/unity/whats-new/2021.3.3>.
- [20] Unity Technologies. *Versiones de Unity Editor*. URL: <https://unity3d.com/unity/qa/lts-releases>.