UNIVERSIDAD DE CONCEPCIÓN
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE INGENIERÍA INFORMÁTICA Y
CIENCIAS DE LA COMPUTACIÓN

DESARROLLO DE UN VIDEO JUEGO CON VINCULACIÓN TERRITORIAL

POR

MARTÍN ENRIQUE MONJE OURCILLEON

MEMORIA PRESENTADA PARA LA OBTENCIÓN DEL TÍTULO DE

INGENIERO CIVIL INFORMÁTICO

PATROCINANTE

MARCELA VARAS CONTRERAS

Agosto de 2023

Concepción, Chile

Sumario

La industria de los videojuegos ha crecido significativamente en las últimas décadas, lo cual hace interesante estudiar el proceso de desarrollo de un videojuego. En este informe se aborda el proceso de desarrollo de un prototipo ilustrativo para un videojuego basado en la mitología y folklore del sur de Chile.
Los objetivos específicos incluyen identificar los aspectos arquitectónicos principales del juego, seleccionar una metodología de desarrollo adecuada y determinar los elementos críticos para el desarrollo completo del videojuego. Además, se busca desarrollar un prototipo que permita levantar recursos financieros y un equipo humano para completar el proyecto.
El género elegido para el videojuego es el de plataformero, con énfasis en combate y movimiento. La herramienta seleccionada para el desarrollo es el motor Unity, que ofrece una amplia gama de recursos y documentación.
El prototipo desarrollado incluye un personaje principal, mecánicas de movimiento y combate, enemigos, sistema de vida y una bitácora folklórica que proporciona información adicional sobre los personajes y criaturas del folklore al jugador.
La evaluación del proyecto demostró que se lograron los objetivos establecidos para el prototipo, y se considera prometedora la idea del videojuego. Se destacó la importancia de la iteración y la mejora continua en el desarrollo.
En cuanto a futuros trabajos, se requiere obtener financiamiento, conformar un equipo de desarrollo y generar una versión del juego que se pueda disponibilizar al mercado, incluyendo gráficos definitivos, niveles, enemigos y posiblemente funcionalidades multijugador.
El desarrollo del prototipo del videojuego fue un éxito en términos de validación de ideas y creación de una base sólida para futuras iteraciones. El juego muestra gran potencial para captar interés cultural y entretener a los jugadores, lo que sugiere que la continuación del proyecto puede ser prometedora.

Abstract

The video game industry has grown significantly in recent decades, which makes it interesting to study the process of developing a video game. This report addresses the development process of a prototype for a video game based on the mythology and folklore of southern Chile.

The specific objectives include identifying the main architectural aspects of the game, selecting an appropriate development methodology and determining the critical elements for the complete development of the game. In addition, it seeks to develop a prototype that attracts financial resources and a development team to complete the project.

The genre chosen for the video game is that of a platformer, with an emphasis on combat and movement. The selected tool for development is the Unity engine, which offers a wide range of resources and documentation.

The developed prototype includes a main character, movement and combat mechanics, enemies, a life system, and a log that provides additional information about the folklore characters and creatures to the player.

The evaluation of the project showed that the objectives established for the prototype were achieved, and the idea of the video game is considered promising. The importance of iteration and continuous improvement in development was highlighted.

As for future work, it requires getting funding, assembling a development team, and developing a complete version of the game ready to be put on the market, including final graphics, levels, enemies, and possibly multiplayer functionality.

The development of the video game prototype was a success in terms of validating ideas and creating a solid foundation for future iterations. The game shows great potential to capture cultural interest and entertain players, which suggests that the continuation of the project has a lot of promise.

Índice

1.Introducción

1.1.Motivación del proyecto

La industria de los videojuegos ha mostrado un gran y sostenido aumento en las últimas décadas, tanto en su relevancia en el mercado, como en la complejidad técnica de los juegos publicados hoy en día, y es por esto que es interesante explorar cómo es el proceso de desarrollo de un videojuego en un contexto actual. Esto se toma además como una oportunidad para explorar temáticas no abordadas por el resto del mercado, como es el folklore y la mitología proveniente de la región sur de Chile la cual no se ve representada comúnmente en el medio de los videojuegos.

Para este proyecto se tomó como punto de partida un concepto concebido por el profesor Pedro Andrade Martinez[1], académico de la Universidad de Concepción, quien tomó el rol de Productor del videojuego durante el desarrollo de esta memoria, el cual corresponde a un rol de supervisión y guía para el proceso de desarrollo de un videojuego. La base de este concepto corresponde a tomar a un personaje importante y reconocido para la historia de la Universidad y de la ciudad, Enrique Molina Garmendia, y ponerlo como el protagonista de una historia original que lo lleve por distintas áreas del sur del país, encontrándose, y enfrentándose, con distintos personajes y criaturas de la mitología y el folklore de la zona. Una descripción más detallada de la historia se encuentra en el Anexo 1. Se pretende introducir a los jugadores a estos personajes mientras ellos juegan, manteniendo el enfoque en la jugabilidad para que el juego sea por sobre todo entretenido, pero sin dejar de lado la identidad y el trasfondo que los personajes representan. Se planteó también el darle a los jugadores la opción de leer más información de los personajes que encuentren al jugar si así lo desean, para profundizar en sus orígenes y cualidades sin interrumpir el flujo del juego.

[1] Pedro Andrade Martinez, Facultad de Ciencias Sociales y Facultad de Humanidades y Arte, Universidad de Concepción. Correo: pandradem@udec.cl

1.2. Objetivo General

El objetivo general para esta Memoria de Título es desarrollar un prototipo ilustrativo de los aspectos claves para un videojuego con vinculación territorial, utilizando herramientas de desarrollo de videojuegos aptas para el proyecto.

1.3. Objetivos Específicos

- Identificar los aspectos arquitectónicos principales de un videojuego, de acuerdo con la visión del Productor.

  Coordinar con el Productor es parte crítica del proyecto, ya que dejar en claro qué tipo de juego se desea desarrollar es el primer paso para determinar los requerimientos que dicho juego debe satisfacer. Distintos tipos de juego ponen énfasis en diferentes áreas de la experiencia de juego, algunos se enfocan principalmente en la jugabilidad de este, mientras que otros pueden priorizar la historia que este presenta o la fidelidad gráfica del juego. Es por esto que establecer dónde se va a poner el énfasis para el juego a desarrollar es de suma importancia.
- Seleccionar un enfoque metodológico y herramientas apropiadas para abordar el desafío.

  Una de las primeras decisiones que se deben tomar para comenzar el desarrollo es escoger una metodología que sea apta para el proyecto, entre las diferentes opciones existentes para el desarrollo de software, como por ejemplo una metodología ágil o una de tipo cascada. Además, hoy en día existe un gran número de alternativas para el desarrollo de videojuegos en particular, como por ejemplo los motores de juego Unity, Unreal Engine o Godot, por lo que es importante determinar cuáles herramientas son más aptas para este proyecto en específico, y que cumplan con las necesidades de este.

- Desarrollar un prototipo que permita levantar recursos financieros y conformar un equipo humano que posibilite completar el desarrollo del videojuego.

  El objetivo de este prototipo es representar lo mejor posible la visión que se tiene para el videojuego, demostrando las características claves de este que lo hacen destacar como una idea interesante y factible, para así captar el interés de posibles inversionistas y desarrolladores dispuestos a prestar los recursos financieros y humanos necesarios para completar el desarrollo del videojuego.
- Identificar y dimensionar los aspectos críticos para el desarrollo completo del videojuego.

  Si bien el objetivo principal de la memoria es desarrollar un prototipo, es importante establecer qué elementos quedan por incluir en el juego completo, los cuales debieran ser implementados en un futuro desarrollo del videojuego final.

1.4.Metodología utilizada

Para este proyecto, se decidió que la metodología más apta para el desarrollo del prototipo era una metodología ágil, y se inició el proceso con la idea de implementar el marco de trabajo Scrum. Esta técnica de desarrollo se basa principalmente en determinar una lista de requerimientos en base a historias de usuario, para luego escoger cierto número de estos a desarrollar dentro de un periodo delimitado de tiempo llamado Sprint. Una vez finalizado este Sprint se evalúa el trabajo realizado junto al Product Owner, intercambiando comentarios, opiniones y posibles cambios que fueran necesarios de implementar antes de comenzar el siguiente Sprint. Luego de múltiples Sprints se llega finalmente a una versión del producto, y se hace una evaluación de este, el cual corresponde al prototipo final.

Sin embargo, debido a las circunstancias encontradas durante el periodo de trabajo de este proyecto, principalmente problemas de disponibilidad y consolidación de reuniones, no fue posible atenerse a la estructura del proceso Scrum como se hubiera deseado. Esto resultó en tener que adaptar el marco de trabajo, particularmente en la frecuencia de las reuniones y en el número de requerimientos que se plantearon para cada Sprint, lo cual se vio reflejado en reuniones menos frecuentes donde se abordó un mayor número de requerimientos comparado con lo que es usual para Scrum, terminando con una metodología que más se aproxima a múltiples instancias de desarrollo en cascada, pero tratando de mantener el concepto de agilidad lo más posible a través de un desarrollo incremental en base a retroalimentación con, en este caso, el Productor del videojuego, Pedro Andrade.

2.Marco Teórico

2.1.Antecedentes

Los videojuegos, también llamados juegos de video, son formas interactivas de entretenimiento digital en donde los jugadores asumen el control de personajes virtuales y participan en desafíos o narrativas dentro de un ambiente virtual, y a través de las decisiones que estos toman, afectan el desarrollo y resultado del juego. Un videojuego utiliza gráficos, sonidos y distintos medios de control para crear experiencias que capten la atención de los jugadores y proveer entretenimiento a estos, sin embargo existe también la posibilidad de aplicar los videojuegos en diferentes áreas más allá del ocio, como la educación y el entrenamiento de diversas habilidades.

El desarrollo de videojuegos tiene sus inicios alrededor de 1950 [1], cuando los primeros experimentos con gráficos y computadoras dieron lugar a la creación de juegos electrónicos rudimentarios. En la década de 1970, la popularidad de los juegos en arcades como “Pong” y “Space Invaders” marcó el comienzo de los videojuegos como una industria capaz de generar gran interés en la población. Con el pasar del tiempo, surgieron nuevas tecnologías y plataformas en donde publicar videojuegos, como son las consolas domésticas, los computadores personales y más recientemente los dispositivos móviles, principalmente celulares. Durante todo este tiempo, los videojuegos evolucionaron rápidamente, desde sus inicios con juegos en dos dimensiones y gráficos sumamente simples, hasta los juegos de hoy donde se puede llegar a tener juegos capaces de representar ambientes complejos en tres dimensiones que son casi indistinguibles de la realidad, creando así experiencias más inmersivas. Hoy en día, los videojuegos son una de las formas de entretenimiento más populares y una industria multimillonaria, impulsada por avances tecnológicos, narrativas complejas y comunidades de jugadores en línea [2].

2.2.Bases Teóricas

- Elementos del desarrollo de videojuegos:

  El proceso de conceptualizar y diseñar un videojuego implica varias etapas, y todo comienza con la definición de conceptos clave, como el género, que tipo de juego se quiere generar, las mecánicas que va a tener, o en otras palabras qué elementos van a diferenciar al juego de otros dentro del mismo género, y la narrativa, la cual incluye tanto los personajes y ubicaciones, como la historia que se va a presentar a lo largo del juego al jugador. Estos conceptos van a determinar la dirección que el resto del desarrollo va a tomar y donde se va a poner mayor enfoque a lo largo de este. A continuación, se elabora un diseño inicial que incluye aspectos como el arte, la música, los niveles y la jugabilidad, aquellos elementos que están más cercanamente relacionados con el desarrollo mismo del juego. A lo largo del desarrollo se generan prototipos para probar y refinar distintas partes del juego, recopilando retroalimentación de distintos testers, para luego incorporar esta información en varias iteraciones continuas para mejorar y pulir el diseño, hasta llegar a una versión final.

- Tecnologías y herramientas en el desarrollo de videojuegos:

  El desarrollo de videojuegos requiere de diversas tecnologías y herramientas para crear los distintos elementos necesarios para proporcionar experiencias interactivas y envolventes a los jugadores. Una de las herramientas más populares y poderosas son los llamados motores de juego, como Unity y Unreal Engine, los cuales proporcionan un entorno de desarrollo completo que incluye herramientas visuales, sistemas de física, sistemas avanzados para el manejo de gráficos, además de soporte multiplataforma, lo que facilita la creación de juegos para diferentes dispositivos. Cada uno de estos motores cuenta con lenguajes de programación específicos, como C# en Unity y C++ en Unreal Engine, los cuales permiten a los desarrolladores crear la lógica del juego y personalizar su funcionamiento más allá de lo que es posible con las herramientas proporcionadas por el motor por sí solas.

Además de los motores de juego, existen varios lenguajes de programación utilizados en el desarrollo de videojuegos. C++ es ampliamente utilizado en proyectos de grandes estudios debido a su buen rendimiento y control de bajo nivel sobre los recursos del sistema. Otros lenguajes populares incluyen C#, utilizado por el motor Unity, y JavaScript, que se emplea en el desarrollo de juegos web y aplicaciones móviles. Estos lenguajes permiten a los desarrolladores implementar las mecánicas específicas al juego en desarrollo, incluyendo diferentes aplicaciones de distintos conceptos de la programación a gran escala, como son la inteligencia artificial, la gestión de recursos, entre otros.

En cuanto a bibliotecas y frameworks útiles para el desarrollo de videojuegos, existe una variedad de opciones disponibles. Por ejemplo, DirectX y OpenGL proporcionan funcionalidades gráficas avanzadas para el desarrollo de juegos de PC. En cambio para la programación de juegos para navegador, se utilizan bibliotecas como Phaser y Three.js las cuales entregan diversas utilidades para el desarrollo en este ámbito. Además, existen bibliotecas especializadas en diferentes áreas del desarrollo como simulaciones de física, (ej. Box2D [3] y Bullet Physics [4]), audio, (ej. FMOD [5] y Wwise [6]), inteligencia artificial, (ej. Behavior3 [7] y GOAP [8]), principalmente para modelar el comportamiento de enemigos y otros personajes dentro de los juegos, y conectividad para juegos multijugador, (ej. Photon [9] y Unity Multiplayer Networking [10]). Estas librerías facilitan el desarrollo de aspectos más complejos dentro de los juegos.

- Aspectos económicos y de negocio en el desarrollo de videojuegos:
  Una parte fundamental del desarrollo de videojuegos son los aspectos económicos y de negocio, los cuales determinan en gran parte el éxito y la sostenibilidad de un proyecto. Existen varios modelos de negocio en la industria de los videojuegos, comúnmente referidos como modelos de monetización dentro de la industria, y dentro de estos, los más comunes son los siguientes: Juegos gratuitos con compras integradas para acceder contenido adicional (freemium), juegos en base a suscripciones para jugar de manera continua, típicamente utilizado para juegos multijugador masivos en línea (MMO), y juegos de venta única, los cuales se paga una única vez y se procede a tener acceso a todo el contenido de forma permanente. Cada uno de estos modelos viene con sus ventajas y desventajas, por lo cual es importante determinar cual de estos es más apto para el tipo de juego que se pretende desarrollar [11]. La elección del modelo adecuado depende del tipo de juego, el público objetivo y los objetivos financieros del desarrollador.
  La monetización es un aspecto clave, y se logra a través de diversas estrategias, como la venta de copias del juego, la venta de contenido adicional, las microtransacciones, la publicidad integrada o los patrocinios. La eficacia de estas estrategias se basa en comprender a la audiencia, equilibrar el valor percibido y mantener una relación justa entre el costo y lo que los usuarios obtienen a cambio.
  El marketing y la promoción son esenciales para atraer a los jugadores y generar interés en el juego. Esto implica desarrollar una estrategia de marketing adecuada, que puede incluir el uso de redes sociales, eventos, influencers y campañas publicitarias dirigidas. Además, la elección de las plataformas de distribución adecuadas, como Steam, App Store o consolas de videojuegos, puede tener un impacto significativo en la visibilidad y el alcance del juego.

3.Desarrollo

3.1.Género y temática elegidos

Una de las primeras decisiones críticas para el subsecuente desarrollo del proyecto corresponde a qué género se va a elegir para el juego a desarrollar. Luego de conversar con el Productor sugiriendo distintas posibilidades, se llegó a la decisión de hacer un juego en el género plataformero, con énfasis en combate y movimiento, tomando inspiración de la serie de juegos Castlevania, la cual incorpora elementos mitológicos para sus ambientes y personajes, además de tener una buena jugabilidad que es fácil de comprender rápidamente.

Figura 1. Captura de pantalla del juego *Castlevania: Grimoire of Souls*

Es por estas razones que se consideró como una buena opción para la visión del Productor, ya que permite mostrar el ambiente y los personajes folklóricos de forma más interactiva, para luego darle la opción a los jugadores de aprender más acerca de los personajes que encuentren en el juego si lo desean mediante el uso de una bitácora que pueden acceder desde el menú.

Teniendo ya tomadas estas decisiones, se establecieron a grandes rasgos los requerimientos necesarios para el proyecto, los cuales son los siguientes:

- Generar una experiencia de juego entretenida y captadora.
- Asegurar que el personaje se sienta bien al jugar, siendo fácil e intuitivo de controlar.
- Entregar al jugador la opción de leer más información acerca de los personajes folklóricos que encuentra dentro del juego.

A continuación se incluye un esquema que representa las interacciones entre entidades más importantes del juego, resumiendo en términos generales la estructura del juego.

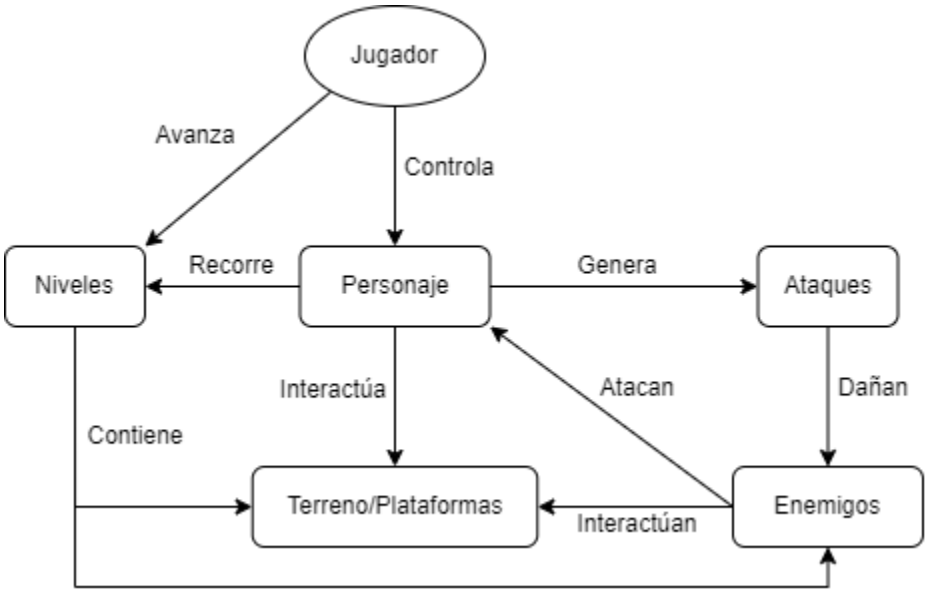


Figura 2. Esquema de interacciones entre entidades del juego.

3.2.Selección de las herramientas a utilizar

Considerando los requerimientos necesarios para el género seleccionado, y tomando en cuenta la disponibilidad y documentación existente, se decidió utilizar el motor de videojuegos Unity, el cual entrega una gran cantidad de herramientas que permiten desarrollar y testear un juego de plataformas en dos dimensiones de forma rápida y eficiente.

Unity funciona en base a escenas, básicamente un nivel o sección específica del juego, tal como un menú o minijuego, la cual almacena múltiples *GameObjects* [12]. Estos *GameObjects* corresponden básicamente a todos los elementos presentes en el juego, desde el personaje jugable, los enemigos, el terreno, hasta elementos como la interfaz gráfica, los efectos de sonido, e incluso la cámara que determina qué elementos mostrar en la pantalla del jugador. Estos GameObjects se diferencian entre ellos y obtienen su funcionalidad a través de *Components* [13], los cuales les entregan características específicas que determinan que se puede hacer con un determinado *GameObject*. De esta misma manera es posible implementar nuevas funcionalidades específicas para el juego en desarrollo a través de *Scripts*, segmentos de código en C# que pueden luego ser enlazadas a un *GameObject* como cualquier otro *Component*, permitiendo así implementar comportamientos más complejos y especializados de lo que se puede hacer solo con los Components ofrecidos por Unity.

Para la gran mayoría de los *Scripts* se necesita de al menos dos funciones principales: *Start()* y *Update()* o *FixedUpdate()*, la primera de estas es llamada por Unity cada vez que se crea una instancia del *GameObject* que tenga dicho *Script* como componente y se usa principalmente para setear variables o condiciones necesarias al principio de la vida del objeto; las otras dos funciones son ejecutadas múltiples veces mientras corre el juego, y la diferencia entre ellas es que *Update()* se ejecuta en cada *frame*, (unidad de tiempo para los videojuegos que separa un momento de otro), mientras que *FixedUpdate()* se ejecuta en un periodo determinado de *frames*, unidades de tiempo usadas por los videojuegos, según se especifica en el proyecto de Unity. Ambas funciones se usan para implementar funcionalidades que deben ocurrir mientras el juego está andando, en particular aquellas que requieran captar las acciones que realiza el jugador. Otro elemento importante para Unity son los llamados *Prefabs*, u objetos prefabricados, los cuales como su nombre se indica son *GameObjects* creados de antemano que pueden luego ser agregados a una o múltiples escenas rápidamente, e incluso durante la ejecución del juego mediante *Scripts*. En cuanto al manejo de los recursos, Unity utiliza un sistema basado en un *heap* o pila de memoria disponible, y se entrega una parte de este a los objetos al momento de ser creados en una escena, y de la misma manera se libera la sección de memoria correspondiente a un objeto cuando este deja de estar en uso.

3.3.Diseño del juego

A continuación se presenta un diagrama que muestra las distintas clases, implementadas dentro de Unity como *Objects*, junto con los *Components* y *Scripts* que estos contienen. Las relaciones entre clases corresponden a los métodos de Unity utilizados para manejar las interacciones entre los objetos correspondientes.

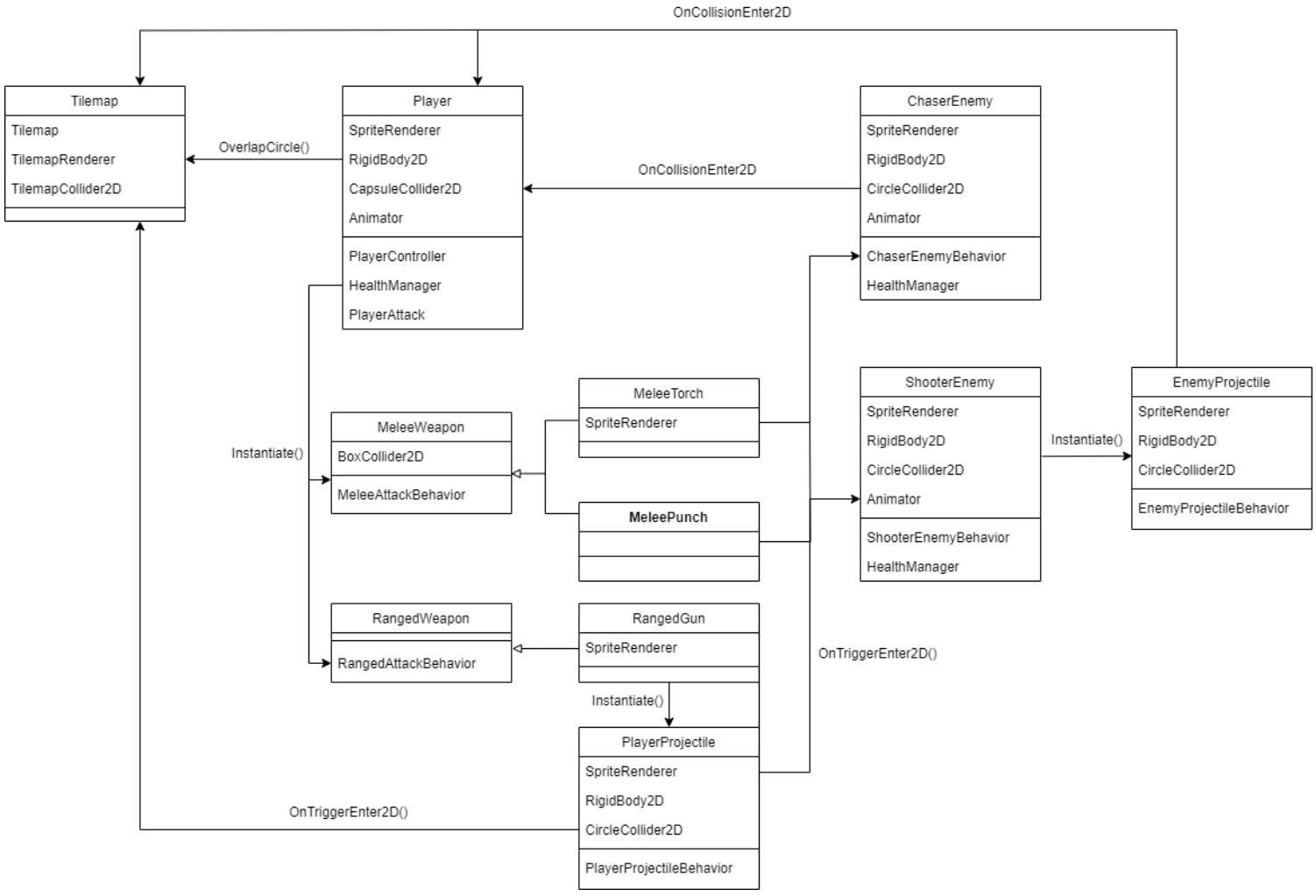


Figura 3. Diseño del juego dentro de Unity.

- *Player*: Objeto correspondiente al personaje principal que maneja el jugador. Interactúa con el terreno (Tilemap) mediante el método *OverlapCircle()*, el cual detecta cuando un círculo posicionado en sus pies se encuentra sobre el suelo, y es necesario para determinar cuándo tiene permitido saltar. Además, crea instancias de los objetos *MeleeWeapon* y *RangedWeapon* mediante el método *Instantiate()* al momento de realizar un ataque.

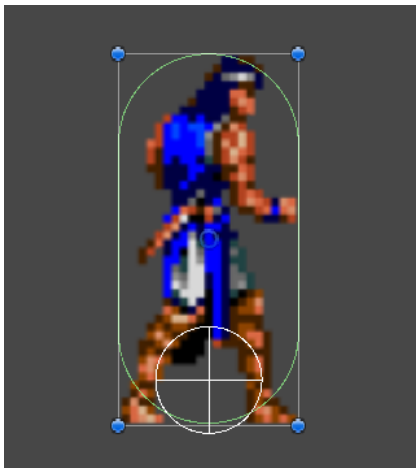


Figura 4. El objeto *Player* como se ve dentro del editor de Unity.

- *Tilemap*: Corresponde al terreno y las plataformas que componen al nivel. Utiliza el sistema de *Tilemaps* de Unity para generar la topografía en base a casillas que se llenan con distintos *Sprites*.

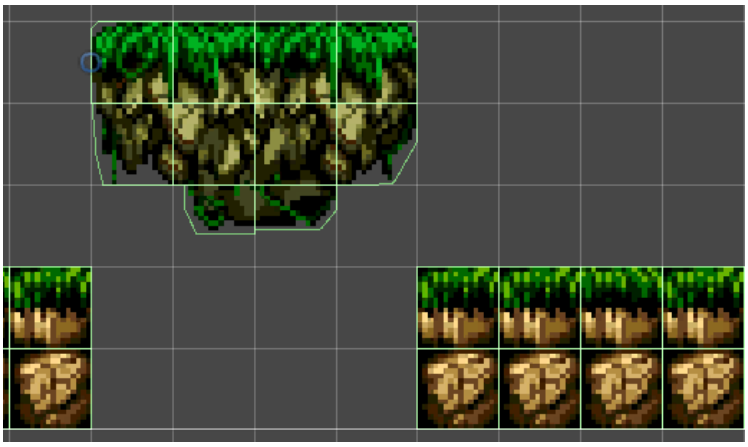
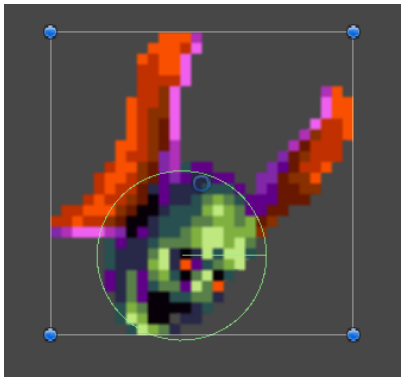


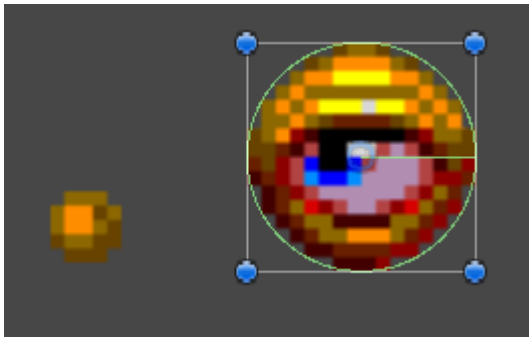
Figura 5. Un segmento del *Tilemap* como aparece dentro del editor de Unity.

- *MeleeWeapon* y *RangedWeapon*: Superclases que determinan el comportamiento base de los dos tipos de arma implementadas dentro del prototipo, armas de cuerpo a cuerpo y armas a distancia.
- *MeleeTorch*, *MeleePunch*, *RangedGun* y *PlayerProjectile*: Objetos correspondientes a las armas utilizadas por el personaje para atacar a los enemigos. Cabe destacar que *MeleePunch* no utiliza un *SpriteRenderer*, ya que corresponde al puño del personaje, por lo que no es necesario utilizar un *Sprite* adicional. *RangedGun* al ser un arma a distancia no realiza daño directamente, en cambio crea instancias del proyectil *PlayerProjectile*, el cual infringe daño a los enemigos. Los objetos responsables de causar daño a los enemigos detectan una colisión con estos mediante el método *OnTriggerEnter2D()* de Unity.

Figuras 6 y 7. Los objetos *MeleeTorch*, *RangedGun* y *PlayerProjectile* como aparecen en el editor de Unity, al ser instanciados por *Player*.

- *ChaserEnemy*, *ShooterEnemy* y *EnemyProjectile*: Objetos correspondientes a los enemigos implementados en el prototipo, y al proyectil disparado por uno de estos. Estos detectan colisiones con el *Player* para luego causar daño a este mediante el método *OnCollision2D()* de Unity.





Figuras 8 y 9. Los objetos *ChaserEnemy*, *EnemyProjectile* y *ShooterEnemy* dentro del editor de Unity.

3.4.Implementación del personaje principal y su movimiento

Teniendo ya decidido el género del juego y los requerimientos principales para el juego, lo primero fue implementar los movimientos básicos del personaje principal, es decir que pueda moverse libremente de forma horizontal, y que pueda saltar para moverse de manera vertical. Con estas dos capacidades básicas el movimiento del personaje está mayormente concretado, para después afinar detalles y llegar a un punto donde se sienta natural de controlar. Considerando el requerimiento de que el controlar al personaje debe ser fácil e intuitivo, y teniendo en cuenta la experiencia de usuario para los jugadores, se puso mayor énfasis en que los movimientos de este sean rápidos y responsivos a las acciones del jugador, es decir que al momento de tratar de mover al personaje, este realice el movimiento indicado lo antes posible. Para implementar estas capacidades primero es necesario crear un *GameObject* que corresponda al personaje del jugador, de aquí en adelante llamado *Player*. Una vez creado es necesario añadirle algunos componentes básicos como *SpriteRenderer* [14], el cual despliega un elemento gráfico, llamado *Sprite*, en la posición correspondiente al *GameObject*, y otros necesarios para que el personaje pueda interactuar con su entorno y otros *GameObjects* de forma correcta, los cuales son *RigidBody2D* [15], necesario para que el *Player* responda a las simulaciones físicas de Unity como fuerzas y gravedad, y *CapsuleCollider2D* [16], que facilita detectar colisiones con otros *GameObjects* para así poder interactuar con estos.

Teniendo ya estos componentes, el paso siguiente es hacer un *Script* que maneje el movimiento del personaje de acuerdo al input del jugador, y añadir este al *Player* como un componente. Este *Script*, llamado *PlayerController*, maneja como se controla al personaje, usando funcionalidad provista por Unity, se detecta cuando el jugador presiona algún botón asignado a uno de los movimientos del *Player*, y se realiza la acción correspondiente.

En particular, se manejan las siguientes acciones:

- Movimiento horizontal: Cuando el jugador oprime un botón correspondiente a una dirección horizontal, las cuales son por defecto las teclas A para izquierda y D para derecha, o las flechas direccionales, el *Script* le dice a Unity que ejerza una fuerza en esa dirección al personaje, lo cual hace que este acelere y eventualmente comienze a moverse en esa dirección. La magnitud de esta fuerza determina qué tan rápido llega el personaje a su velocidad máxima, y puede ser modificada mediante un parámetro dentro del Script llamado *xAcceleration*, y fue importante testear distintos valores para este hasta encontrar uno que resulte en que el movimiento del personaje produjera un resultado satisfactorio. Además al mismo momento se chequea a que dirección está mirando el personaje, y si la dirección presionada por el jugador es distinta a esta, se hace que el personaje mire a esta nueva dirección. Otro parámetro importante es *maxXSpeed*, que determina la velocidad horizontal máxima del personaje, la cual es necesaria para que el personaje no siga acelerando infinitamente, y funciona seteando el valor actual de la velocidad horizontal a dicho valor fijo cuando el personaje trata de ir más rápido que este. Finalmente, se implementó un sistema que aplica roce con el suelo para que el personaje desacelere rápidamente cuando el jugador deje de presionar en una dirección, para evitar que el personaje siga moviendose cuando no se desee o que “patine” por el suelo. Esto se hace fácilmente gracias al parámetro *Drag* del componente *RigidBody2D* de *Player*, el cual determina la magnitud del roce en las simulaciones de física del personaje, y para obtener el efecto deseado basta con aumentar este valor en las condiciones ya mencionadas.

- Movimiento vertical: Al momento que el jugador presiona el botón correspondiente a saltar, se realiza primero un chequeo para determinar si el personaje se encuentre tocando al suelo, el cual se realiza mediante el uso de un *GameObject* llamado *GroundCheck*, que se encuentra siempre posicionado a los pies del personaje, y dentro del *Script* se genera un círculo en esta posición, y se chequea si algún objeto denominado como suelo se encuentra dentro de este, y de ser así se considera que el personaje está tocando al suelo.

  De esta manera, cuando el jugador intenta saltar, y este chequeo resulta positivo, se procede a hacer que el personaje salte. Esto se hace de manera similar al movimiento horizontal, ejerciendo una fuerza al personaje pero esta vez en el eje vertical, causando que acelere hacia arriba rápidamente. La magnitud de esta fuerza es determinada también por un parámetro, en este caso llamado *jumpForce*.

  El código del *Script PlayerController*, que maneja la gran mayoría del comportamiento descrito, se puede encontrar en su totalidad en el Anexo 2.

3.5.Implementación del espacio de juego

Junto con implementar al personaje principal, fue necesario implementar a su vez un espacio por donde este pudiera desplazarse, para así poder testear el correcto funcionamiento del *Script*. En primera instancia fue suficiente generar un nuevo *GameObject* con solamente dos componentes, un *SpriteRenderer* con un sprite de una plataforma, y un *RigidBody2D* para que Unity lo trate como sólido y así el *Player* pueda interactuar con la plataforma. Para poder cumplir con el objetivo de generar una experiencia captadora y entretenida para el jugador, hace falta tener niveles más elaborados en términos del terreno que estos presentan, por lo que más adelante se implementaron las plataformas de manera diferente. Esto se hizo utilizando un sistema de Unity llamado *Tilemap* [17], el cual es utilizado para generar el terreno de los niveles de forma más rápida y eficiente, permitiendo así implementar niveles más complejos e interesantes con mucho menos esfuerzo.
Este funciona a través de 2 *GameObjects* que funcionan en conjunto, un *Grid*, que subdivide internamente a la totalidad de la escena en múltiples celdas, y *Tilemap*, que coloca plataformas o segmentos de estas dentro de las celdas, para luego manejar en conjunto todas estas plataformas dentro de un solo objeto. Unity incluye además un editor de *Tilemap* que permite agregar rápidamente plataformas dentro del *Grid*, lo cual facilita considerablemente el proceso de crear y editar el terreno de los niveles.

3.6.Implementación de enemigos

Una vez implementado al personaje y las plataformas necesarias para crear un nivel, se procedió a implementar enemigos para poblar dicho nivel y dar un desafío al jugador, lo cual es importante para mantenerlo interesado y motivado en el juego, creando así una buena experiencia de juego. Para comenzar se decidió implementar dos tipos de enemigos simples, uno que persiguiera al *Player* cuando este se encuentre a cierta distancia del enemigo, y otro que dispare un proyectil en dirección al *Player* cuando este se acerque. Ambos enemigos poseen mayormente los mismos componentes, *SpriteRenderer*, *RigidBody2D*, y *CircleCollider2D*, diferenciándose entre ellos según sus *Scripts*, *ChaserEnemyBehavior* para el primer enemigo, y *ShooterEnemyBehavior* para el segundo, los cuales determinan las acciones que realizan.

El comportamiento del primer enemigo se logra calculando la distancia que existe entre este y el *Player* para un momento dado, y en caso de que esta sea menor a su rango de detección, determinado por el parámetro *detectionRange*, se aplica una fuerza al enemigo para hacer que este se mueva en dirección al *Player*. En caso de que *Player* no se encuentre dentro de este rango, se aumenta el roce del enemigo, para que este se detenga y no siga persiguiendo al *Player*.

El segundo enemigo funciona disparando al *Player* cuando este se encuentra dentro de cierto rango del enemigo, considerando además un timer interno, determinado por el parámetro *shootDelay*, para evitar que el enemigo dispare demasiado seguido. El proceso de disparar se compone de distintas partes, y la más importante es la primera de estas, que consiste en crear una nueva instancia de un *Prefab*, correspondiente al proyectil que va a disparar el enemigo, y que determina su comportamiento una vez sea disparado mediante un *Script*. Luego, se calcula la dirección a la cual disparar el proyectil, que corresponde a la dirección entre el enemigo y el *Player*, para después setear la velocidad del proyectil en esta dirección y con una magnitud determinada por el parámetro *projectileSpeed*. Como ya se mencionó, el *Prefab* del proyectil incluye su propio *Script*, *EnemyProjectileBehavior*, y este maneja eventuales colisiones con otros objetos, verificando si estos corresponden al personaje o al terreno del nivel, destruyéndose en ambos casos. El proyectil también se destruye una vez pasado cierto tiempo o cuando sale de los límites de la cámara, para evitar que queden instancias de proyectiles innecesarios gastando recursos.

Los *Scripts* relevantes, *ChaserEnemyBehavior*, *ShooterEnemyBehavior*, y *EnemyProjectileBehavior*, se encuentran en los anexos 3, 4, y 5 respectivamente.

3.7.Implementación de ataques y sistema de vida

Una vez implementados el personaje y los enemigos, se pasó a implementar un sistema para manejar ataques del personaje hacia los enemigos, para así hacer daño a estos y derrotarlos. Para esto se creó un nuevo *Script* que agregar al personaje, llamado *PlayerAttack*, y otro que se añadió tanto al personaje como a los enemigos, *HealthManager*.

El script *HealthManager* es bastante simple, basta con que este almacene dos valores relacionados a la vida del *GameObject*, *maxHealth*, que determina el máximo valor que el personaje o enemigo puede tener, y *currentHealth*, que corresponde al valor actual que tiene la vida, el cual varía dependiendo del daño que recibe. Finalmente, el script se encarga de destruir al *GameObject* cuando su vida es igual a cero, permitiendo así al jugador poder derrotar a los enemigos una vez que haga suficiente daño a estos, y viceversa.

El script *PlayerAttack* se encarga de controlar los ataques del personaje de acuerdo al input del jugador. Cuando este presiona el botón para atacar, se verifica primero que no se está realizando ya un ataque, y de ser así se comienza la rutina para generar el *GameObject* correspondiente al ataque. Primero se determina la posición en la cual se debe posicionar el arma en relación al personaje, lo cual se hace fácilmente gracias a un *GameObject* auxiliar llamado *HandPosition* que se encuentra ubicado sobre la mano del personaje, así que basta con obtener la posición de este objeto y crear el arma sobre este.

Luego de esto, se le indica al arma cuánto daño debe realizar y cuánto debe durar, y el resto del comportamiento se controla dentro del *Script* correspondiente al arma, *MeleeAttackBehavior* o *RangedAttackBehavior*, dependiendo de qué tipo de arma se tiene seleccionada, armas cuerpo a cuerpo o armas a distancia.

Este primer *Script* maneja el proceso de realizar daño a los enemigos cuando el personaje usa armas de cuerpo a cuerpo, en el caso del prototipo corresponden a puños y a una antorcha. El script funciona detectando cuando el arma colisiona con un enemigo, verifica que esta es la primera vez que se encuentra con este enemigo en particular, para así evitar que un arma dañe varias veces a un mismo enemigo. Luego de esto, se llama al método *TakeDamage()*, del Script *HealthManager* del enemigo, y se le indica que reciba una cantidad de daño correspondiente a la fuerza del arma. El *Script* además se encarga de eliminar la instancia del arma cuando el ataque termina, lo cual se hace mediante una cuenta regresiva que se inicia al crearse el arma, y una vez terminado el periodo indicado por la variable *lifespan*, el *GameObject* correspondiente al arma se destruye.

El segundo *Script* se encarga del proceso correspondiente a las armas a distancia, que en el caso del prototipo corresponde a una pistola. El funcionamiento de este tipo de ataque comienza generando una nueva de instancia de un proyectil, luego estableciendo su posición, velocidad y dirección, y el resto del comportamiento se maneja dentro de su propio *Script* asignado al *prefab* del proyectil, *PlayerProjectileBehavior*. Este tiene un comportamiento similar al *Script* usado por los proyectiles generados por los enemigos que disparan al *Player*, pero con la diferencia de que este *Script* detecta colisiones con enemigos, y aplica daño a estos de la misma manera que las armas cuerpo a cuerpo, es decir llamando a la función *TakeDamage()* del *HealthManager* correspondiente al enemigo, y aplicando la cantidad de daño correspondiente. Una vez que el proyectil detecta una colisión con el enemigo y aplica el daño, este se autodestruye. Esto también ocurre si colisiona con el terreno del nivel, o una vez terminado su periodo de vida.

Los Scripts relevantes, *HealthManager*, *PlayerAttack*, *MeleeAttackBehavior*, *RangedAttackBehavior*, y *PlayerProjectileBehavior*, se encuentran en los anexos 6, 7, 8, 9 y 10, respectivamente.

3.8.Implementación de la cámara

En Unity, cada escena en un proyecto contiene un *GameObject* correspondiente a la cámara por defecto, y este utiliza *Components* especiales que permiten determinar qué elementos de la escena se deben mostrar en la pantalla del jugador mientras esté jugando. Sin embargo, para casi todo tipo de juego no basta con tener una cámara estática, siendo necesario que esta pueda seguir al *Player* mientras este se mueve dentro de los niveles, por lo cual es fundamental el agregar un *Script* a la cámara que determine su comportamiento, estableciendo las reglas de cómo debe seguir los movimientos del personaje, para así poder mostrar adecuadamente los elementos necesarios al jugador, y no generar alguna dificultad a la hora de jugar causada por no entregar información necesaria del nivel al jugador. En el caso del prototipo, esto se hizo a través del *Script CameraBehavior*, y el comportamiento deseado se describe a continuación.

En general es deseable que la cámara se enfoque en el *Player* lo más posible, pero al mismo tiempo, hacer que la cámara siga los movimientos de este exactamente no genera los mejores resultados cuando el personaje tiene que moverse con demasiada frecuencia, causando que la cámara se mueva demasiado, pudiendo llegar incluso a desorientar a los jugadores. Es por esto que dentro del prototipo se decidió optar por una alternativa que haga que la cámara solo se mueva cuando se considere necesario. Para esto se implementaron dos límites horizontales a cierta distancia de los bordes de la cámara creando así un área más limitada dentro de esta misma, designando dicha área como la zona donde el personaje puede moverse sin que la cámara siga sus movimientos. Solo una vez que el personaje trata de salir de esta área es cuando la cámara comienza a moverse para seguir al personaje.

En términos más específicos el *Script* funciona de la siguiente manera: En cada instante se chequea si la posición en el eje X actual del personaje se encuentra fuera del área designada por los límites implementados. Si se detecta que este es el caso se calcula una nueva posición deseada *desiredPosition* para la cámara, la cual corresponde a su posición actual trasladada en el eje X según la posición del personaje. Luego de calcular esta *desiredPosition*, se procede a mover la cámara de forma gradual, haciendo que su nueva posición sea el punto intermedio entre su posición actual y la posición deseada, llegando eventualmente a esta nueva posición en un instante futuro, siempre y cuando el personaje no siga moviéndose fuera de la zona permitida, dejando que la posición deseada se mantenga estática. El *Script CameraBehavior*, se encuentra en su totalidad en el anexo 11.

3.9.Implementación de bitácora folklórica

Otra parte del juego que fue implementada fue la llamada bitácora folklórica, la cual corresponde a una sección donde los jugadores pueden ver a todos los personajes basados en folklore que han encontrado durante el juego, y leer más información acerca de ellos. Esto se hizo utilizando el sistema de UI, Interfaz de Usuario, de Unity, para crear los elementos necesarios para desplegar la información de forma adecuada.

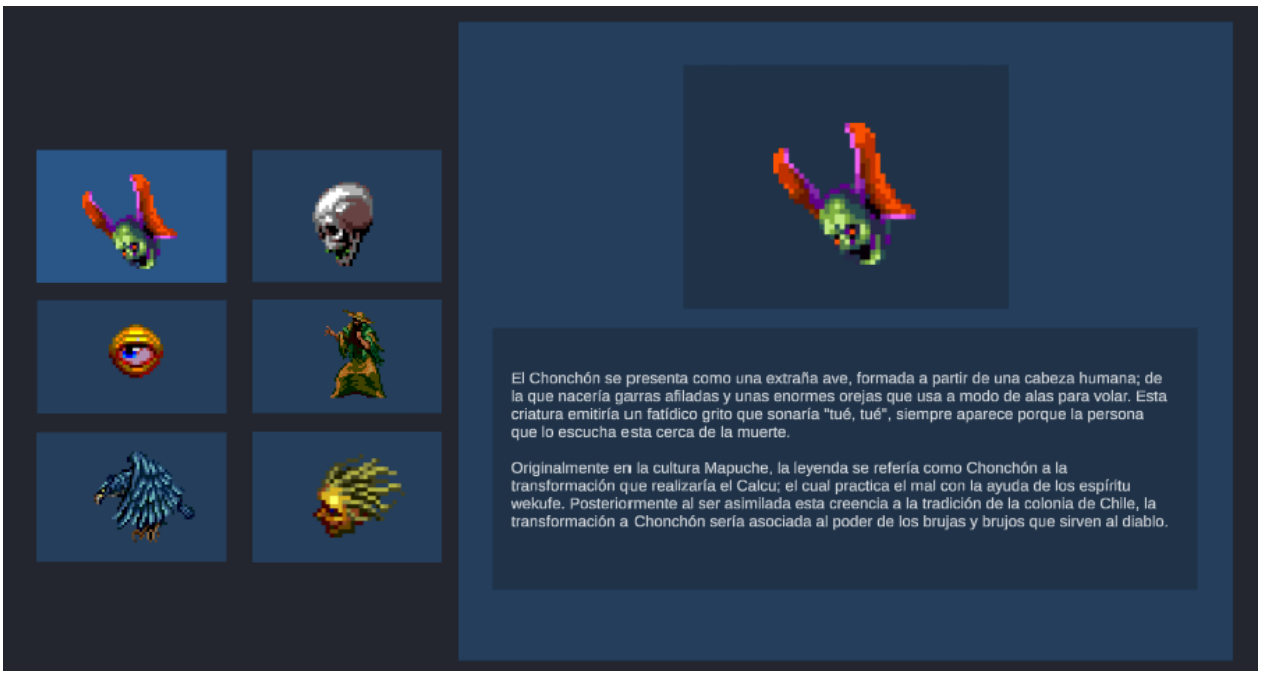


Figura 10. Primera implementación de la bitácora folklórica dentro del editor Unity.

4. Evaluación del Proyecto

Una vez desarrollado el prototipo hasta llegar a un punto que se considera apto para demostrar los elementos claves del videojuego a desarrollar, se procedió a evaluar aquello que fue logrado dentro de este. Esta evaluación fue realizada dentro de una última reunión con el Productor, Pedro Andrade, donde se mostró el trabajo realizado en detalle, y se determinó el nivel de completitud de los requerimientos dentro de las distintas partes del prototipo.

Recordando que los requerimientos establecidos para el proyecto fueron los siguientes:

- Generar una experiencia de juego entretenida y “que enganche”.
- Asegurar que el personaje sea fácil e intuitivo de controlar, debe sentirse bien al jugar.
- Entregar al jugador la opción de leer más información acerca de los personajes folklóricos que encuentra dentro del juego.

Tomando cada uno de los aspectos desarrollados dentro del proyecto, se obtuvieron las siguientes conclusiones en cuanto a la completitud de los requerimientos:

El personaje principal demuestra una buena facilidad de control, las acciones son responsivas e intuitivas, lo cual genera una reacción positiva por parte del jugador. Poner énfasis en la rapidez de respuesta de los controles del personaje fue clave para que estos se sintieran agradables.

El resto de las mecánicas implementadas dentro del juego presentan un nivel de desarrollo suficiente para demostrar el potencial que presenta el juego. Los distintos tipos de ataques y enemigos implementados ejemplifican la variedad en la jugabilidad que es posible tener dentro del juego, y el cómo estos elementos se diferencian entre ellos e interactúan unos con otros deja la puerta abierta a un sinnúmero de posibilidades a la hora de diseñar niveles dentro del juego.

Por último, la implementación de la bitácora demuestra un nivel necesario de accesibilidad para los jugadores, mostrando claramente información relevante de forma comprensiva, manteniendo una clara conexión con el resto del juego a través de la incorporación de los personajes dentro de los niveles, ayudando así a los jugadores a generar interés por saber más acerca de los personajes que estos encuentren.

Para corroborar la evaluación realizada junto al Productor, se hizo un proceso de testeo junto con la ayuda de personas externas al proceso de desarrollo, para este caso, fueron amistades no relacionadas al área de la informática. Se les dió acceso al prototipo desarrollado para que estos pudieran interactuar con este libremente, y luego se les proporcionó una breve encuesta donde estos pudieron evaluar el nivel de completitud de distintos aspectos del prototipo y del juego en su totalidad. Los resultados de esta encuesta pueden encontrarse en su totalidad dentro del anexo 12.

Por otra parte, también se realizó una evaluación de los aspectos técnicos del prototipo. El primero de estos y uno de los más relevantes para el rendimiento del juego es el manejo de recursos, ya que una mala implementación en este ámbito puede resultar en una mala experiencia para los jugadores. Dentro del desarrollo del prototipo se tuvo particular cuidado a la hora de generar nuevas instancias de *GameObjects* dentro del juego, ya sea por medio de los enemigos generando proyectiles o el personaje principal usando sus armas para atacar. Siempre se consideró el proceso de desechar correctamente estos objetos creados cuando estos ya no fueran necesarios, para así poder asegurar que no se mantuvieran referencias innecesarias gastando memoria, garantizando así que el juego nunca vaya a tratar de utilizar más recursos de los que tiene disponibles.
Utilizando una herramienta de monitoreo de memoria provista por Unity, se puede verificar que no se usa más memoria de la que se tiene asignada al prototipo dentro del *heap* en ningún momento.

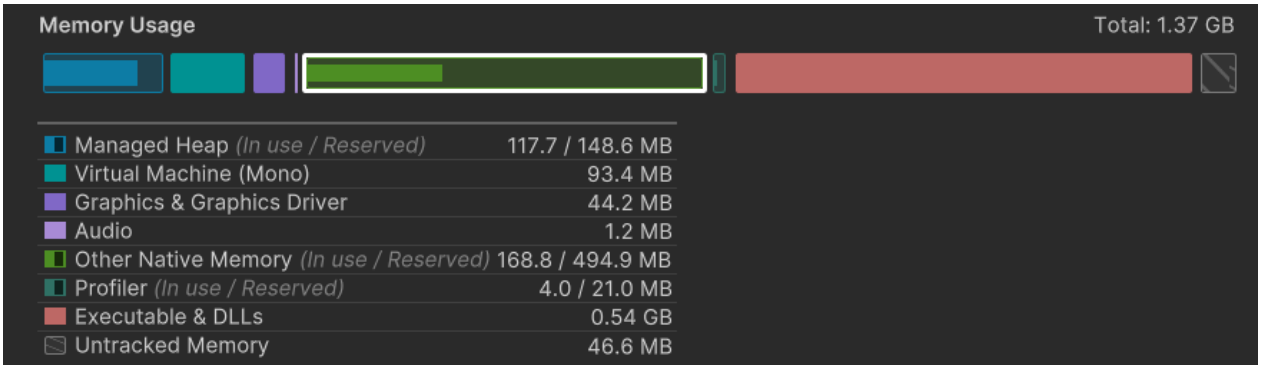


Figura 11. Reporte de monitoreo de memoria dentro de Unity. Destacar que el uso de memoria en *heap* (en azul) se encuentra bien dentro del límite asignado.

Otro aspecto que fue considerado y prontamente evaluado fue la eventual publicación del juego dentro de distintas plataformas de distribución de videojuegos. Esto se realizaría de forma bastante simple una vez que el juego se encuentre terminado, bastando con tener en cuenta los protocolos correspondientes a cada una de las plataformas elegidas para publicar el juego. En particular, se consideraron 4 plataformas principales, verificando los procedimientos necesarios para ofrecer el juego para descarga de los usuarios en cada uno de estos:

- Google Play Store [18].
- Apple App Store [19].
- Steam [20].
- Epic Games Store [21].

5. Conclusiones

Al final de este proceso, se pueden rescatar distintas conclusiones acerca del trabajo realizado. En resumen, el desarrollo de este prototipo para el videojuego planteado ha sido una experiencia beneficiosa y prometedora. A través de este proyecto, se ha logrado diseñar, desarrollar y exponer una versión inicial del juego que cumple con los criterios básicos para ser funcional y demostrar las cualidades que ofrece este videojuego a los jugadores, y el valor que este presenta como una herramienta para generar interés en la cultura y el folklore de esta zona del país.

Además, se ha podido apreciar que la creación de un prototipo es una estrategia efectiva para validar ideas y obtener retroalimentación temprana de parte del Productor. El enfocarse en los aspectos fundamentales del juego permite ahorrar tiempo y recursos, evitando la sobreingeniería y concentrándose en las características esenciales para demostrar la identidad del juego y lo que se pretende lograr con este.

El diseñar y desarrollar las mecánicas del videojuego usando las herramientas proporcionadas por Unity fue un proceso fundamental para el éxito del proyecto, y testeó las capacidades de buscar soluciones adecuadas para cumplir con los requerimientos establecidos. El desarrollo de videojuegos es un área que cuenta con un gran número de recursos y documentación fácilmente disponible, lo cual facilita considerablemente el encontrar posibles implementaciones para todo tipo de mecánicas que se deseen incorporar, por lo que el saber cómo elegir la mejor opción para un proyecto en particular se vuelve una tarea sumamente importante. Dentro de esta memoria se priorizó el implementar soluciones que fueran lo más simples posible, con el fin de enfocarse en implementar las partes fundamentales del juego rápidamente, procurando que las interacciones entre éstas sean correctas y fáciles de comprender para los jugadores, y sin dejar de lado el cumplir con los requerimientos establecidos para el proyecto.

A su vez, esta memoria ha sido una oportunidad de aplicar metodologías ágiles aprendidas dentro del currículum de la carrera, como Scrum, que a pesar de las complicaciones durante el periodo de esta memoria, sin duda han ayudado a organizar el proceso de desarrollo de manera eficiente, fomentando la comunicación y la adaptabilidad a medida que el proyecto avanzaba.

El proceso de desarrollo del prototipo también ha brindado valiosas lecciones sobre la importancia de la iteración y la mejora continua. A medida que se reciben comentarios por parte del Productor y se evalúa el rendimiento del juego, se han podido identificar más claramente las áreas de mejora y priorizar los cambios necesarios a estas para futuras iteraciones.

Aunque este prototipo es solo el punto de partida, ha sentado las bases para futuras iteraciones y mejoras del videojuego. Se ha establecido una sólida base de jugabilidad, y experiencia de usuario, que permitirá construir sobre ella y agregar nuevas características en el futuro, expandiendo la experiencia de los jugadores.

En conclusión, el desarrollo de un prototipo para este videojuego ha sido un éxito en términos de validación de ideas, obtención de retroalimentación temprana y construcción de una base sólida para el juego. Esto resulta en grandes posibilidades para el futuro del desarrollo del videojuego y continuar trabajando en el proyecto, mejorando y expandiendo el juego para ofrecer una experiencia aún más enriquecedora a los jugadores.

6. Trabajo a Futuro

Entre aquellos aspectos que quedan pendientes luego de finalizar el trabajo realizado en esta memoria, el más relevante e inmediato es el tema de la búsqueda de financiamiento. Una opción es proponer al estado el valor cultural que presenta el juego, y el beneficio que trae utilizar este medio para generar interés en un sector más joven de la población, el cual generalmente requiere mayores incentivos para interiorizarse en la cultura.

Una vez obtenido el financiamiento necesario para finalizar el desarrollo del videojuego en su totalidad, con el equipo de desarrollo adecuado, se puede proceder a terminar de desarrollar el juego usando lo avanzado en este prototipo como punto de partida.

En particular se considera como importante de implementar lo siguiente:

- Establecer un equipo de desarrollo, incorporando miembros con distintas competencias necesarias para el desarrollo, Audio, Diseño Gráfico, Diseño de Nivel, etc.
- Implementar gráficos definitivos, eligiendo un estilo visual que sea captivante.
- Finalizar la historia general del juego, y generar una secuencia de niveles en base a esta.
- Diseñar dichos niveles, incluyendo mecánicas específicas a algunos de estos, y estableciendo ubicaciones del sur de Chile para ambientar los niveles.
- Ampliar el repertorio de enemigos incluidos en el juego, considerando que criaturas o personajes folklóricos serían apropiados.
- Evaluar la posibilidad de implementar un modo de juego multijugador, incluyendo jugabilidad en línea.

- Subir el juego terminado a distintos canales de distribución apropiados, (Steam, Epic Games Store, Origin, etc).
    - A continuación se incluye un diagrama que muestra la organización de datos para el juego una vez que este sea publicado en una plataforma de distribución:

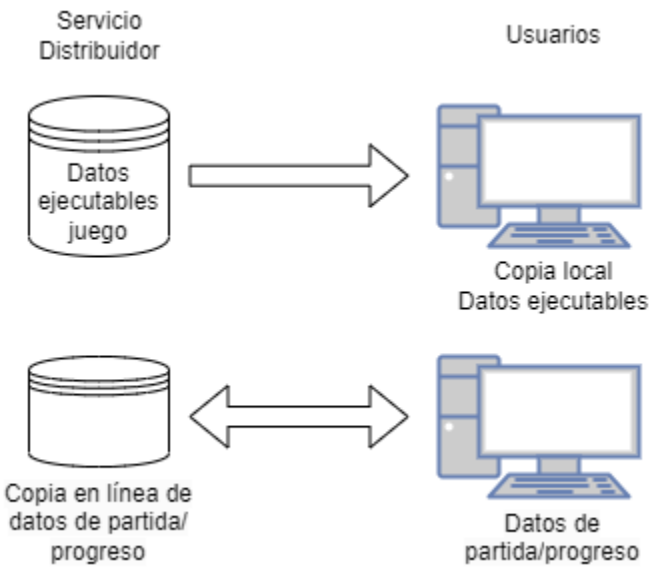


Figura 12. Diagrama de organización de datos para plataformas de distribución.

7. Bibliografía

[1]. Historia de los videojuegos - Artículo de la Facultad Informática de Barcelona. https://www.fib.upc.edu/retro-informatica/historia/videojocs.html

[2]. Video Games & Esports: Building on 2020's Rapid Growth. www.globalxetfs.com/Video-Games-Building-on-2020s-Rapid-Growth.pdf

[3]. Box2D - Librería de simulaciones para cuerpos rigidos en dos dimensiones. https://box2d.org/documentation/

[4]. Bullet Physics SDK - Librería para manejo de colisiones y simulación de física. https://github.com/bulletphysics/bullet3

[5]. FMOD - Librería de audio adaptativo. https://www.fmod.com/learn#documentation

[6]. Wwise SDK - Librería de audio interactivo. https://www.audiokinetic.com/en/library/edge/?source=SDK&id=index.html

[7]. Behavior Trees - Librería para el modelado de comportamiento de agentes. https://www.behaviortrees.com/#/dash/home

[8]. GOAP - Librería para control de personajes autónomos en juegos. https://alumni.media.mit.edu/~jorkin/goap.html

[9]. Fusion - Librería para manejo de juegos multijugador. https://doc.photonengine.com/fusion/current/getting-started/fusion-intro

[10]. Unity Multiplayer Networking - La librería oficial de Unity para manejo de interacciones multijugador. https://docs-multiplayer.unity3d.com/

[11]. Video Game Monetization Mechanisms in Triple A (AAA) Video Games. https://link.springer.com/chapter/10.1007/978-3-030-72132-9_33

[12]. Documentación de Unity - GameObject https://docs.unity3d.com/ScriptReference/GameObject.html

[13]. Documentación de Unity - Component https://docs.unity3d.com/ScriptReference/Component.html

[14]. Documentación de Unity - SpriteRenderer
https://docs.unity3d.com/ScriptReference/SpriteRenderer.html
[15]. Documentación de Unity - RigidBody2D
https://docs.unity3d.com/ScriptReference/Rigidbody2D.html
[16]. Documentación de Unity - Collider2D
https://docs.unity3d.com/ScriptReference/Collider2D.html
[17]. Documentación de Unity - Tilemaps
https://docs.unity3d.com/ScriptReference/Tilemaps.Tilemap.html
[18]. Procedimiento para publicar un juego en Google Play
https://support.google.com/googleplay/android-developer/answer/9859152
[19]. Procedimiento para publicar un juego en Apple Play Store
https://developer.apple.com/ios/submit/
[20]. Procedimiento para publicar un juego en Steam
https://partner.steamgames.com/doc/sdk/uploading
[21]. Procedimiento para publicar un juego en Epic Games Store
https://store.epicgames.com/en-US/distribution

Anexos

Anexo 1: Contexto de la historia del juego. Provisto por el Productor Pedro Andrade.

Concepción, 1904

Son cerca de las 20:00 hrs, del día 21 de julio de 1904. Las farolas de las calles de Concepción ya están prendidas y el frío es tremendo. La neblina deja ver poco más de allá de una media cuadra. El profesor Enrique Molina se encuentra preparando sus últimas cosas para salir del Liceo de Concepción, para enfrentar este frío que ya no le es desconocido. Se disponía a tomar su abrigo, cuando un golpe en la puerta, lo saca de sus pensamientos de un sobresalto. "*¿Quién será a estas horas?*", pensó, mientras extrañado se dispuso a abrir la puerta. Al hacerlo, se encontró con la figura de una mujer anciana, con el pelo gris y con arrugas que bien podrían tener más experiencia en esta tierra que toda la vida misma de Enrique. En su mano derecha, llevaba un bolso de cuero ajado y sin brillo. Sin embargo, lo que más llamó su atención, era la vestimenta de la mujer: a partir de lo que había leído del sacerdote Claude Joseph, eran los tradicionales ropajes de una machi mapuche:

Buenas noches, profesor Garmendia, perdón que lo interrumpa en su jornada laboral – le dijo la mujer – *mi nombre es María Coña, ¿puedo pasar?*

La verdad es que ya me iba, podría recibirla mañana – contestó Enrique.

Es importante, profesor, si no fuese una urgencia, no estaría aquí. Tengo una información que es muy urgente que sepa. Necesito… necesitamos de su ayuda.

Bueno, siendo así, tome asiento – dijo Enrique, moviendo una pesada silla de madera. Su curiosidad era más importante que su cansancio.

Como le decía, mi nombre es María Coña y vengo con un gran encargo. Y confiamos… esperamos… que pueda ayudarnos – respondió la mujer, mientras se acomodaba en el asiento y poniendo sus manos entrecruzadas sobre la mesa.

Creo que hay un error, Sra. María, yo solo soy un profesor y a menos que sea un tema con alguno de los alumnos, su hijo…

No, profesor, disculpe – lo interrumpió la mujer – *no se trata de eso. Necesitamos que nos ayude en una delicada situación.*

¿Pero quiénes son "nosotros"? – preguntó Enrique, ya un poco impaciente.

Bueno, profesor. Le cuento. Sé que esto puede parecer una locura, pero no lo es. Las fuerzas de lo natural y lo no natural, a veces convergen y donde menos lo esperamos. Le pido que mantenga una mente abierta.

Adelante, Sra. María, la escucho

(COMIENZA RELATO DE MARIA)

Hace poco más de 40 años, llegó un francés a nuestras tierras. Quilapán, uno de nuestros últimos grandes líderes, le permitió el paso para instalarse. Con el tiempo, conoció nuestras costumbres, pero algo le ocurrió. No sabemos qué fue, pero un día se comenzó a autodenominar como Rey de la Araucanía. Un delirio para nosotros, pero él se lo creía con más voluntad que de hechos. Quilapán por un tiempo no prestó mayor atención, pero un día este francés llegó con una corona, hecha de plata y con piedras rojas, que no conocíamos. Su mirada tenía el mal en ella y ya no era humano, era un ser extraño y sombrío. Su presencia se volvió un mal augurio. Cada pueblo por donde pasaba sufría extrañas inundaciones producto de feroces lluvias. Mucha gente moría ahogada en extrañas circunstancias, sufría ataques de cueros y otros eran atormentados por chonchones. Quilapán convocó a machis de distintas aldeas, para buscar liberar el mal que este francés representaba. Así, una noche, Quilapán convocó a este huinca en la cima del cerro más alto de Nahuelbuta, para decirle que ya no era bienvenido y que debía largarse para siempre del Wallmapu. El francés solo se rio burlonamente y dijo a Quilapán que él había llegado para quedarse y no se iría hasta que el último de los mapuches muriera bajo su manto de agua. Que ahora sería el fin del mundo de los hombres y Tren-Tren no podría ayudarlos. En ese minuto, Quilapán y sus machis comprendieron la realidad: la ambición del huinca había permitido que Kai-Kai lo poseyera. La batalla en el cerro fue feroz, ya que Kai-Kai atacó con todas sus criaturas. Las machis se defendían como podían, apelando al newén.

Al final pudimos retener a Kai-Kai, a pesar de que muchas de las nuestras murieron. Logramos volver a apresar al demonio en la corona del francés, quien luego fue acusado y devuelto a su país. Nunca lo culpamos a él, sólo fue una víctima de los engaños de Kai-Kai. Desde entonces, la corona ha quedado vigilada por los Guardianes de la Orden de la Cruz del Sur, quienes la guardaron acá, en Concepción. Lamentablemente, hace algunos días, la Secta del Sol Negro, hombres que buscan el poder absoluto, robaron la corona y buscan el renacer de Kai-Kai Vilú, para según ellos, comenzar una sociedad nueva y pura. No saben el poder maldito con el que se enfrentan. El ritual se llevará a cabo acá en los bosques que rodean Concepción, por eso necesitamos que usted recupere esa corona. En tres días será el Wetripantu. En esta fecha será el ritual de invocación. Debemos detenerlo antes de que suceda. Confiamos en usted, Profesor Molina.
(TERMINA EL RELATO DE MARÍA Y SE AGREGAN ENTRADAS A LA BITÁCORA DEL JUEGO: EL REY DE LA ARAUCANÍA, TREN-TREN, KAI-KAI, ÑIDOL LONGKO QUILAPAN, NEWEN, WALLMAPU, NAHUELBUTA, MACHI Y WETRIPANTU.)

¿Pero por qué yo? Sólo soy un simple profesor de liceo – se cuestionó Enrique

Es el newén que guía a los Guardianes de la Cruz del Sur quién lo escogió, profesor – respondió la Machi – *Su rectitud, sabiduría, solidaridad y fuerza son valores cada vez más escasos en nuestro mundo. Este podría ser el final de la humanidad como la conocemos y creemos que usted, que es un extraño a nuestras tierras, puede ser el puente de nuestras culturas. Es la voluntad de Tren-Tren.*

Es propio de los caracteres débiles el considerar las situaciones difíciles no como difíciles, sino como irremediables (Frase real de Enrique Molina). *No es mi caso. Cuenten conmigo* – respondió Enrique – *pero debe decirme como partir.*

Acá encontrará lo que necesita para comenzar su cruzada, profesor – dijo la Machi subiendo el pesado maletín a la mesa – *y creo que debería iniciarla acá cerca, en el Altacura. Se dice que hacia allá huyeron los sectarios.*

Pues entonces, no perdamos tiempo.

COMIENZA EL JUEGO.

Anexo 2: Script *PlayerController*

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

public class PlayerController : MonoBehaviour
{
    public float xAcceleration = 50f; //Que tan rapido acelera el personaje al correr
    public float maxXSpeed = 8f; //Que velocidad maxima tiene al correr
    public float drag = 20f; //Que tan rapido se detiene el personaje cuando deja de correr
    public float jumpForce = 10f; //Cuanta fuerza se le aplica al personaje al saltar -> Que tan alto salta
    public Transform groundCheck; //Componente a los pies del personaje para detectar cuando toca el suelo
    public float groundCheckRadius = 0.1f; //Que tan grande es el componente para detectar el suelo
    public LayerMask whatIsGround; //Grupo de objetos que corresponderan al suelo
    public Animator animator; //Componente que se encarga de las animaciones del personaje

    private Rigidbody2D rb; //Componente de cuerpo rigido, necesario para ubicar al personaje en el nivel y cuales son sus propiedades
    private bool isGrounded; //Variable que indica si el personaje se encuentra tocando al suelo o no

    private void Start()
    {
        rb = GetComponent<Rigidbody2D>();
    }

    private void Update()
    {
        Vector2 currentVelocity = rb.velocity;
        float xInput = Input.GetAxisRaw("Horizontal");

        if (xInput != 0f) //Si xInput no es cero, entonces el jugador esta tratando de moverse en una de las direcciones horizontales
        {
            Vector3 scale = transform.localScale; //Codigo para determinar si el personaje debe cambiar la direccion a la cual esta mirando
            if (xInput > 0f)
            {
                scale.x = 1;
                transform.localScale = scale;
            }
            else if (xInput < 0f)
            {
                scale.x = -1;
                transform.localScale = scale;
            }
            float moveAmount = xInput * xAcceleration * Time.deltaTime; //Time.deltaTime determina el tiempo que pasa de acuerdo a la velocidad que corre el juego,
            Vector2 moveDir = new Vector2(moveAmount, 0f);              //para que el movimiento del personaje sea consistente
            rb.AddForce(moveDir, ForceMode2D.Impulse);
        }

        if (Mathf.Abs(rb.velocity.x) > maxXSpeed) //Si el personaje esta moviendose mas rapido que la velocidad maxima horizontal, se reduce su velocidad
        {
            rb.velocity = new Vector2(Mathf.Sign(rb.velocity.x) * maxXSpeed, rb.velocity.y);
        }

        rb.drag = 0f;

        isGrounded = Physics2D.OverlapCircle(groundCheck.position, groundCheckRadius, whatIsGround); //Chequeo de si el personaje esta tocando al suelo

        if (isGrounded &&  Input.GetButton("Jump") == false && currentVelocity.x != 0f) //Si el personaje esta en movimiento cuando esta en el suelo,
        {
            if (xInput == 0f || Mathf.Sign(xInput) != Mathf.Sign(currentVelocity.x)) //y el jugador no esta precionando una direccion, o esta tratando de cambiar de direccion,
            {
                rb.drag = drag; //se aplica una fuerza de roce, para evitar que el personaje "patine"
            }
        }

        if (isGrounded && Input.GetButtonDown("Jump")) //Si el jugador presiona el boton para saltar, y el personaje esta tocando al suelo
        {
            rb.drag = 0f;
            rb.AddForce(Vector2.up * jumpForce, ForceMode2D.Impulse); //Se aplica una fuerza vertical al personaje para que este salte en el aire
        }
        animator.SetFloat("Xspeed", Mathf.Abs(currentVelocity.x)); //Se entregan los valores actuales de velocidad en ambos ejes al animador
        animator.SetFloat("Yspeed", currentVelocity.y);
    }
}
```

Anexo 3: Script *ChaserEnemyBehavior*

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

public class ChaserEnemyBehavior : MonoBehaviour
{
    public float speed = 5f;            //La velocidad actual del enemigo
    public float maxSpeed = 10f;        //La velocidad maxima que se desea para el enemigo
    public float stoppingDistance = 1f; //La distancia a la cual el enemigo debe detenerse al acercarse al enemigo, para que este no pase de largo
    public float detectionRange = 4f;   //La distancia a la cual el enemigo debe detectar al jugador y comenzar a perseguirlo
    public float drag = 5f;             //La magnitud del roce que se aplicara al enemigo cuando este se detenga al dejar de perseguir al jugador

    private Rigidbody2D rb;             //El componente RigidBody del enemigo
    private Transform target;           //La referencia al objetivo a perseguir, es decir, al jugador
    private Vector3 prevPosition;       //La posicion del enemigo en el instante anterior, necesario para determinar si debe cambiar la direccion hacia donde esta mirando
    private bool facingRight = true;    //La direccion a la cual esta mirando el sprite del enemigo
    private float distanceToTarget = 0f;//La distancia actual al objetivo

    private void Start()
    {
        rb = GetComponent<Rigidbody2D>();
        target = GameObject.FindGameObjectWithTag("Player").transform; //Obtener la referencia al jugador
        prevPosition = transform.position;
    }

    private void FixedUpdate()
    {
        if (target != null)
        {
            Vector3 scale = transform.localScale;                       //Verificar si es necesario dar vuelta al sprite del enemigo
            if (transform.position.x > prevPosition.x && !facingRight)
            {
                scale.x = 1;
                transform.localScale = scale;
                facingRight = true;
            }
            else if (transform.position.x < prevPosition.x && facingRight)
            {
                scale.x = -1;
                transform.localScale = scale;
                facingRight = false;
            }
            prevPosition = transform.position;

            distanceToTarget = Vector2.Distance(transform.position, target.position); //Se calcula la distancia al jugador

            if (distanceToTarget <= detectionRange)                                   //Si el jugador esta dentro del rango de deteccion
            {
                rb.drag = 0f;
                Vector2 direction = target.position - transform.position;             //Se hace que el enemigo vaya en direccion al jugador
                direction.Normalize();

                Vector2 velocity = direction * speed;
                velocity = Vector2.ClampMagnitude(velocity, maxSpeed);

                rb.AddForce(velocity - rb.velocity, ForceMode2D.Impulse);

                if (distanceToTarget < stoppingDistance)                              //Si el enemigo se acerca mucho al jugador, se detiene, para evitar que pase de largo
                {
                    rb.velocity = Vector2.zero;
                }
            }
            else
            {
                rb.drag = drag;                                                       //Si el jugador sale del rango de deteccion, se aplica roce al enemigo para que se detenga
            }
        }
    }
}
```

Anexo 4: Script *ShooterEnemyBehavior*

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

public class ShooterEnemyBehavior : MonoBehaviour
{
    public GameObject projectilePrefab;     //El Prefab del proyectil a disparar
    public float shootDelay = 2f;           //Cuanto tiempo debe haber entre cada disparo
    public float projectileSpeed = 10f;     //La velocidad a la cual debe ir el proyectil
    public float projectileLifetime = 2f;   //Tiempo maximo que debe permanecer en el aire el proyectil
    public float detectionRange = 5f;       //Rango de distancia donde el enemigo debe comenzar a disparar al jugador

    private Transform target;               //Objetivo al cual disparar, en este caso al jugador
    private float shootTimer;               //Cuanto tiempo ha pasado desde el ultimo disparo

    private void Start()
    {
        target = GameObject.FindGameObjectWithTag("Player").transform;  //Se consigue la referencia al jugador
        shootTimer = 1f;
    }

    private void Update()
    {
        if (target != null)
        {
            shootTimer -= Time.deltaTime;

            if (Vector2.Distance(transform.position, target.position) <= detectionRange) //Si el jugador se encuentra dentro del rango de deteccion,
            {                                                                            //y el enemigo no ha disparado recientemente, el enemigo dispara
                if (shootTimer <= 0f)
                {
                    Shoot();
                    shootTimer = shootDelay;                                             //Luego de disparar se inicia nuevamente la cuenta regresiva
                }                                                                        //hasta que pueda volver a disparar
            }
        }
    }

    private void Shoot()
    {
        GameObject projectile = Instantiate(projectilePrefab, transform.position, Quaternion.identity); //Crear un nuevo projectil y setear su posicion y rotacion

        Vector2 direction = target.position - transform.position;                                        //Calcular la direccion a la cual disparar
        direction.Normalize();                                                                           //en base a la posicion del jugador

        Rigidbody2D projectileRigidbody = projectile.GetComponent<Rigidbody2D>();                        //Setear la velocidad del projectil
        projectileRigidbody.velocity = direction * projectileSpeed;

        Destroy(projectile, projectileLifetime);                                                         //Destruir el proyectil una vez que pase su periodo de vida

    }
}
```

Anexo 5: Script *EnemyProjectileBehavior*

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

public class EnemyProjectileBehavior : MonoBehaviour
{
    public int damage = 10; //El daño que va a hacer este proyectil

    private void OnCollisionEnter2D(Collision2D collision) //Cuando el proyectil detecte una colision
    {
        if (collision.gameObject.CompareTag("Player") || collision.gameObject.CompareTag("Terrain")) //Verifica si fue con el jugador o el terreno, cualquier otro caso se ignora
        {
            if (collision.gameObject.CompareTag("Player"))  //Si se colisiono con el jugador, inflingir daño a este
            {
                HealthManager health = collision.gameObject.GetComponent<HealthManager>();
                if (health)
                {
                    health.TakeDamage(damage);
                }
            }
            // Destruir el proyectil cuando hace contacto con el jugador o con el terreno
            Destroy(gameObject);
        }
    }

    private void Update()
    {
        Vector2 screenPosition = Camera.main.WorldToScreenPoint(transform.position);
        if (screenPosition.x < 0f || screenPosition.x > Screen.width ||
            screenPosition.y < 0f || screenPosition.y > Screen.height)
        {
            // Destruir el proyectil cuando sale de la pantalla
            Destroy(gameObject);
        }
    }
}
```

Anexo 6: Script *HealthManager*

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

public class HealthManager : MonoBehaviour
{
    public int maxHealth = 100;         //La vida maxima que puede tener el objeto
    public int currentHealth = 100;     //La vida actual que tiene el objeto

    public void TakeDamage(int damage)  //Cuando el objeto recibe daño, reducir su vida
    {
        currentHealth -= damage;
        if (currentHealth <= 0)         //Si la vida actual del objeto es igual o menor a cero, el objeto muere
        {
            Die();
        }
    }

    public void Heal(int amount)        //Se deja abierta la posibilidad de que los objetos puedan recuperar vida
    {
        currentHealth = Mathf.Min(currentHealth + amount, maxHealth);
    }

    private void Die()                  //De momento, cuando el objeto muere solo se destruye la instancia en Unity,
    {                                   //en el futuro se puede agregar que haya una animacion antes de que se destruya u otros comportamientos necesarios
        Destroy(gameObject);
    }
}
```

Anexo 7: Script *PlayerAttack*

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

public class PlayerAttack : MonoBehaviour
{
    public int punchDamage = 5;             //La cantidad de daño y tiempo de duracion para cada ataque
    public float punchDuration = 0.25f;
    public int torchDamage = 10;
    public float torchDuration = 0.5f;
    public int gunDamage = 10;
    public float gunDuration = 0.75f;
    public float gunProjectileSpeed = 15f;  //La velocidad del proyectil de la pistola
    public GameObject punchPrefab;          //Los Prefab para cada tipo de ataque
    public GameObject torchPrefab;
    public GameObject gunPrefab;
    public Transform handPosition;
    public Animator animator;

    public LayerMask attackLayers;  //Que objetos deben considerarse para los ataques, en este caso los enemigos

    private int currentAttack = 0;  //La forma de ataque seleccionada actualmente, 0 = Puño, 1 = Antorcha, 2 = Pistola
    private float duration = 0.25f; //La duracion del tipo de ataque seleccionado actualmente, necesario para las animaciones


    private void Update()
    {
        if (Input.GetButtonDown("WeaponSwitch"))    //Cuando el jugador presiona el boton para cambiar de arma, se actualiza el tipo de arma seleccionada
        {
            currentAttack = (currentAttack + 1) % 3;
            switch (currentAttack)
            {
                case 0:
                    duration = punchDuration;
                    break;
                case 1:
                    duration = torchDuration;
                    break;
                case 2:
                    duration = gunDuration;
                    break;

            }
        }
        if (GameObject.FindWithTag("Weapon") == null && Input.GetButtonDown("Attack") && animator.GetBool("Attacking") == false) //Cuando el jugador presiona el boton para atacar,
        {                                                                                                                         //y no hay un ataque en progreso
            Invoke("Attack",0.27f);             //Se llama a la funcion para atacar, luego de un pequeño retraso para que termine la animacion
            animator.SetBool("Attacking", true);
            StartCoroutine(AnimCountdown());
        }
    }

    private void Attack() //Para cada tipo de ataque se crea una instancia del Prefab correspondiente, se determina su posicion, y se le entregan los valores necesarios a su Script
    {
        switch (currentAttack)
        {
            case 0:
                GameObject punch = Instantiate(punchPrefab, handPosition.position, Quaternion.identity);
                punch.transform.SetParent(transform);
                MeleeAttackBehavior punchBehavior = punch.GetComponent<MeleeAttackBehavior>();
                if (punchBehavior != null)
                {
                    punchBehavior.damage = punchDamage;
                    punchBehavior.lifespan = punchDuration;
                }
                break;
            case 1:
                float direction = Mathf.Sign(transform.localScale.x);
                Vector3 position = handPosition.position;
                Vector3 offsetPosition = new Vector3(position.x + 1.0f * direction, position.y, position.z);
                GameObject weapon = Instantiate(torchPrefab, offsetPosition, Quaternion.identity);
                weapon.transform.SetParent(transform);
                MeleeAttackBehavior torchBehavior = weapon.GetComponent<MeleeAttackBehavior>();
                if (torchBehavior != null)
                {
                    torchBehavior.damage = torchDamage;
                    torchBehavior.lifespan = torchDuration;
                }
                break;
            case 2:
                GameObject gun = Instantiate(gunPrefab, handPosition.position, Quaternion.identity);
                Vector3 scale = gun.transform.localScale;
                scale.x = transform.localScale.x;
                gun.transform.localScale = scale;
                gun.transform.SetParent(transform);
                RangedAttackBehavior gunBehavior = gun.GetComponent<RangedAttackBehavior>();
                if (gunBehavior != null)
                {
                    gunBehavior.damage = gunDamage;
                    gunBehavior.lifespan = gunDuration;
                    gunBehavior.projectileSpeed = gunProjectileSpeed;
                }
                break;
        }

    }
    private IEnumerator AnimCountdown() //Se utiliza una cuenta regresiva para determinar cuando debe terminar la animacion del ataque
    {
        yield return new WaitForSeconds(duration + 0.27f);
        animator.SetBool("Attacking", false);
    }

}
```

Anexo 8: Script *MeleeAttackBehavior*

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

public class MeleeAttackBehavior : MonoBehaviour
{
    public int damage; //Cuanto daño hace el ataque a los enemigos
    public float lifespan; //Cuanto tiempo debe durar el ataque antes de desaparecer
    public LayerMask attackLayers; //Que objetos se debem considerar como enemigos

    private List<GameObject> hitEnemies = new List<GameObject>();   //Lista de enemigos que ya ha atacado esta instancia,
                                                                    //para evitar atacar al mismo enemigo multiples veces con un solo ataque
    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (attackLayers.value == (attackLayers.value | (1 << collision.gameObject.layer))) //Chequear si el objeto con el que se colisiono es un enemigo
        {
            if (!hitEnemies.Contains(collision.gameObject)) //Chequear que este enemigo no haya sido ya dañado por este mismo ataque
            {
                hitEnemies.Add(collision.gameObject);
                HealthManager health = collision.GetComponent<HealthManager>();
                if (health)
                {
                    health.TakeDamage(damage); //Aplicar daño al enemigo
                }
            }
        }
    }

    private void Start()
    {
        StartCoroutine(Countdown()); //Al crearse la instancia del ataque, se llama a esta funcion para que se destruya una vez que termine
    }

    private IEnumerator Countdown()
    {
        yield return new WaitForSeconds(lifespan);
        Destroy(gameObject);
    }
}
```

Anexo 9: Script *RangedAttackBehavior*

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

public class RangedAttackBehavior : MonoBehaviour
{
    public int damage;                  //Cuanto daño hace el ataque a los enemigos
    public float lifespan;              //Cuanto tiempo debe durar el ataque antes de desaparecer (pero no el proyectil)
    public float projectileSpeed = 15f; //La velocidad a la cual debe ir el proyectil
    public GameObject projectilePrefab; //El Prefab correspondiente al proyectil

    private void Start()
    {
        Shoot();
    }

    private void Shoot()
    {
        GameObject projectile = Instantiate(projectilePrefab, transform.position, Quaternion.identity); //Crear un nuevo projectil y setear su posicion y rotacion

        Vector2 direction = transform.parent.localScale.x > 0 ? Vector2.right : Vector2.left;           // Calcular la direccion a la cual disparar

        Rigidbody2D projectileRigidbody = projectile.GetComponent<Rigidbody2D>();                      // Setear la velocidad del projectil
        projectileRigidbody.velocity = direction * projectileSpeed;

    }
}
```

Anexo 10: Script *PlayerProjectileBehavior*

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

public class PlayerProjectileBehavior : MonoBehaviour
{
    public int damage = 10;         //La cantidad de daño que va a infligir el proyectil
    public float lifespan = 5f;     //Cuanto tiempo debe permanecer en el aire el proyectil
    public LayerMask attackLayers;  //Que objetos se deben considerar como enemigos
    public LayerMask terrainLayers;

    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (attackLayers.value == (attackLayers.value | (1 << collision.gameObject.layer))) //Chequear si el objeto con el que se colisiono es un enemigo
        {
            HealthManager health = collision.GetComponent<HealthManager>();
            if (health)
            {
                health.TakeDamage(damage); //Aplicar daño al enemigo
            }
            Destroy(gameObject);
        }
        else if (terrainLayers.value == (terrainLayers.value | (1 << collision.gameObject.layer)))
        {
            Destroy(gameObject);
        }
    }

    private void Start()
    {
        StartCoroutine(Countdown()); //Al crearse la instancia del proyectil, se llama a esta funcion para que se destruya una vez que pase su periodo de vida
    }

    private IEnumerator Countdown()
    {
        yield return new WaitForSeconds(lifespan);
        Destroy(gameObject);
    }

    private void Update()
    {
        Vector2 screenPosition = Camera.main.WorldToScreenPoint(transform.position);
        if (screenPosition.x < 0f || screenPosition.x > Screen.width ||
            screenPosition.y < 0f || screenPosition.y > Screen.height)
        {
            // Destruir el proyectil cuando sale de la pantalla
            Destroy(gameObject);
        }
    }
}
```

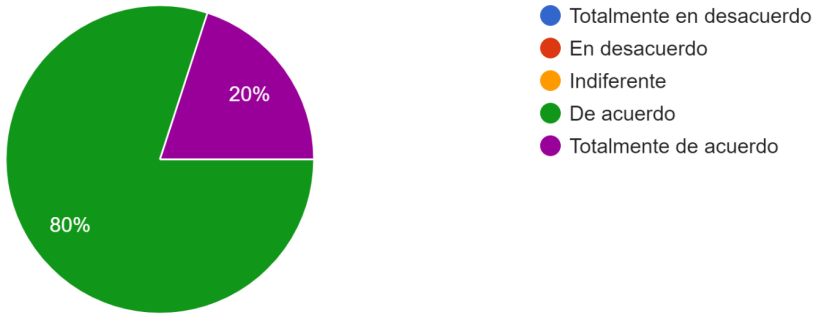
Anexo 11: Script *CameraBehavior*

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

public class CameraBehavior : MonoBehaviour
{
    public Transform target; //El objetivo que la camara va a tratar de seguir, para este caso, el jugador
    public float smoothSpeed = 0.125f; //La velocidad con la que se desea que se acerque la camara a la posicion del objetivo
    public float leftLimit = -5.0f; //Las distancias de los bordes de la camara donde se va a empezar a seguir el objetivo cuando trate de pasar por dichos limites
    public float rightLimit = 5.0f;

    private Vector3 desiredPosition; //La posicion deseada para cada momento, la cual va a cambiar solamente cuando el objetivo salga de los limites establecidos
    private bool moveCamera = true;

    private void LateUpdate()
    {
        if (target != null)
        {
            float targetX = target.position.x;

            // Verificar que el jugador este dentro de los limites establecidos
            // En caso que no lo este, determinar la nueva posicion a donde se debe mover la camara en base a su nueva posicion
            if (targetX > transform.position.x + rightLimit)
            {
                desiredPosition = new Vector3(targetX - rightLimit, transform.position.y, transform.position.z);
                moveCamera = true;
            }
            else if (targetX < transform.position.x + leftLimit)
            {
                desiredPosition = new Vector3(targetX - leftLimit, transform.position.y, transform.position.z);
                moveCamera = true;
            }
            else
            {
                moveCamera = false;
            }

            if (moveCamera) // Mover la camera hacia la posicion deseada si el jugador se encuentra fuera de los limites
            {
                Vector3 smoothedPosition = Vector3.Lerp(transform.position, desiredPosition, smoothSpeed * Time.deltaTime);
                transform.position = smoothedPosition;
            }
        }
    }
}
```

Anexo 12: Pauta de calificación para testers

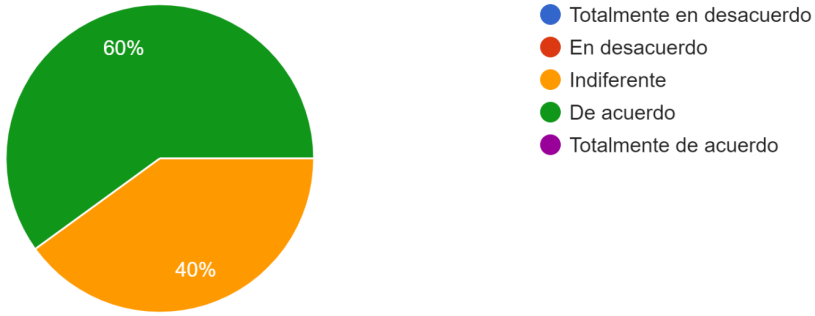
En cuanto al movimiento del personaje, considero que fue fluido y responsivo, se sintió fácil de controlar.

5 responses



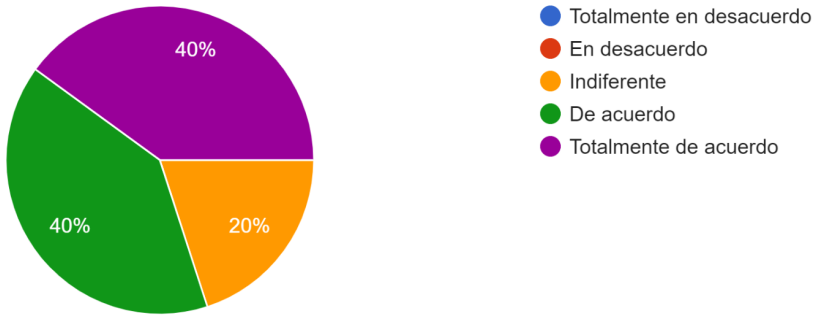
En cuanto a la implementación de los enemigos, considero que fueron suficientemente interesantes y entregan un buen nivel de dificultad al juego.

5 responses



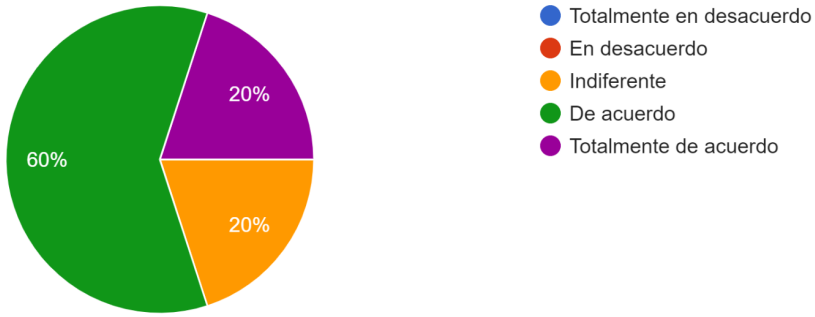
En cuanto a la implementación de la cámara, considero que se muestra correctamente la información relevante del juego, haciendo un buen seguimiento a los movimientos del personaje.

5 responses



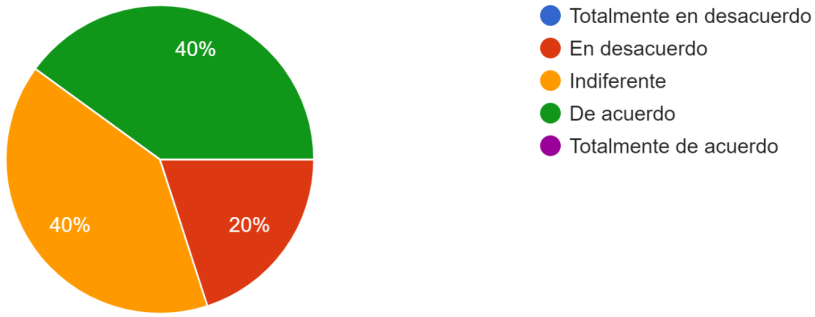
Considero que usar el medio de los videojuegos es una buena forma de captar el interés de una mayor audiencia hacia el folklor y mitología del sur de Chile.

5 responses



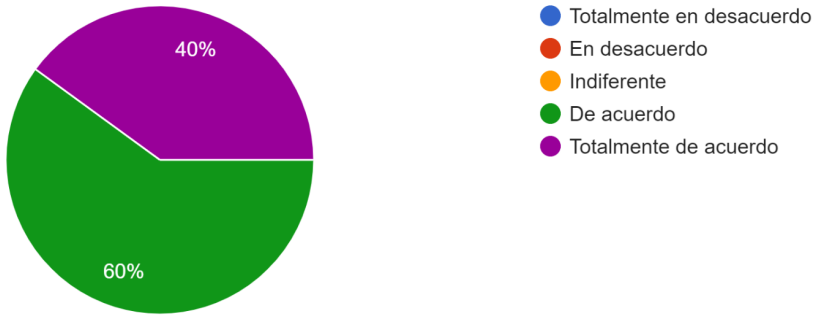
Me parece interesante la idea de usar una figura histórica como Enrique Molina de personaje principal para esta historia.

5 responses



Considero que este prototipo demuestra alto potencial, y me gustaría verlo terminado.

5 responses



Comentarios u opiniones

5 responses

Creo que es un juego interesante, me gusta la idea de usar la mitologia chilena en la historia

Me parecio atractivo el poder conocer datos interesantes sobre la mitologia del pais mediante un juego

El juego era entretenido, pero los enemigos no aportaban suficiente dificultad

Me gusta la idea de usar un videojuego para mostrar tematicas poco representadas en la actualidad como el folclor

El prototipo es simple pero entretenido, si el juego completo agrega mas contenido creo que le podria ir muy bien.