# 3. Desarrollo

## 3.1. Conceptualización y diseño

Para la conceptualización del videojuego se utilizó el modelo conceptual para el diseño de videojuegos educativos de Ruiz (2014); esta propuesta consta de 12 aspectos fundamentales a tomar en cuenta al momento de bosquejar el concepto de un videojuego serio, sin embargo, para la conceptualización del presente trabajo se descartaron los conceptos de *socialización* y *servicios*, empleando solamente diez, esto debido al alcance del videojuego que no requiere de dichos aspectos.

Esta última decisión se tomó puesto que el juego no cuenta con ningún componente de interacción para la socialización entre jugadores y por último, el juego requiere de enfocar la atención del jugador solamente en jugar, además ambos aspectos requieren de más tiempo de desarrollo y no son estrictamente necesarios para completar el juego y llegar al objetivo principal, que es desarrollar el pensamiento computacional.

A continuación, se describen cada uno de los aspectos de conceptualización y diseño del prototipo didáctico de un videojuego para el aprendizaje de estructuras secuenciales, selectivas e iterativas orientado a niños.

### 3.1.1. Mecánica

Como mecánicas del juego se tomaron en cuenta las acciones que el jugador puede realizar en todo momento, las acciones de los enemigos, las acciones de los objetos del entorno, así como las interacciones que el jugador puede tener con estos. Se separaron por entidad y se trabajaron de manera individual para luego juntar las interacciones entre ellas.

#### 3.1.1.1. Personaje principal

Se definieron diferentes estados y acciones que el jugador puede realizar con el personaje principal, estas mecánicas se dividieron en dos tipos, mecánicas de movimiento y mecánicas de acción. Las de movimiento componen los estados que el personaje principal puede tener a lo largo del juego, los cuales son el estado *quieto* y *en movimiento*; el primero se ejecuta cuando el jugador no está tocando los controles haciendo que el personaje principal se quede quieto esperando, el segundo se ejecuta cuando el jugador utiliza los controles de movimiento y hacen que el personaje principal se mueva en la dirección que el jugador indicó, estas direcciones pueden ser arriba, abajo, adelante y atrás.

Las mecánicas de acción constan de las acciones que el jugador puede realizar diferentes a las mecánicas de movimiento, las acciones son *golpear* y *recoger.* La primera se ejecuta en el momento en el que el jugador presiona el botón designado para golpear, y hace que el personaje principal lance un golpe hacia la dirección que esté mirando en ese momento, esta acción se puede realizar mientras se está *quieto* o *en* *movimiento*; la segunda acción ocurre cuando el personaje principal pasa por encima de un objeto del mapa que pueda recogerse, esta es una acción que se realiza de manera automática pero también implica que el jugador se mueva hacia el objeto.

#### 3.1.1.2. Enemigo

Las mecánicas del enemigo se realizan de manera automática y se basan en el cumplimiento de ciertas condiciones. Contrario al personaje principal, los enemigos sólo tienen mecánicas de movimiento los cuales son *quieto* y *atacando*, el primero es similar al del personaje principal con la diferencia de que sale del estado de manera automática; la segunda es similar a la mecánica de *en movimiento* en el aspecto de que el enemigo se mueve con la diferencia de que sigue al jugador en un área definida alrededor del lugar donde se encuentra. Estas dos están relacionadas ya que una lleva a la otra, cuando el jugador entra al área definida el enemigo sale del estado de espera y comienza a seguir al jugador de manera automática.

#### 3.1.1.3. Objetos

Algunos objetos del juego tienen sus propias mecánicas las cuales son activadas por el jugador, aquellos objetos con los que el jugador puede interactuar son *ítems* para pasar el nivel e *ítems* valiosos (llave de colores y diamantes respectivamente), árboles con *ítems* en la copa, botones y puertas.

Los *ítems* tienen la mecánica de ser recogidos del suelo por el jugador. Los tienen árboles la mecánica de tirar el *ítem* que se encuentra la copa, lo cual hace posible que el jugador pueda recoger este y utilizarlo para pasar el nivel. Los botones activan un mecanismo, específicamente la activación de un puente desaparecido el cual aparece cuando se utiliza el botón. El último objeto es la puerta, esta tiene la mecánica de abrirse solamente si el jugador tiene la llave correcta para abrirlo.

El personaje utiliza el *golpear* para enfrentar enemigos, tirar objetos de los árboles y presionar botones para activar el mecanismo de puente; utiliza *en movimiento* para recoger objetos tirados en el mapa, así como abrir puertas cuando se tiene la llave correspondiente; los enemigos usan *atacando* para herir al jugador si estos llegan a tocarlo.

Las mecánicas de las diferentes entidades se resumen en la **tabla** **1**.

Tabla

|  |  |
| --- | --- |
| Entidad | Mecánica |
| Personaje principal | * Movimiento: se divide en estar quieto en un lugar del mapa y moverse en una dirección especificada. * Golpear: acción que permite herir enemigos y derrotarlos, también se utiliza para tirar objetos de los árboles y activar botones. * Recoger: se usa para obtener objetos tirados en el mapa. |
| Enemigo | * Movimiento: se divide en estar quieto en un lugar del mapa y moverse para atacar y herir al jugador. |
| Objetos | Varía dependiendo del objeto.   * *Ítem*   + Recogible: puede ser recogido por el jugador. * Árbol   + Tirar *ítem*: tirar algún objeto que se encuentre en la copa del árbol. * Botón   + Presionable: puede ser presionado por el jugador y activa un mecanismo de puente. * Puerta   + Abrir: se abre dependiendo de si el usuario cuenta con la llave necesaria para dicha acción. |

La interacción de las mecánicas del jugador con los enemigos y los diferentes objetos genera la composición de la jugabilidad de todo el juego, que es recorrer el mapa derrotando enemigos y obteniendo los objetos necesarios para pasar el nivel actual.

Esta jugabilidad fue la que permitió que el juego pueda ser utilizado como un medio para el aprendizaje de estructuras secuenciales, selectivas e iterativas; los aprendizajes fueron embebidos dentro de cada nivel y su respectiva solución, sin ser evidente a simple vista para el jugador, esto con la idea de que se centrara en completar el juego y que desarrollar el pensamiento computacional fuera una consecuencia de este objetivo.

### 3.1.2. Retos

Se definieron retos principales que el jugador deberá enfrentar, estos son conseguir todas las llaves del mapa para abrir las puertas y así pasar al siguiente nivel, derrotar a los enemigos que se encuentren en el mapa para no recibir daño, evitar ser tocado por los enemigos para no perder vida, tratar de terminar el nivel lo más deprisa que se pueda y recoger objetos valiosos del mapa. Este último es opcional y no es necesario para pasar al siguiente nivel; pero de realizarlo permitirá que el jugador complete totalmente el nivel, sirve como un incentivo para explorar el mapa y completar el juego en un 100%.

El jugador deberá tener cuidado de los enemigos ya que en caso de recibir mucho daño de los enemigos se tendrá que volver a iniciar el nivel, se optó por recargar una porción de la vida total del jugador cada cierto tiempo, este se recarga cada 3 minutos. También deberá tener cuidado con el tiempo para completar el nivel debido a que los enemigos reaparecerán de manera automática después de 10 minutos de haberlos derrotado.

### 3.1.3. Escenario

Los escenarios definidos son de tipo 2D con un enfoque permanente en el personaje principal, esto quiere decir que el escenario se moverá con los movimientos del jugador, como si una cámara siempre estuviera siguiendo al jugador.

Todos los escenarios creados tienen la misma ambientación de tipo bosque, con montañas repartidas en el mapa y ríos que evitarán el paso del jugador; en las **figuras** **1** y **2** se puede observar el tipo de ambientación de los escenarios. Estos están conectados de manera que forman un mapa más grande que representa el mundo del juego.

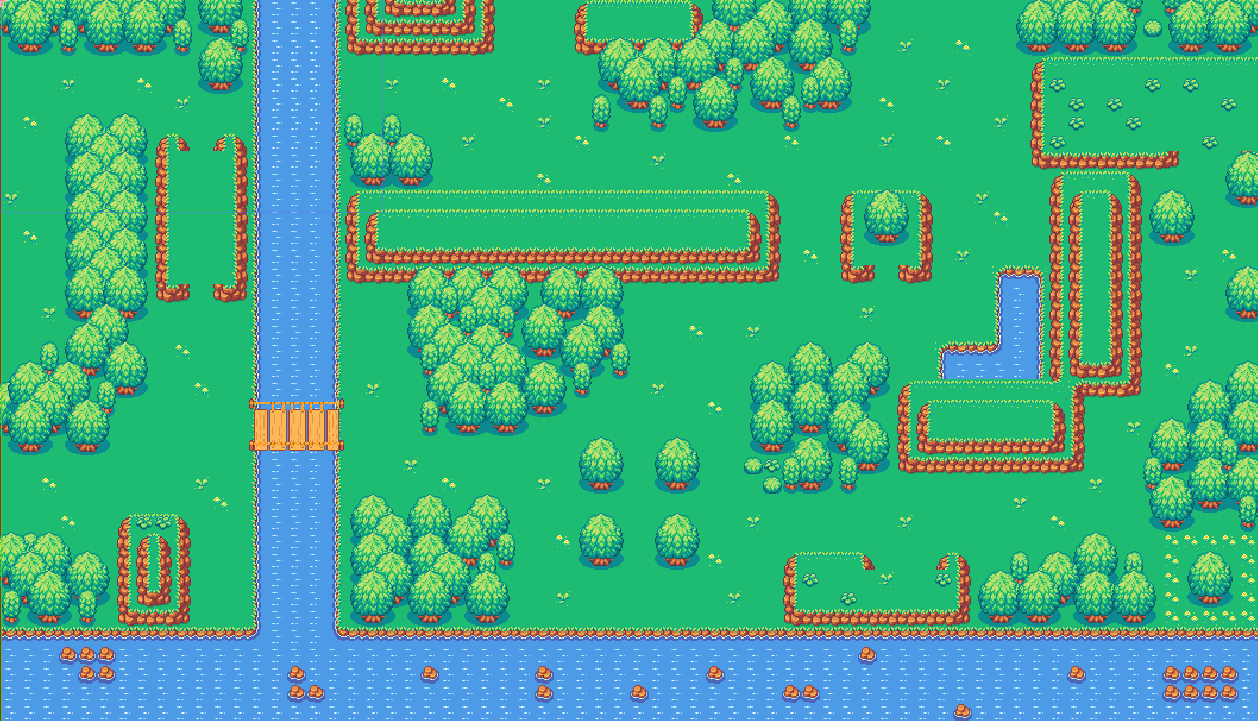
 

Figura Figura

### 3.1.4. Caracterización de personajes

#### 3.1.4.1. Personaje principal

Para el diseño del personaje principal se tomaron en cuenta características necesarias para ser visualmente agradables para el jugador, ya que es un juego dirigido a niños; se buscó un diseño que les gustara y les pareciera bonito, de esa manera estarían interesados en jugar con él.

Se indagaron diferentes opciones hasta que se encontró un diseño que cumplía con las características buscadas (véase la **figura** **3**). Este personaje se puede describir como un perrito vestido y con una capa el cual se nombró como *Shushu*.

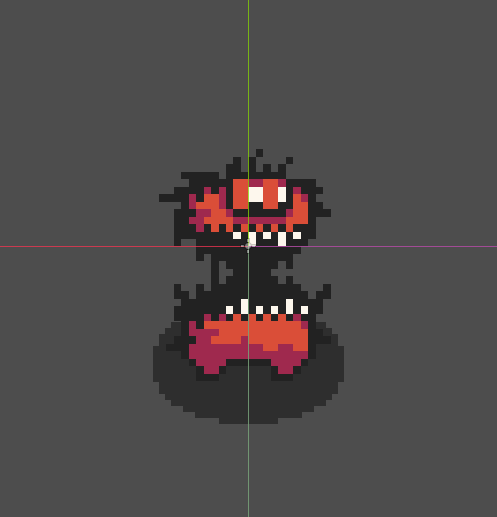


Figura

Tiene la particularidad de ser un animal con el que todos los niños están familiarizados, además de ser un diseño muy bien logrado; el perrito representa un amigo fiel con el que siempre podrán contar, que siempre los defenderá y siempre hará todo para hacerlos felices, además la capa hace que se vea más llamativo y simule ser un héroe para los niños.

#### 3.1.4.2. Enemigos

Se decidió que el diseño del enemigo fuera sencillo, pero con un diseño intimidante y que no quedara fuera de lugar con el estilo del escenario y con el diseño del jugador; después de buscar se decidió por utilizar el diseño de la **figura** **4**. Como se puede observar, el diseño es parecido a un monstruo de color rojo, esta característica también permite que resalte de los fondos del nivel y sea más sencillo localizarlo, también se eligió debido a que es un poco más grande que el jugador haciendo que sea más intimidante para él. Además, debido a la ambientación de la historia, es parte de los peligros que habitan el bosque donde nuestro personaje principal se pierde y tendrá que enfrentar para poder regresar con su familia.



Figura

### 3.1.5. *Feedback*

Se optaron como mecanismos de *feedback* los diferentes elementos de la interfaz de usuario; estos avisan al usuario sobre el estado de salud del personaje, así como los objetos recogidos a lo largo del nivel. Estos elementos se pueden observar en la **figura 5**.

**Insertar imagen de la interfaz de usuario**

### 3.1.6. *Debriefing*

Este elemento como tal no forma parte del juego, pero se decidió mostrar un mensaje para el usuario cuando finalice cierta cantidad de niveles (véase **figura 6**), esto para explicarle de manera resumida el conocimiento que acaba de aplicar para lograr pasar el nivel.

**Insertar imagen de mensaje**

### 3.1.7. *Storyline*

Se creó una historia corta que explicara y fundamentara los objetivos del juego, y permitiera que el jugador tuviera una motivación para terminar el juego. La historia se desarrolló en torno al personaje principal. La premisa de la historia es la siguiente: *un día Shushu salió con su familia a un picnic en el bosque, sin embargo, Shushu se distrajo y se separó de su familia por lo que olfateo su aroma para poder encontrarlos; lo que no esperaba es que en el camino tendría que cruzar la parte más profunda del bosque y que se encontraría con varios restos y dificultades que tendría que superar para poder regresar con ellos.*

Como se puede observar la premisa es sencilla y fácil de comprender, de esa manera se puede lograr una fácil comprensión de la situación, además de que le da al jugador una razón y explicación del juego y porqué se debería de completar.

### 3.1.8. Recompensa

Como se mencionó anteriormente en la sección de retos, el jugador tiene la opción de recoger todos los objetos valiosos del nivel, al hacer esto se le premiará con mejoras para su personaje, dependiendo de la cantidad de objetos que recoja entonces se le asignará una recompensa; se puede ver la lista de recompensas basado en el número de objetos en la **tabla 2**.

Tabla

|  |  |
| --- | --- |
| Cantidad de objetos valiosos | Recompensa |
| 9 | Armadura que reduce el daño de los enemigos. |
| 18 | Espada de metal que hace más daño y no se rompe. |

Como se pudo ver en la tabla anterior, las recompensas ayudan al jugador a pasar más fácil los niveles del juego, además de que tendrá la posibilidad de terminar el juego con el 100% de progreso.

### 3.1.9. Persistencia

La persistencia en el juego se acató mediante el típico punto de guardado automático de los juegos de tipo RPG, el guardado se realiza de manera automática cuando el jugador llega a cierta parte del nivel donde se activa el punto de guardado, esto permite que el jugador pueda retomar el juego desde un punto avanzado y no tener que empezar desde el inicio cada vez. La partida será almacenada en la memoria del teléfono celular del jugador.

### 3.1.10. Interfaz del juego e interacción

La interfaz del juego se definió con elementos que permiten que el usuario pueda revisar su estado actual (cuántos objetos ha recogido, cuánta vida le queda y cuánto durará su arma actual). La interacción se realiza mediante botones que se visualizan en la pantalla del teléfono, este se compone de una cruceta de direcciones y un botón para atacar, estos son los que el jugador utilizará para realizar cualquier acción dentro del juego. Se puede ver la interfaz definida en la **figura 7**.

**Insertar imagen de la interfaz**

## 3.2. Planificación

Esta fase del desarrollo se utilizó para medir la cantidad de tiempo dedicada a cada parte del juego, los tiempos de inicio y de finalización de cada etapa. Siguiendo la metodología SUM para el desarrollo de videojuegos, se tomaron como fases: concepto, elaboración, beta, cierre y gestión de riesgos. Los tiempos dedicados a cada etapa se pueden apreciar en la **figura 8**.

**Insertar diagrama de gantt**

## 3.3. Elaboración

En esta fase del proyecto se llevó a cabo todo el bosquejado y programación del juego, esta última se hizo siguiendo una planificación, seguimiento, desarrollo y cierre por iteraciones presente en la metodología SUM para el desarrollo de videojuegos. Cada iteración tenía una duración de dos semanas, pero también tenían cierta dependencia con los objetivos planteados en la planificación por lo que algunas veces duraron menos tiempo.

### 3.3.1. Planificación de la iteración

Este punto se ejecutó mediante el cumplimiento de objetivos las cuales representaban una parte funcional del resultado final del videojuego y que se descomponían en tareas específicas que juntas completaban el objetivo, todo esto mientras que de manera paralela y siempre presente se llevaba a cabo la fase de gestión de riesgos, con la cual se detectaban los problemas que surgieran en el momento, se evaluaban y se planteaban diferentes alternativas de solución.

El objetivo planteado en la planeación de la iteración se descomponía en tareas que se desarrollaban de manera individual y que al final se unían para generar una parte funcional del juego. Los objetivos requerían cierta cantidad de tiempo y esfuerzo dependiendo de la complejidad que implicaba desarrollarlas. La ejecución de un objetivo se iba decidiendo conforme se terminaba otra, se revisaba el estado actual del juego y se evaluaba que partes del juego seguían faltando para terminar el desarrollo, de igual manera se evaluaba cada objetivo para definir su complejidad y prioridad, si un objetivo era muy complejo entonces este se descomponía en sub objetivos que implicaban tener tareas aisladas entre sí. Los objetivos de toda la fase de desarrollo se ejecutaron en el orden que se puede observar en el diagrama de la **figura 9**.

**Adjuntar diagrama de Gantt o algún otro que liste los objetivos para el desarrollo**

### 3.3.2. Seguimiento de la iteración

Esta parte se llevó a cabo cada que se finalizaba un objetivo, esto con la finalidad de verificar como avanzar con la iteración actual; dependiendo de la finalización de los objetivos planteados para la iteración, esta podía terminar antes de completarse las dos semanas definidas para cada una, en estos casos entonces se pasaba a la siguiente planeación de iteración, para así definir los siguientes objetivos. Si era el caso contrario, y quedaban objetivos por cumplir, entonces se definían las tareas para el siguiente objetivo de la iteración y así proceder normalmente con el desarrollo de ese objetivo.

### 3.3.3. Desarrollo de características

Fue la parte más laboriosa de todo el desarrollo, ya que fue donde se llevó a cabo la programación del juego en cada iteración. En esta parte fue que se desarrolló cada objetivo planteado; a continuación, se describirá el trabajo realizado con cada objetivo.

#### 3.3.3.1. Diseño de niveles

El diseño de niveles se realizó mediante un bosquejo rápido de cada nivel que conforma el juego. Se diseñó de tal manera que al pasar un nivel se desarrollara el pensamiento computacional. Esta etapa no requirió de ningún tipo de codificación ni la creación de un diseño refinado, solamente se requería de una idea inicial para tener un punto de partida, y esta se fue refinando en la etapa de programación de nivel.

Se tomaron en cuenta la posición de inicial del jugador, la posición de la meta o salida del nivel, la posición de los *ítems* del juego, la posición de árboles normales y árboles con *ítems*, y los elementos que conforman la naturaleza que ambienta el juego. La combinación de la posición de los elementos y los árboles definen el camino que el jugador puede seguir en el nivel. Ejemplos de bosquejos del diseño de niveles se pueden observar las **figuras 10 y 11**.

Para el desarrollo del juego se diseñaron ocho niveles que el jugador tendrá que superar para terminarlo en su totalidad. Como se mencionó al principio, cada nivel fue diseñado con la intención de enseñar un tipo de estructura que fomenta el aprendizaje del pensamiento computacional.

Los niveles 1, 2 y 3 se diseñaron como niveles de introducción a la jugabilidad y mecánicas del juego, así como en la enseñanza de estructuras secuenciales; esta enseñanza se puede notar en uno de los retos del juego, que es buscar la llave para abrir la puerta que le permitirán pasar al siguiente nivel. La enseñanza se encuentra en los pasos a seguir del jugador, explorar el mapa, derrotar enemigos, recoger la llave y abrir la puerta. Estos niveles se pueden observar en las **figuras 12, 13 y 14**. Los enemigos aparecen desde el nivel 3 para introducir la mecánica de golpe al jugador, hasta el nivel final del juego donde cada vez habrán más enemigos.

**Adjuntar imágenes de los niveles**

Los niveles 4 y 5 se enfocan en la enseñanza de estructuras selectivas, y se mantuvieron presentes las estructuras secuenciales dentro de estos; se añade una situación que en los niveles anteriores no estaba presente, y es la introducción de un puente incompleto y que no se puede cruzar, esto se resuelve activando el mecanismo al presionar el botón cercano al puente que hace aparecer este y permite cruzarlo. La enseñanza de estructuras selectivas se encuentra presente en la decisión tomada por el jugador frente a esta situación donde se pregunta qué hacer si el puente está incompleto y qué hacer cuando está completo. Estos niveles se pueden observar en las **figuras 15 y 16**.

**Adjuntar imágenes de los niveles**

Los niveles 6, 7 y 8 introducen la enseñanza de estructuras cíclicas, siempre manteniendo lo aprendido en niveles anteriores, como las estructuras secuenciales y selectivas. En estos niveles hay más de una puerta, las cuales están dispuestas en un orden específico por lo que el jugador tendrá que buscar las llaves en ese mismo orden; para asegurar que el jugador recoja las llaves en el mismo orden, se bloquearon las posibles entradas que el jugador puede tomar para recoger una llave. Este camino se desbloquea cada que se abre una puerta. Las estructuras cíclicas se encuentran en la actividad de volver a buscar la llave que abre la siguiente puerta hasta que todas las puertas estén abiertas. Estos niveles se pueden observar en las **figuras 17, 18 y 19**.

**Adjuntar imágenes de los niveles**

#### 3.3.3.2. Diseño de interfaz de usuario.

En esta etapa, al igual que el diseño de niveles, se realizó un bosquejo de la parte que el jugador vería para seguir su estado actual en el nivel, este se conforma por los siguientes elementos visuales: la cantidad de vida restante del jugador, los *ítems* recogidos, la vida restante de su arma y los botones en pantalla que el jugador requiere para darle instrucciones al personaje principal.

La interfaz también contiene elementos necesarios durante el tiempo no jugable, como lo son la pantalla inicial, los menús de juego que permiten al jugador iniciar el juego y el menú de pausa. Estos se pueden ver en las **figuras 20, 21, 22, 23, 24, 25 y 26 (una imagen por cada elemento de la interfaz)**.

#### 3.3.3.3. Desarrollo de niveles.

Teniendo los bocetos de niveles definidos, se buscaron diferentes *game assets* que se adaptaran al boceto inicial. Dhule (2022) describe a los *assets* como los elementos artísticos que conforman cualquier videojuego, estos son la apariencia gráfica de los personajes, enemigos, objetos, escenarios, letras, botones, música, efectos de sonido, entre otros componentes visuales que pueden conformar el juego.

Para la programación de niveles se buscaron *assets* gráficos para representar los escenarios plasmados en el boceto anterior de diseño de niveles. Se optó por elegir un conjunto de imágenes ambientado en un bosque diseñado para juegos de tipo RPG (véase la **figura 27**). El conjunto incluye más elementos gráficos, pero sólo se utilizaron los que representan la naturaleza para el escenario (árboles, hierba, agua, ríos, rocas). Así mismo se utilizaron más paquetes que incluían otro tipo de elementos gráficos, los cuales se combinaron para generar un conjunto de elementos que se utilizaron en todos los niveles del juego.

**Adjuntar imagen del pack de assets para los escenarios**

También debido a la necesidad específica de *assets* para situaciones más puntuales se decidió crear nuevos elementos gráficos y modificar algunos otros (puentes, botones, puertas, árboles y otros objetos), algunos ejemplos se pueden ver en las **figuras 28, 29 y 30**.

**Adjuntar imágenes de assets modificados y creados para el terreno**

Habiendo elegido los *assets* que el juego utilizaría, estos se cargaron dentro de *Godot Engine* para su uso en el entorno de programación. Se utilizaron los elementos del escenario (árboles, montañas y hierba) y junto a una funcionalidad de *Godot Engine* se creó un *TileMap*, esta permite crear escenarios de manera rápida clonando los *assets* utilizados en el *tilemap*, utiliza un sistema de *clickear* en un punto del lienzo y arrastrar un *asset* seleccionado, con esto se pueden crear escenarios grandes en muy poco tiempo; y habiendo definido un bosquejo de todos los niveles, la tarea de crearlos en *Godot Engine* fue rápida. El resultado de esto se puede observar en la **figura 31**.

**Adjuntar imagen de los escenarios**

#### 3.3.3.4. Programación de personajes y objetos

Al igual que los objetos del escenario, los *assets* de los personajes, enemigos y objetos con los que el jugador puede interactuar se cargaron dentro del entorno de programación. Cabe resaltar una gran diferencia entre *assets* de personajes u objetos (como lo son los personajes principales, enemigos o cualquier otro elemento que tenga movimiento o cualquier otro tipo de animación) de los *assets* para los escenarios, ya que estos requieren de un conjunto de imágenesdonde el personaje u objeto se encuentre en diferentes posiciones para que así, con el uso de técnicas de animación típica de dibujos animados, se pueda simular el movimiento de dicho elemento. Esta particularidad hizo más complicada la tarea de encontrar *assets* que cumplieran con los criterios escogidos para los personajes y enemigos. En la **figura 32** se puede apreciar el *asset* completo del personaje principal, este se compone de muchas imágenes que juntas representan todas las animaciones que el personaje principal tiene. Cada animación requiere de una cantidad de imágenes por lo que, si el personaje tiene muchas animaciones diferentes, se traduce en más imágenes que el *asset* debe de contener.

**Adjuntar imagen del asset del jugador**

*Godot Engine* tiene dos tipos de desarrollo de juegos, el primero es mediante el método de arrastrado de bloques que se juntan con otros para hacer funcionar el juego, el segundo es mediante código de programación, el motor cuenta con su propio lenguaje llamado *GD Script* el cual guarda un gran parecido al lenguaje de programación *Python*; para toda la programación del juego se utilizó *GD Script* con un paradigma orientado a objetos, esto trata a todos los *assets* del juego como objetos con propiedades individuales e independientes, lo cual permite que cuando se modifique un objeto, todos los demás del mismo tipo tendrán las mismas propiedades y modificaciones.

##### 3.3.3.4.1. Programación de personaje principal

Primero se programaron las animaciones del personaje principal; *Godot Engine* cuenta con una herramienta para realizar animaciones con *assets* cargados en el entorno de programación. Las diferentes animaciones del personaje principal son: correr hacia adelante, hacia atrás, hacia arriba, hacia abajo; atacar desarmado hacia adelante, hacia atrás, hacia arriba, hacia abajo; atacar con un arma hacia adelante, hacia abajo, hacia arriba y hacia abajo; las posiciones de estar quieto también cuentan como animaciones dentro de *Godot Engine* por lo que se suman las animaciones de mirar hacia adelante, hacia atrás, hacia arriba y hacia abajo.

Una vez hechas las animaciones, era necesario especificar en qué momento debía reflejarse cierta animación en el juego; por ejemplo, cuando se dé la instrucción de caminar hacia adelante se debe mostrar la animación de caminar hacia adelante. Se separaron las animaciones en dos tipos para trabajar de manera ordenada en animaciones de movimiento y animaciones de golpeo; se procedió con las de movimiento que se activan con los botones de dirección adelante, atrás, arriba y abajo. Esto se hizo mediante la programación de una instrucción activada por la pulsación de una tecla en la interfaz del juego.

Cabe recalcar que las animaciones por sí solas no hacen que el jugador se pueda mover, por lo que también se agregó la instrucción en el mismo bloque de código al ejecutarse la animación. Este mismo proceso se realizó para todas las animaciones de movimiento con sus botones de dirección correspondiente. Cuando el jugador deja de presionar el botón de movimiento, el personaje regresa a su posición de quieto en la dirección de movimiento anterior, el resultado de todo esto se puede observar en las siguientes **figuras 33, 34 y 35**.

**Insertar imágenes que reflejen el trabajo descrito**

Después de programar el movimiento del jugador, se procedió con la programación de la mecánica de golpeo. Este proceso se hizo de manera similar al de movimientos, con la diferencia de que se utiliza un botón de la interfaz de usuarios dedicado para esta acción y se ejecutan las animaciones de golpeo, al finalizar el golpe la animación de quieto se ejecuta al instante; la dirección del golpeo y la dirección quieta del personaje principal dependen de la dirección de movimiento hecha antes de realizar el golpe. Esta acción se puede observar en las **figuras 36, 37 y 38**.

**Insertar imágenes que reflejen el trabajo descrito**

###### Gestión de riesgos

Terminada la programación de las animaciones principales, se notó un problema y es que cuando el jugador chocaba con un objeto del escenario ya sea este un árbol, una llave, una roca, el río o cualquier otro elemento dentro del escenario, el jugador podía atravesar estos como si no existieran. Esto se debió a la falta de *objetos de colisión* en el juego. Dhule (2022) explica las colisiones como cuerpos con una forma definida cuyo comportamiento se rige por la física del mundo real. Por ejemplo, si corremos hacia una pared sin parar, al entrar en contacto con la pared sentimos este contacto por ser ambos objetos con un cuerpo y una masa, al entrar en contacto podrían suceder diferentes cosas como por ejemplo parar de manera abrupta o podría ser posible tirar la pared; al entrar en contacto con la pared podemos decir que colisionamos con la pared o chocamos con la pared; el mismo comportamiento se aplica a los videojuegos, donde las colisiones siguen este mismo principio de sentir el contacto entre dos objetos y que algo suceda al entrar en contacto.

Habiendo identificado el problema, se procedió primero a crear objetos de colisión para todos los objetos del mapa, *Godot Engine* cuenta con un propio sistema de creación y manipulación de colisiones el cual se configura por medio de parámetros en el objeto y cuyo comportamiento al colisionar con otros objetos se define con código *GD Script*. Las colisiones de los objetos que componen los escenarios se pueden observar en la **figura 39**.

**Insertar imagen del escenario con colisiones**

Al personaje principal también se le asignó colisiones para que este pueda tener contacto con los objetos del entorno. De esta manera, con ambos elementos del juego con sus respectivos objetos de colisión y ahora contrario a la prueba anterior, el jugador ya no podía traspasar los árboles, rocas, el río y otros elementos del escenario. Los resultados se pueden observar en la **figura 40**.

**Insertar imagen del escenario colisionando con el jugador**

Otro problema surgió con este cambio y es que el jugador ahora podía salir del escenario en las partes donde no había elementos con los que el jugador pudiera colisionar (véase **figura 40**). Para resolver esto se creó un objeto de colisión que delimitó el escenario, entonces cada que se intentaba pasar los límites definidos del escenario, era detenido por una pared invisible con la que chocaba (véase **figura 41**).

**Insertar imágenes del jugador saliendo del escenario y ya no**

##### 3.3.3.4.2. Programación de cámara

Uno de los objetivos definidos en la etapa de concepto era recorrer el mapa para encontrar objetos por lo que el jugador no debería poder ver el mapa completo sino una sola parte del total de este. Para lograr eso se utilizó un recurso con el que *Godot Engine* también cuenta y es la creación de cámaras que enfocan ciertos lugares; en este caso se utilizó una cámara que seguía constantemente al personaje principal, en la **figura 42 y 43** se puede ver la cámara empleada y cómo esta sigue al jugador a todos lados.

**Insertar imágenes de la cámara del jugador siguiéndolo**

##### 3.3.3.4.3. Programación de llaves

Habiendo finalizado con los objetos del escenario y por ahora con las mecánicas del personaje principal, se procedió a programar las mecánicas de los objetos del escenario con los que el usuario podrá interactuar, estos son llaves, objetos valiosos, árboles y puertas.

Se inició con la programación de las llaves las cuales tienen la mecánica de ser recogidos por el jugador al pasar encima. Para esta mecánica se utilizó un engaño visual para el jugador, él vería como si el personaje principal aparentemente guardara la llave, pero internamente en el juego la llave deja de existir en su totalidad, es decir es eliminado del nivel actual; a pesar de esto en las propiedades del objeto del jugador se deja constancia de haber recogido la llave. Esto se ejecuta en el momento en el que el objeto de colisión del jugador hace contacto con el objeto de colisión de la llave, el mismo comportamiento aplica para todas las llaves del juego y también para los objetos valiosos y otros como las ramas y las puertas, en las **figuras 44, 45, 46 y 47** se puede observar este acontecimiento en ambos objetos.

**Insertar imágenes de los objetos recogidos**

##### 3.3.3.4.4. Programación de puertas

La diferencia de esta mecánica en las puertas es que cuando la colisión entre el jugador y la puerta se da, esta última no desaparece, sino que se reemplaza la imagen de puerta cerrada con una puerta abierta, haciendo parecer que se abre. Para que esto suceda es necesario que el jugador cumpla con la condición de haber recogido anteriormente la llave del mismo color que la puerta. Solamente entonces se abre cuando ambos objetos colisionan como se puede ver en las **figuras 48, 49 y 50**.

**Insertar imágenes de mecánica de puerta**

###### Gestión de riesgos

Otro problema se detectó al momento de haber terminado con la programación de las mecánicas de las puertas y que no estaba siendo tomado en cuenta al momento de planear este objetivo; y es que se suponía que cuando el jugador lograra abrir todas las puertas automáticamente debería de pasar al siguiente nivel, pero esto no pasaba.

*GD Script* de *Godot Engine* funciona con métodos ya definidos en el lenguaje de programación y hacen que la tarea de cambiar de niveles a otros se puede hacer sin mayor problema, sólo se requirió de una acción que ejecutara el bloque de código asignado a esta tarea y el problema se resolvió.

##### 3.3.3.4.5. Programación de árboles

###### Gestión de riesgos

En el caso de los árboles algunos tienen la mecánica de poder ser golpeados y tirar un objeto que tengan encima, esto con la finalidad de que el jugador pueda recoger este objeto y utilizarlo en el juego. Esta mecánica requirió de un elemento extra para que pudiera funcionar de la manera que se deseaba y es que era necesario que el golpe que el personaje da en el juego también tuviera una colisión para que esta pudiera reaccionar con la colisión del árbol y así reconocer que hubo un contacto con el golpe del jugador y el objeto que sufrió el golpe. La particularidad de la colisión del golpe es que solamente se activa cuando se da la orden de golpeo (**véase las figuras 51 y 52**), contrario a la colisión normal del usuario que siempre está activa debido a que es necesario para que se pueda reaccionar a todos los choques que tenga con todos los objetos del juego.

**Insertar imágenes de mecánica de golpeo**

Ya definida la colisión del golpe del jugador, se procedió normalmente con el desarrollo de la mecánica del árbol de tirar un objeto. El cual guardaba cierto parecido con la técnica empleada en la puerta donde en vez de eliminar el objeto, como ocurre con las llaves, se utilizaron dos imágenes uno donde el árbol tiene el objeto en la copa y otra donde el objeto desaparece del árbol y aparece en el suelo, esto para simular la caída (**véase las figuras 53 y 54**).

**Insertar imágenes de mecánica de caída de objeto**

##### 3.3.3.4.6. Programación de puentes

Siguiendo el desarrollo de mecánicas de objetos, se procedió con la mecánica de los puentes del juego. Al analizarlo se pudo observar como este guardaba un gran parecido con la mecánica de los árboles, ya que requiere de un golpe que active una acción que afectará al objeto; la diferencia era el lugar donde el golpe iba dirigido, la primera iba dirigido al árbol y en la segunda el golpe iba dirigido a un botón cercano al puente, al dar este golpe se emplean recursos para hacer creer al usuario que la acción se está realizando después de golpear. En el caso del puente lo que sucede después es que la parte rota del puente vuelve a aparecer lo que permite que el jugador pueda cruzarlo de manera normal (**véase las figuras 55, 56 y 57**).

**Insertar imágenes de mecánica de puente completándose**

##### 3.3.3.4.7. Programación de enemigos

En el juego sólo existe un tipo de enemigo y este es duplicado para generar muchos enemigos iguales durante todo el juego. Los enemigos, a diferencia de los demás objetos dentro del juego, tienen la capacidad de moverse, similar al jugador principal, pero estos lo hacen de manera automática y lo hacen únicamente para seguir al jugador. Debido a esta pequeña similitud se inició la programación de los enemigos de manera similar a como se hizo con el personaje principal. Cabe recalcar que no se utilizaron varias animaciones como se hizo con el personaje principal, ya que no se vio necesario profundizar en una complejidad parecida al personaje principal en este aspecto; una de las razones se debió a que los enemigos no siempre estarán enfocados en la cámara del jugador y estos serán eliminados cuando se encuentren con el jugador.

Se procedió programando las animaciones de los enemigos, estos solamente tienen un tipo de animación que es estar moviéndose en su lugar (véase la **figura 58**); esta animación se ejecuta en dos direcciones, atrás y adelante, y esta dependerá de la dirección en que el jugador es perseguido, esto se puede observar en la **figura 59**.

**Insertar imágenes de enemigo parado y en diferente dirección**

Para poder hacer que el enemigo persiguiera al jugador, se utilizó una técnica basada en un área de acción que detecta cuando el jugador ha entrado dentro del área (véase la **figura 60**), cuando él entra, se activa el movimiento del enemigo haciendo que se sienta atraído hacía la posición del jugador y lo sigue mientras se mueva o se quede quieto, de esta manera se simula la persecución del jugador (véase la **figura 61**).

**Insertar imágenes del área de acción y la persecución del jugador**

Al igual que todos los objetos del juego, los enemigos también requieren de un objeto de colisión para que cuando el jugador choque con el enemigo, se pueda producir daño hacia el jugador disminuyendo así su vida. De manera contraria cuando el objeto de colisión del golpe choque con el enemigo este último recibirá daño, disminuirá su vida y retrocederá un poco hacia el lado contrario que se estaba moviendo (véase **figura 62**).

**Insertar imágenes de las colisiones entre enemigo y jugador**

#### 3.3.3.5. Programación de interfaz de usuario y menús

##### 3.3.3.5.1. Programación de la interfaz de usuario

Habiendo concluido con la programación de mecánicas principales del juego, se procedió con la parte no jugable del juego, pero aún así muy importante para brindarle a jugador un *feedback* de lo que ocurría mientras jugaba. Esta parte fue definida en un borrador en la parte de diseño de interfaz de usuario, y de la misma manera que sucedió con el diseño de niveles se buscaron los *assets* necesarios para pasar del borrador a la programación en *Godot Engine*, los *assets* elegidos se pueden observar en las **figuras 63, 64 y 65**.

**Insertar imágenes de los assets de la interfaz de usuario**

Primero se colocó cada *asset* en su lugar dentro de la pantalla de juego (véase la **figura 66**) para después sincronizar las acciones ocurridas durante del juego con los elementos de la interfaz. Un ejemplo de esto es cuando el jugador recibe daño, el elemento que representa la vida del jugador tiene que reflejar esta pérdida de vida (ver **figura 67 y 68**).

**Insertar imágenes de feedback de daño y cambios en la interfaz**

Esto se pudo realizar gracias a las propiedades que se asignaron al objeto del personaje principal; como la vida restante, los objetos recogidos y la durabilidad del arma que tienen en el momento; estas propiedades fueron enlazadas con el elemento que correspondían en la interfaz de usuario y cuando estas propiedades sufrían algún cambio dentro del juego entonces el elemento también se actualizaba para reflejar este cambio y que pudiera ser visto por el jugador.

##### 3.3.3.5.2. Programación de menús del juego

Los menús como tal son elementos no jugables del juego, pero estos son los que le permiten al usuario que pueda tomar decisiones que afectarán al juego como programa en general. Al iniciar el programa permiten al jugador empezar la partida, cambiar la configuración del juego o salir del juego. Mientras se está jugando permiten al usuario realizar las acciones anteriores además de pausar el juego para evitar que la partida se siga ejecutando y así el jugador pueda navegar en el menú sin ser atacado por enemigos en la partida.

Al inicio se construyó un borrador del cómo se verían los menús en el juego, y siguiendo ese mismo borrador como ejemplo se generaron *assets* para utilizar en el diseño y la programación. En la **figura 69** se puede observar el borrador realizado junto con el *asset* diseñado en base a este.

**Insertar imágenes de borrador de menú y asset diseñado**

El juego cuenta con dos menús diferentes, el menú de la pantalla inicial y el menú de pausa del juego. Como se pudo ver en la **figura 69** estas acciones son *Iniciar una partida nueva*, *Cargar la última partida* y *Salir del juego*. El menú diseñado no cuenta con una opción de configuración ya que no hubo ningún elemento configurable por el usuario. Además, la principal diferencia entre ambos es que el menú de pausa sólo tiene la opción de salir del juego.

A cada botón del menú se le asignó su acción correspondiente, estos se activan en el momento en el que el jugador toca el botón desde su pantalla haciendo que se ejecute un bloque de código dedicado para esta.

La primera permite al jugador iniciar una partida nueva cada vez que lo utilice, antes de realizar la acción se pedirá al usuario que confirme esto (véase la **figura 70**). La segunda permitirá al jugador cargar la partida en el último momento de guardado que se alcanzó en el juego, de esa manera no perderá el progreso que ya lleva acumulado en el juego. La última hace que el juego se cierre del celular del usuario, para esto también se le solicitará al jugador que confirme nuevamente que realmente quiere salir del juego.

**Insertar imágenes de la confirmación del usuario**

#### 3.3.3.6. Programación de guardado del juego

El guardado del juego permite que el jugador pueda iniciar la partida en el mismo lugar donde se haya quedado, además de conservar todos los objetos que haya conseguido anteriormente. Esta característica del juego requirió de la manipulación de archivos dentro del dispositivo celular del jugador.

Se definió un archivo donde todos los objetos y estados del jugador fueran guardados. De manera interna en el archivo esto se hizo con una estructura *JSON* debido a que *Godot Engine* tiene métodos para manipular este tipo de estructura de datos, esta estructura se puede visualizar en la **figura 71**. El archivo se puede visualizar como si fuera una caja que almacena todos los objetos y estados del jugador en un momento específico del juego.

**Insertar ejemplo JSON del archivo**

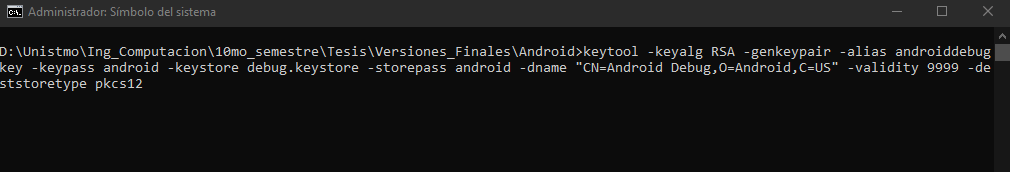
Debido a que leer y escribir sobre el archivo no era lo más eficiente se optó por un sistema de guardado automático que ejecutaba el código dedicado al guardado del juego cuando el jugador consiguiera una llave, un objeto valioso, abriera una puerta o consiguiera pasar el nivel. En estos momentos específicos se escribía sobre el archivo y se almacenaba la información del jugador.

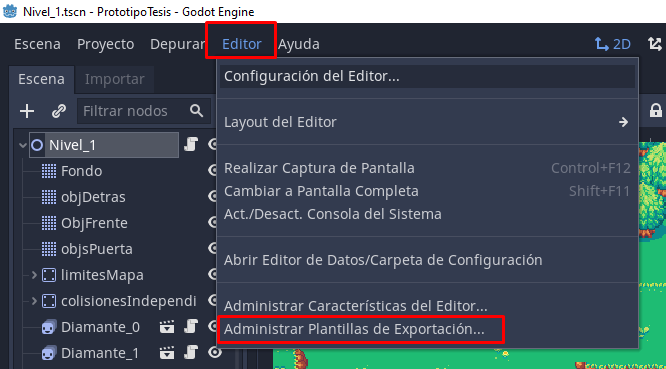
Para iniciar el juego en este punto donde se guardó por última vez, también se utilizó un bloque de código que permitía la lectura del archivo con los datos almacenados anteriormente. Este también se ejecutaba en momentos específicos del juego cuando el jugador seleccionara la opción en el menú de juego y cuando la vida del jugador se terminaba. Entonces se lee el archivo y se procede a cargar los datos devolviendo al jugador al último punto de guardado.

#### 3.3.3.7. Exportación de Godot a Android

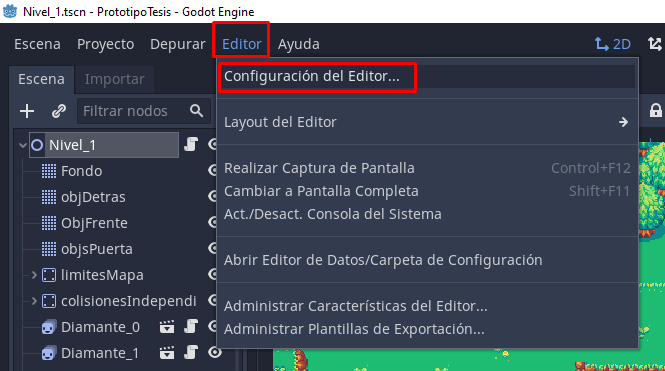
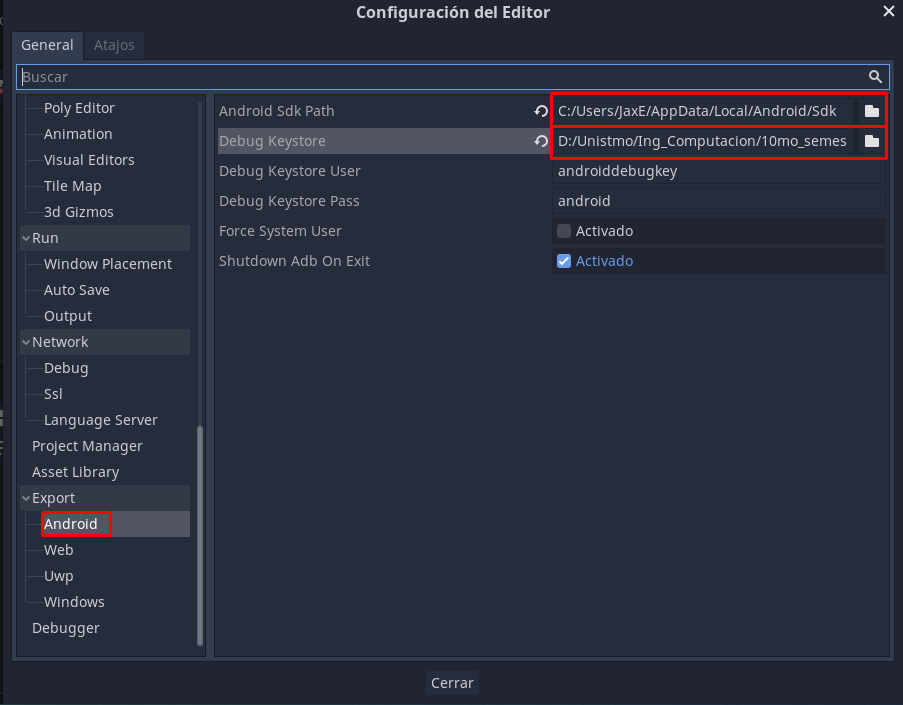
Habiendo terminado la programación del videojuego era necesario un último paso, el cual consistía en que el juego pudiera ser ejecutado en dispositivos móviles con *Android*; ya que, así como estaba, este solamente podía ser ejecutado directamente en *Godot Engine* y sin el programa no se podía jugar.

La documentación oficial en línea de *Godot* explica cómo realizar la exportación a *Android*. Fue necesario crear un archivo *debug.keystore* para poder realizar la instalación de la aplicación exportada a un dispositivo móvil. Esto se realizó ejecutando, dentro de la carpeta que contendrá la aplicación, el siguiente comando en la consola de comandos de *Windows*: *keytool -keyalg RSA -genkeypair -alias androiddebugkey -keypass android -keystore debug.keystore -storepass android -dname "CN=Android Debug,O=Android,C=US" -validity 9999 -deststoretype pkcs12*. Este paso se puede observar en la **figura 72**.

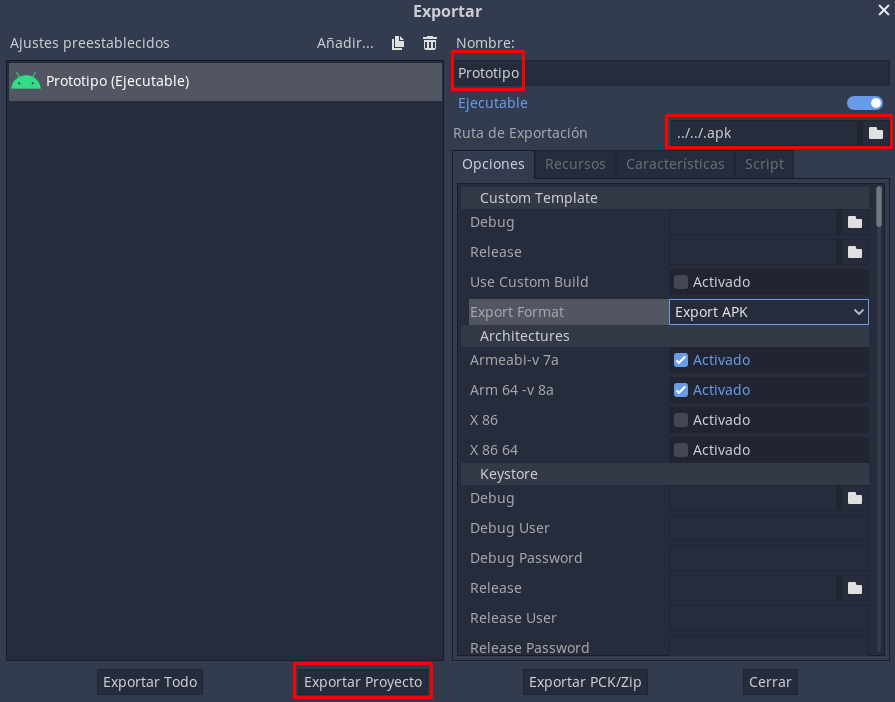
Después de crear el archivo se configuró el entorno de desarrollo *Godot* para poder ejecutar el archivo. Se descargaron las plantillas de exportación necesarias para que *Godot Engine* pueda realizar la exportación. Para ello se accede a la opción Editor, se selecciona la opción *Administrar plantillas de exportación* (véase **figura 73a**) y se da clic en *Descargar e instalar* (véase **figura 73b**), automáticamente empezará a descargar las plantillas requeridas para la exportación del juego.

Ahora se establecen las rutas del *SDK* así como el archivo *debug.keystore*. Para ello se accedió a la configuración del editor (véase **figura 74a**), se seleccionó la opción Androidy se especificó la ruta donde se instaló el *SDK* de *Android* y elarchivo *debug.keystore* (véase **figura 73b**).

Con esta configuración realizada, ahora se pudo exportar el juego a la plataforma de *Android*. Para ello se siguieron los siguientes pasos; desde la opción Proyecto, se seleccionó la opción Exportar (véase **figura 75a**), se le asignó un nombre a la aplicación, se seleccionó una ruta donde el archivo exportado estaría almacenado y se seleccionó la opción Exportar Proyecto (véase **figura 75b**).

Estos pasos finales generaron un archivo con extensión *APK* nativa de aplicaciones para dispositivos con *Android*. Este archivo se transfirió al dispositivo móvil, se instaló para así poder realizar pruebas de jugabilidad y detectar errores para su corrección.

##### Gestión de riesgos

Para la exportación del juego se encontró con un problema que no se había previsto, y es que era necesario descargar e instalar *Android Studio* para poder realizar la exportación a la plataforma de *Android*, además de descargar e instalar un paquete *SKD* de *Java* (se utilizó la versión 11 de *Open JDK*). El programa se instaló en un equipo con un procesador Intel Core i3-9100F, 32 GB de memoria RAM, 1 TB de disco duro SSD y una tarjeta gráfica dedicada NVIDIA GeForce GXT 1050 Ti, el cual pudo ejecutar el programa de manera normal y se pudo proceder con la exportación del juego.

## 3.4. Beta

### Pruebas

En esta fase el principal punto evaluado fue la jugabilidad y las mecánicas del juego; esto con la intención de detectar errores que pudieran afectar o parar el progreso del jugador mientras va avanzando en los niveles. Las pruebas fueron realizadas por dos terceros ajenos al proyecto, con el conocimiento suficiente sobre programación, videojuegos y detección de errores en programas de software.

Cada uno obtuvo una copia jugable del videojuego, así como un formato de reporte de error (véase **anexo 1**) para que llenaran con información referente al error; este formato sirvió para reportar un error encontrado en la jugabilidad o mecánicas que afectaran al funcionamiento de la versión beta del juego. El reporte contiene una captura de pantalla del error, una descripción del error y los pasos para replicar el error.

### Errores encontrados

### Resolución de errores

# 4. Referencias

Dhule, M. (2022). Adding Game Graphics. In: Beginning Game Development with Godot. Apress, Berkeley, CA. <https://doi.org/10.1007/978-1-4842-7455-2_2>