



INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

ESCOM

Trabajo Terminal

“Yolotl: un videojuego para fomentar la cultura”

2017-A035

Presentan
Hernández Bautista Yasmine Pilar

Márquez Hernández Karla Rocío

Directores
M. en C. Rafael Norman Saucedo Delgado.

Lic. Ulises Vélez Saldaña.



Noviembre 2017



No. de TT:2017-A035

17 de noviembre de 2017

Documento Técnico Parte A

“Yolotl: un videojuego para fomentar la cultura”

Presentan
Hernández Baustista Yasmine Pilar¹

Márquez Hernández Karla Rocío²

Directores

M. en C. Rafael Norman Saucedo Delgado. ***Lic. Ulises Vélez Saldaña.***

RESUMEN

En México la industria de videojuegos tiene una alta demanda de consumo; sin embargo, existen pocos estudios que desarrollen videojuegos basados en la cultura mexicana. Actualmente en México existe un fuerte desinterés en la cultura nacional. El presente trabajo terminal consiste en el desarrollo de un videojuego que fomente la cultura con temática de la cultura mexica.

Palabras clave. – Cultura mexica, desarrollo tecnológico, ingeniería de software, videojuego.

¹daughterofthewind10@gmail.com

²winged.zelda@gmail.com

Capítulo 1

Advertencia

“Este documento contiene información desarrollada por la Escuela Superior de Cómputo del Instituto Politécnico Nacional, a partir de datos y documentos con derecho de propiedad y por lo tanto, su uso quedará restringido a las aplicaciones que explícitamente se convengan.” La aplicación no convenida exime a la escuela su responsabilidad técnica y da lugar a las consecuencias legales que para tal efecto se determinen. Información adicional sobre este reporte técnico podrá obtenerse en: La Subdirección Académica de la Escuela Superior de Cómputo del Instituto Politécnico Nacional, situada en Av. Juan de Dios Bátiz s/n Teléfono: 57296000, extensión 52000.

Índice general

1. Advertencia	2
2. Introducción	5
3. Antecedentes	6
3.1. Propuesta	6
3.1.1. Planteamiento del problema	6
3.1.2. Marco Teórico	7
3.1.3. Planteamiento de la solución	11
3.2. Ajustes al trabajo terminal	11
3.2.1. Corrección del enfoque de la solución	11
3.2.2. Nueva división de trabajo	11
3.2.3. Actualizando el motor de juego	12
3.2.4. Máquina de estados	13
3.3. Contribuciones	18
3.3.1. El juego	18
3.3.2. Modelo de negocios en un videojuego	20
3.3.3. Costo de hacer un videojuego	23
3.3.4. Vicio en el videojuego	23
3.3.5. Modelo de datos	26
3.3.6. Estrategias para combatir la adicción entre los usuarios	26
3.3.7. Modelo de Negocios	27
3.4. Trabajo realizado durante trabajo terminal 1	27
3.4.1. Presentado en TT1	27
3.4.2. Etapa de Preproducción	28
3.4.3. Etapa de producción	30
4. Trabajo realizado	33
4.1. Etapa de producción de <i>Huddle</i>	33
4.1.1. Detalles del proyecto	33
4.1.2. Sprint plan 1	34
4.1.3. Sprint backlog 1	35
4.1.4. Buglist	36
4.1.5. Task Slips 1	36
4.1.6. Sprint plan 3	39
4.1.7. Sprint backlog 3	40
4.1.8. Task Slips 3	41

4.1.9. Cuarto <i>sprint</i> de producción	45
4.1.10. Sprint plan 5	47
4.1.11. Feature log 5	48
4.1.12. Sprint backlog 5	48
4.1.13. Task Slips 5	49
4.1.14. Sexto <i>sprint</i> de producción	53
4.1.15. Sprint plan 7	66
4.1.16. Sprint backlog 7	67
4.1.17. Task Slips 9	68
4.1.18. Octavo <i>sprint</i> de producción	72
4.1.19. Sprint plan 9	80
4.1.20. Feature log 9	82
4.1.21. Sprint backlog 9	82
4.1.22. Task Slips 9	83
4.1.23. Décimo <i>sprint</i> de producción	87
5. Resultados obtenidos	96
5.1. Pruebas	96
5.1.1. Sprite packer	96
5.1.2. Mecánicas del juego	96
5.1.3. Dinámicas del juego	97
5.1.4. Estética del juego	98
5.1.5. Prueba unitaria	98
5.1.6. Prueba de integración	99
5.1.7. Prueba de sistema	100
5.1.8. Prueba de rendimiento	101
5.1.9. Prueba de rendimiento	101
5.1.10. Cuestionario 4	105
5.2. Niveles terminados	109
6. Conclusiones	110
7. Anexos	111
7.1. Interfaces	111
7.2. Diseño de Personajes	111
7.3. Modelo de Datos	111
7.4. Control de adicción en el jugador	111
7.5. Maquetas de niveles	111

Capítulo 2

Introducción

En la época actual nos vemos rodeados por tecnología por todos lados, esta ya es parte de nuestra vida diaria, de nuestras actividades tanto de trabajo, de nuestro medio de comunicación, como medio de información e incluso de nuestro medio de entretenimiento. Las generaciones recientes han crecido con la evolución de la tecnología a un ritmo acelerado, a tal punto que la tecnología ya es parte de su cultura.

Dentro de la evolución de la tecnología se encuentra los videojuegos, una industria de entretenimiento. Nos daremos a la tarea de investigar los puntos de impacto que tiene. Pues a primera instancia podremos observar el tipo de personas que juegan, los ingresos que se generan en esta industria, los tipos de industria que existen, las consecuencias positivas, las consecuencias negativas que generarían, las ganancias como profesionista en esta rama y como realizar un proyecto de esta naturaleza.

Los videojuegos hacen al jugador involucrarse con varios sentidos a la vez en lo que se le presenta, así se crea una experiencia propia como cualquiera de la vida pues provoca la inmersión. El INJUVE ha dado a conocer información donde se puede observar quelos jóvenes y los videojuegos llevan una interacción diaria y sobre cualquier tema, desde educativo hasta de ocio.

Por otra parte tenemos que la sociedad ignora en su mayoría los aspectos históricos culturales propios de su país, específicamente de México. Se denota un gran desinterés por parte de la gente el siquiera conocer su legado.

Tomando en cuenta todos los aspectos anteriores, el proyecto consistirá en juntar ambas ideas para usar el videojuego como difusión del aspecto cultural mexicano. Se investigará más a fondo la historia de los videojuegos para elegir el público objetivo potencial o las personas alcanzables, los procesos y metodología para poder realizar un videojuego, aspectos a considerar para el proyecto, la cultura y ramas que abarca dentro de la sociedad, la cultura y la tecnología como se relacionan entre sí, las herramientas con las que se cuentan para la realización del videojuego, las teorías que pueden usarse, los conflictos que pueden ocurrir, como solucionar los problemas que se nos presenten y por su puesto el cambio o impacto que se tenga al presentarlo al público.

Al final se presentará un videojuego como producto con pruebas de diferentes tipos para demostrar el resultado ante la sociedad y los cambios que se presentaron en las personas.

Capítulo 3

Antecedentes

En este capítulo se presenta a manera de resumen el trabajo realizado durante la etapa del desarrollo correspondiente a Trabajo Terminal 1. Primeramente se presenta la propuesta del trabajo, es decir la definición del problemas, la definición de algunos conceptos como son el videojuego, la cultura, las metodologías de desarrollo, la definición de la solución; después se procede en presentar los ajustes que se realizan en el proyecto una vez que se concluyo el periodo del trabajo terminal 1, tales como la asignación de trabajo, el enfoque del problema y la actualización del motor de juego; seguido de esto se presentan las observaciones realizadas por los sinodales durante la presentación del trabajo terminal 1 y como fueron solucionadas. Finalmente se hace un resumen del trabajo realizado durante el trabajo terminal 1, correspondiente a la etapa de preproducción y a los dos primeros sprints de la etapa de producción.

3.1. Propuesta

En esta sección se presenta a manera de resumen las propuestas y los conceptos definidos durante el trabajo terminal 1, tales como el planteamiento del problema, conceptos y definiciones referentes al videojuego y su desarrollo, la definición y delimitación de la cultura y el planteamiento de la solución que se desarrolla durante el trabajo terminal.

3.1.1. Planteamiento del problema

En México existe un fuerte desinterés y desconocimiento hacia su cultura e historia nacional. De acuerdo con la Tercera Encuesta Nacional de Cultura Constitucional, el 52.7 % de los encuestados desconoce el año en que se aprobó la constitución nacional y no la relaciona con la Revolución Mexicana **RefConsti**. Con base en la encuesta realizada por Parametría, empresa dedicada a la investigación estratégica de la opinión y análisis de resultados, solo el 32 % de su encuestados supó que México se independizó de España, el 51 % desconoce el país del que se independizó México, mientras que el resto del porcentaje de los encuestados piensa que México se independizó de otro país que no es España; la misma encuesta realizada por Parametría señala que el 25 % de los encuestados mencionaron personajes históricos ajenos a la independencia de México como participes de ésta y el 12 % respondió no saber que personajes históricos participaron en la independencia **RefParametria**.

3.1.2. Marco Teorico

En esta sección se presentan los conceptos básicos para comprender el trabajo realizado durante el desarrollo del trabajo terminal, tales como la definición del videojuego, sus características, su clasificación, las metodologías de desarrollo, las herramientas para el desarrollo y la cultura.

Videojuego

El grupo de periodista especializado en tecnología y desarrollo de software Carricay define al videojuego como: una aplicación interactiva orientada al entretenimiento que, a través de ciertos mandos o controles, permite simular experiencias en la pantalla de un televisor, una computadora u otro dispositivo electrónico"**Ref_DefVideo**.

Al igual que con otros productos tecnológicos, la evolución de los videojuegos ha sido vertiginosa, resultando complicado mencionar características comunes para todos los videojuegos. Sin embargo, en el libro "*Marketing y videojuegos: Product placement, in-game, adevertising y advergaming*" se menciona que existen seis características comunes en los videojuegos: Interactividad, entretenimiento, jugabilidad, simulación \virtualidad, inmersión y multiplataformidad**RefCarac**; a continuación se menciona en que consisten cinco de las seis características, esto debido a que la última no se encuentra presente en todos los juegos y el mismo autor de la obra la menciona como una característica opcional a tomar en cuenta:

- **Interactividad:** En el artículo "*Defining Virtual Reality: Dimensions Determining Telepresence*" se define la interactividad como la capacidad de los usuarios para participar y modificar la forma y el contenido de un entorno mediado en tiempo real**RefInteractividad**.
- **Entretenimiento:** en el artículo "Las Tecnologías del Entretenimiento: Pasado, Presente y Futuro", el entretenimiento "se asocia, usualmente, de hacer algo que nos divierte, algo que podemos hacer solos o con otros, para entretenernos o divertirnos, en nuestro tiempo libre, o tal vez, algo que nos relaje o que nos haga reír"**RefEntretenimiento**.
- **Jugabilidad:** en el libro "*Marketing y videojuegos: Product placement, in-game, adevertising y advergaming*" se define la jugabilidad como "la relación que existe entre todas las acciones reacciones e interacciones tanto del videojugador como el videojuego como entre los propios sistemas y subsistemas programados en el videojuego"**RefCarac**.
- **Simulación \Virtualidad:** La simulación "se trata de una representación a medida cuyo objetivo nos permite interactuar y relacionarnos con lo representado según nuestros intereses"**RefCarac**.
- **Inmersión:** Con base en el libro "La vida en la pantalla: La construcción de la identidad en la era de internet", la inmersión es un proceso psicológico que se produce cuando la persona deja de percibir de forma clara su medio natural al concentrar toda su atención en un objeto, narración, imagen o idea que le sumerge en un medio artificial **RefInmersion**. Por su parte en la tesis "Libertad dirigida: Análisis formal del videojuego como sistema, su estructura y su avataridad", la inmersión se entiende como la coherencia de la ficción del juego y su aceptación por el jugador.**refInmersionNavarro**

Los videojuegos pueden ser clasificados con base a su jugabilidad, en el libro "Juego. Historia, Teoría y Práctica del Diseño Conceptual de Videojuegos"**Ref_JuegoDisenio** se propone la siguiente clasificación.

- **Juegos de acción:** Son juegos usualmente de temática violenta. El jugador lucha por su supervivencia, para ello se vale de armas o habilidades de combate.
- **Juegos de estrategia:** Para que el jugador logre sus objetivos en este tipo de juegos, éste debe de planear una estrategia, normalmente a largo plazo.
- **Juegos de Rol:** La mecánica de los juegos de rol gira en torno a un grupo de héroes, con habilidades y progresión definidos; el grupo de héroes debe de trabajar coordinadamente para cumplir un objetivo; estos héroes pueden ser controlados por un solo jugador o por varios. El jugador deberá explorar un mundo de gran tamaño haciendo evolucionar a sus personajes y sus habilidades.
- **Videojuego de aventura:** Son parecidos a los juegos de Rol; con la peculiaridad de que tienen una progresión más lineal y no se hace tanto énfasis en los combates, siendo su eje principal la narrativa.
- **Videojuegos de deportes:** Son todos aquellos videojuegos que tratan sobre deportes que no involucren la conducción de un vehículo. Pueden ser juegos sobre fútbol, fútbol americano, tenis, etc.
- **Videojuegos de carreras de vehículos:** Son todos aquellos se centran en las carreras con todo tipo de vehículos, mayoritariamente automóviles.
- **Videojuegos puzzle:** Este tipo de juego involucra la resolución de un problema a partir de la utilización de una serie limitada de recursos, por lo que si los recursos no se utilizan de la manera correcta el problema no podrá ser solucionado.

Dentro de la clasificación de los juegos de acción entran los juegos de plataforma, definidos por una jugabilidad donde el jugador debe de controlar a un personaje con el que se desplazará saltando entre plataformas y esquivando todo tipo de obstáculos y enemigos **Ref_JuegoDisenio**. Es importante que se entienda el concepto del videojuego, sus características, su clasificación y la jugabilidad básica de un juego de plataforma ya que el presente Trabajo Terminal gira entorno al desarrollo de un videojuego de plataforma.

Metodología de desarrollo

Las metodologías de desarrollo de software son un conjunto de procedimientos, técnicas y ayudas a la documentación para el desarrollo de productos software **Ref_metodología**. Para el presente trabajo terminal se consideran las siguientes metodologías como candidatas a implementar para guiar el desarrollo:

- **Metodología en cascada:** Sigue una progresión lineal por lo que cualquier error que no se haya detectado con antelación afectara todas las fases que le sigan provocando una re-definición en el proyecto y por ende un aumento en los costos de producción del sistema **Ref:CarCascada**. Esta metodología se divide en las siguientes etapas:
 - Análisis de los requisitos del software.
 - Diseño.
 - Codificación.

- **Pruebas.**
 - **Mantenimiento.**
- **Metodología en Scrum:** *Scrum* parte de la visión general que se desea que el producto alcance; a partir de esta visión se inicia la división del proyecto en diferentes módulos. *Scrum* implementa una jerarquía entre los módulos en donde los módulos de mayor jerarquía son los que se desarrollaran al inicio del proyecto o durante las primeras iteraciones o *sprints* **Ref_ScrumRef**. Cada sprint se compone de las siguientes fases:
- **Concepto.**
 - **Especulación.**
 - **Exploración.**
 - **Revisión.**
 - **CierreRef_ScrumGuia.**
- **Metodología de Programación extrema:** Es una metodología de desarrollo ágil y adaptable, soporta cambios de requerimientos sobre la marcha. Su principal objetivo es aumentar la productividad y minimizar los procesos burocráticos, por lo que el software funcional tiene mayor importancia que la documentación**Ref_XP**.
- **Metodología Huddle:** Es una metodología cuya funcionalidad se basa en la metodología *Scrum*, con la diferencia de que está orientada al desarrollo de videojuegos. De naturaleza ágil, resulta óptimo para equipos multidisciplinarios de 5 a 10 personas; es iterativa, incremental y evolutiva **Ref_Huddle**. *Huddle* se divide en las siguientes etapas:
- **Preproducción.**
 - **Producción.**
 - **Postmorten.**

Tras un riguroso análisis comparativo entre metodologías, se elige a *Huddle* como la metodología a guiar el desarrollo del Trabajo Terminal; esta elección se basa principalmente en que dicha metodología esta enfocada a videojuegos y no requiere ser adaptada por lo que se puede llevar a cabo el proyecto de manera directa sin tener que invertir tiempo en adaptar la metodología a las necesidades del desarrollo de un videojuego.

Herramientas de desarrollo

Como cualquier desarrollo de software, el desarrollo de un videojuego requiere se software especializado tal como un motor de juego, editores de imágenes, software de diseño, de edición de audio, etc. En este apartado se van a definir algunas de las herramientas utilizadas durante la elaboración del trabajo terminal.

La primera herramienta a definir es el del motor de juego. El motor de juego, también conocido como *Game Engine*, parte del concepto de reutilización; es decir, es posible generar juegos a partir de un código base y común mediante una separación adecuada de los componentes fundamentales, tal como visualización de gráficos, control de colisiones, físicas, entrada de datos etc **Ref:MotorGraf**; esto permite a quienes trabajen en un juego puedan centrarse en todos aquellos

detalles que hacen al juego único. Dentro del mercado existen diferentes opciones de motores de juego tales como *Unity3D*, *UnrealEngine* y *CryEngine*, por citar algunos. Para el presente trabajo terminal se decide por utilizar *Unity3D* ya que ofrece:

- Desarrollo multiplataforma, lo que permite aumentar la escalabilidad del proyecto.
- Curva de aprendizaje rápido.
- Comunidad de desarrolladores activa.
- Tres opciones de lenguajes de programación para utilizar: *C, JavaScript* y *Boo*.
- No requiere de muchos recursos para su instalación.
- Uso de diferentes tipos de licencia lo que permite contar con una licencia gratuita, de pago y una de negocios. No existiendo mucha diferencia de funcionalidad entre la licencia libre y la de pago.

En lo que refiere a la creación del entorno gráfico del videojuego, es decir de sus sprites, se decide utilizar los *software* de diseño *Adobe Photoshop* y *Corel Draw*. Ya que al momento de elegir dichos softwares ya se contaba con experiencia previa sobre su funcionamiento y no requiere ningún tipo de periodo de prueba para familiarizarse con su funcionamiento. Ambos *softwares* son de pago y para el desarrollo del presente trabajo terminal se utiliza una licencia personal por lo que si se desea comercializar el juego va a ser necesario adquirir otro tipo de licencia para la generación de *sprites*.

Cultura

Una vez explicado lo que es el videojuego, su metodología de desarrollo y las herramientas a usar para desarrollarlo, es preciso definir lo que es la cultura; para tal objetivo el presente trabajo se vale de la definición propuesta por la Organización de las Naciones Unidas para la Educación, la Ciencia y la Cultura (UNESCO, por sus siglas en inglés). La UNESCO define la cultura como “el conjunto de los rasgos distintivos, espirituales y materiales, intelectuales y afectivos que caracterizan a una sociedad o un grupo social. La cultura engloba, además de las artes y las letras, los modos de vida, los derechos fundamentales al ser humano, los sistemas de valores, las tradiciones y las creencias; de igual forma la cultura da al hombre la capacidad de reflexionar sobre sí mismo**RefCultura**”. Bajo su misma definición la UNESCO, se plantea que la importancia de la cultura radica en su capacidad de hacer a los seres humanos racionales, críticos y éticamente comprometidos; ya que, través de ella se disciernen los valores y se efectúan opciones. Siendo por medio de ella que el hombre se expresa, toma conciencia de sí mismo, se reconoce como un proyecto inacabado, pone en cuestión sus propias realizaciones, busca incansablemente nuevas significaciones, y crea obras que lo trascienden **RefCultura**.

Para efectos del presente trabajo terminal, este únicamente va a abordar la cultura de carácter histórica, es decir la cultura que hace referencia a la herencia social, es decir aquella que relaciona a la sociedad con su pasado **RefculturaClasificacionEl**.

3.1.3. Planteamiento de la solución

Con el fin de fomentar la cultura y la historia se desarrolla Yolotl, un videojuego de plataforma y aventuras en dos dimensiones para dispositivos móviles android 5.1 de gama media alta.

Las razones por las que se aborda la solución del problema con un videojuego se debe principalmente a diferentes factores tales como:

- **El estado de la industria mexicana de los videojuegos:** En el 2017 México ocupó el 12 puesto en cuanto a consumo de videjuegos percibiendo un ingreso de 1.4 mil millones de dólares en esta industria. A su vez México cuenta con 49.2 millones de jugadores [Ref_JuegosGanancia](#).
- **El auge de los juegos para teléfonos móviles:** En el 2017 la industria del videojuego tuvo ganancias de 108.9 mil millones de dolares de los cuales el 32 % de las ganancias fueron generadas por los teléfonos inteligentes y un 10 % por las tablets; con este porcentaje los teléfonos superaron a las consolas de mesa en ingresos [Ref_JuegosGanancia](#).
- **El consumo de teléfonos móviles en México:** En el 2017 México contaba con 52 millones de usuarios de teléfonos móviles, lo que lo ubicó en el 9 puesto a nivel mundial en el consumo de teléfonos inteligentes [Ref_TelefonosGanancia](#).
- **La interactividad de un videojuego:** Como se menciona en el artículo *Identification with the Player Character as Determinant of VideoGames Enjoyment*: en los videojuegos, la interactividad juega un papel importante para identificar y adoptar un determinado concepto, ya que dentro del videojuego el jugador no es un espectador, pues participa directamente en la historia e interactúa con el mundo del personaje; esto genera una relación íntima entre el jugador y el personaje puesto que es gracias al jugador que el personaje puede avanzar en la historia y a su vez es gracias al personaje que el jugador puede interactuar con la historia [PlayerIdentification](#).

3.2. Ajustes al trabajo terminal

En esta sección se definen todas las nuevas estrategias a seguir para agilizar y optimizar el desarrollo del juego.

3.2.1. Correción del enfoque de la solución

-en lugar de gamificación se usa serious games. ¿Que causó el cambio? ¿Porque?

3.2.2. Nueva división de trabajo

Para realizar la segunda parte del trabajo terminal se decidió cambiar la división del trabajo con el objetivo de agilizar el desarrollo. Al terminar los sprints anteriores (Poner los problemas). Por tal motivo se decide reorganizar la asignación de tareas, en lugar de que los miembros del equipo de desarrollo se encarguen del mismo nivel, se reparten los niveles restantes del desarrollo entre los integrantes del equipo. Quedando la asignación de los niveles como se ve en la figura 3.1 .

Esta división de trabajo permite que los niveles se desarrollen de manera paralela y no de manera secuencial como se había trabajado hasta este *sprint*; simulando de esta forma un flujo de trabajo

Asignación de los niveles del juego	
Karla Rocío Márquez Hernández	Yasmine Pilar Hernández Bautista
Nivel 01, la chica y el perro (Ciudad)	Nivel 02, Nadie cruza mis dominios (Plataforma)
Nivel 01, la chica y el perro (Selva)	Nivel 02, Nadie cruza mis dominios (Jefe)
Nivel 03, la guarida del jaguar (Plataforma)	Nivel 04, Alas de obsidiana (Plataforma)
Nivel 03, la guarida del jaguar (Jefe)	Nivel 04, Alas de obsidiana (Jefe)
Nivel 05, el viento del norte (Plataforma)	Nivel 06, Sin gravedad (Plataforma)
Nivel 05, el viento del norte (Jefe)	Nivel 06, Sin gravedad (Jefe)
Nivel 07, castigo (Plataforma)	Nivel 08, la última batalla del jaguar (Plataforma)
Nivel 07,castigo (Jefe)	Nivel 08, la última batalla del jaguar (Jefe)
Nivel 09, el último caballero del rey (Plataforma A)	Nivel 10, el rey del Mictlán (Jafe fase 01)
Nivel 09, el último caballero del rey (Plataforma B)	Nivel 10, el rey del Mictlán (Jafe fase 02)
Nivel 09, el último caballero del rey (Jefe A)	Nivel 10, el rey del Mictlán (Jafe fase 03)

Figura 3.1: Asignacion de tareas

similar a procesamiento multihilo, en el que cada integrante del equipo es un hilo y desarrolla sus tareas de manera paralela al otro.

3.2.3. Actualizando el motor de juego

Se decidió cambiar la versión 5.4 de *Unity* por la versión 2017.3.1f de *Unity3D*. Esta versión incluye herramientas que agilizan la creación de niveles como el uso de:

- **Tilemap:** Herramienta para el mapeado de niveles. Esta herramienta facilita la creación de mapas al crear una malla sobre la que se arrastraran diferentes *Sprites* que se hayan importado previamente al tilemap (ver figura 3.2). En la sección () se profundizará su funcionamiento.
- **Cinemachine:** Asset que permite controlar la cámara de la escena, con este asset se le puede indicar que objeto se desea que la cámara siga y se puede asignar un área que limitara el movimiento de la cámara (ver figura 3.4). *Cinemachine* se descarga directamente desde la tienda de assets de *Unity* y fue desarrollado por los ingenieros de *Unity*, lo que significa que no genera conflictos o no requiere de configuraciones extras al proyecto para importar. En la sección () se profundizará su funcionamiento.
- **Sprite Packer:** Si bien no es una herramienta para construcción de niveles o un asset, esta herramienta es una de las más útiles que se agregó a la nueva versión de *Unity* ya que, como su nombre lo indica, permite el empaquetado de *sprites* (ver figura). Empaquetar los *sprites* es una práctica que optimiza el renderizado de objetos, ya que el controlador de gráficos de *Unity* realiza una sola llamada por paquete cuando renderiza los objetos y con esa única llamada renderiza todos los objetos de la escena que se encuentren en ese paquete; si los



Figura 3.2: Vista de la escena cuando se tiene un *GameObject* de tipo *Tilemaps* para la construcción de niveles

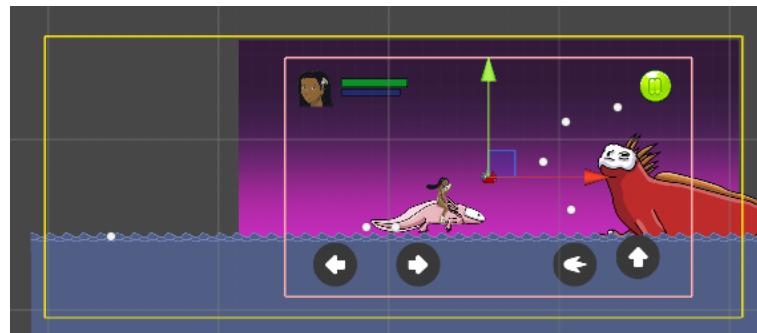


Figura 3.3: Vista de la escena cuando se tiene un *GameObject* de tipo *Tilemaps* para la construcción de niveles

sprites no se encontraran dentro de un paquete el controlador de gráficos de *Unity* haría una llamada por cada *sprite*.

Por el impacto que tendrían las nuevas herramientas de la versión de *Unity*, se propuso utilizarla en lugar de la versión 5.6.2f1. Antes de actualizar la versión de *Unity* se investigó si el proyecto sufriría algún impacto negativo como falta de compatibilidad de componentes por la diferencia de versiones. Al comprobar que existía una total compatibilidad entre ambas versiones en cuanto a trasladar un proyecto de la versión 5.6.1f a la versión 2017.3.1f. Se determinó que la nueva versión de *Unity* sería la que se emplearía para el resto del desarrollo del juego.

3.2.4. Máquina de estados

Dentro del desarrollo de videojuegos existen personajes no jugables, abreviados como NPC del inglés (no playable character) y para su funcionamiento se utilizan diferentes métodos o teorías. Dado que existía como riesgo acciones lineales o repetitivas para los jefes enemigos se decidió por emplear la teoría de autómatas. Para las acciones del jefe en los niveles tres, cinco, siete y nueve se utiliza un autómata finito. El estado final es "Morir", donde este representa la destrucción del objeto.

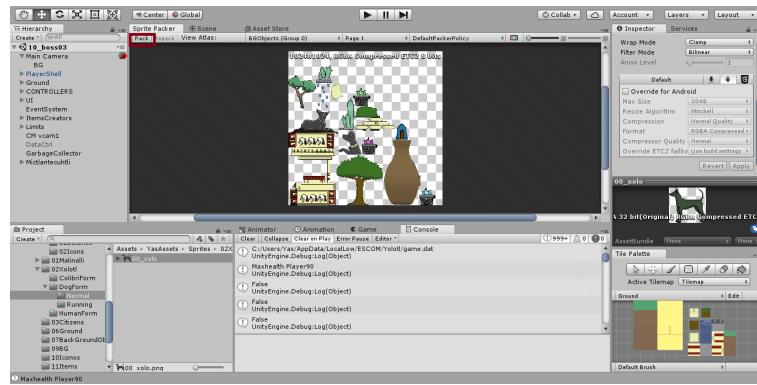


Figura 3.4: Vista de la pestaña del *Sprite Packer*.

Jefe nivel 3

Dada las acciones descritas anteriormente en el documento de diseño, se tiene coraza, impacto, lluvia de rocas, rugido. Además se agregan otras acciones no contempladas para la ayuda de transiciones. La nomenclatura queda como sigue:

- Morir ->Mo
- Coraza ->Co
- Impacto ->Im
- Correr x4 ->Cx4
- Correr x1 ->Cx1
- Rugido ->RA
- Lluvia de rocas ->LR
- Vida 0 ->V0
- Esperar 2 seg. ->E2s
- Esperar 1 seg. ->E1s
- Jugador cerca ->Jc

Los estados son:

$$Q = \{ Mo, Co, Im, Cx4, Cx1, RA, LR \}$$

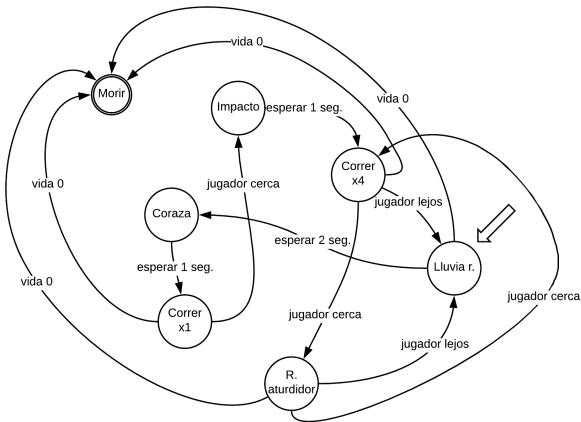
El alfabeto son: Jl, V0, E2s, E1s, Jc

El estado inicio es: $q_0 = LR$

El estado final es: $F = Mo$

Como vemos en la imagen 3.5 queda representada la máquina de estados.

Figura 3.5: Máquina de estados que realiza el jefe del nivel 3



Jefe nivel 5

Dada las acciones descritas anteriormente en el documento de diseño, se tiene ventisca y tornado. Además se agregan otras acciones no contempladas para la ayuda de transiciones. La nomenclatura queda como sigue:

- Morir ->Mo
- Estático x4 ->Ex4
- Estático x2 ->Ex2
- Tornado ->To
- Ventisca ->Ve
- Vida 0 ->V0
- Jugador lejos ->Jl
- Esperar 1 seg. ->E1s
- Jugador cerca ->Jc

Los estados son: Q = Mo, Ex4, Ex2, To, Ve

El alfabeto son: Jl, V0, E1s, Jc El estado inicio es: q0 = Ex4

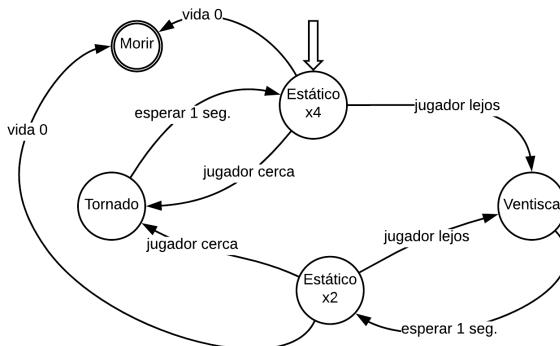
El estado final es: F = Mo

Como vemos en la imagen 3.6 queda representada la máquina de estados.

Jefe nivel 7

Dada las acciones descritas anteriormente en el documento de diseño, se tiene lluvia de flechas, manotazo, lava. Además se agregan otras acciones no contempladas para la ayuda de transiciones. La nomenclatura queda como sigue:

Figura 3.6: Máquina de estados que realiza el jefe del nivel 5



- Morir ->Mo
- Estático x4 ->Ex4
- Estático x3 ->Ex3
- Estático x2 ->Ex2
- Manotazo ->Ma
- Lava ->La
- Lluvia flechas ->Lf
- Vida 0 ->V0
- Rand 0 ->R0
- Rand 1 ->R1
- Rand 2 ->R2

Los estados son: Q = Mo, Ex4, Ex2, Ex3, Ma, La, Lf

El alfabeto son: V0, R0, R1, R2

El estado inicio es: q0 = Ex3

El estado final es: F = Mo

Como vemos en la imagen 3.7 queda representada la máquina de estados.

Jefe nivel 9

Dada las acciones descritas anteriormente en el documento de diseño, se tiene estocada, filo y sablazo. Además se agregan otras acciones no contempladas para la ayuda de transiciones. La nomenclatura queda como sigue:

- Morir ->Mo
- Estocada ->Es

Figura 3.7: Máquina de estados que realiza el jefe del nivel 7

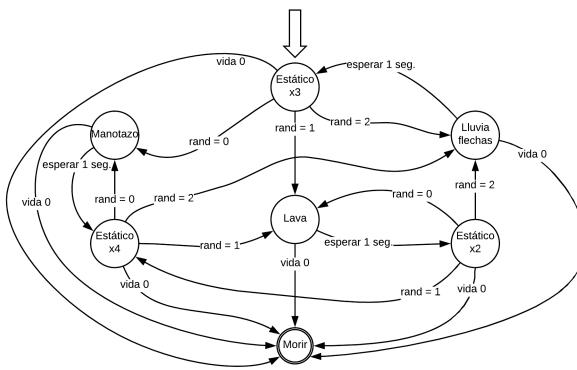
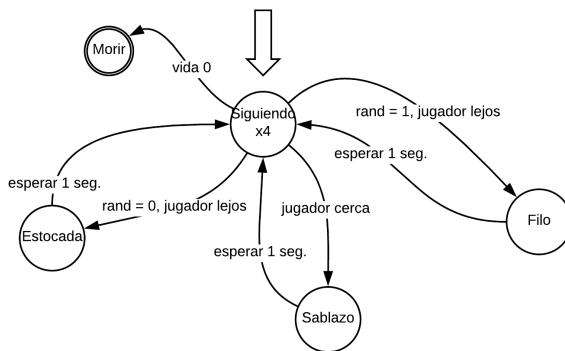


Figura 3.8: Máquina de estados que realiza el jefe del nivel 9



- Estático x4 ->Ex4
- Sablazo ->Sa
- Filo ->Fi
- Jugador lejos ->Jl
- Rand 1 ->R1
- Rand 0 ->R0
- Vida 0 ->V0
- Esperar 1 seg. ->E1s
- Jugador cerca ->Jc

Los estados son: Q = Mo, Ex4, Es, Sa, Fi

El alfabeto son: Jl, Jc, E1s, R1, R0

El estado inicio es: q0 = Ex4

El estado final es: F = Mo

Como vemos en la imagen 3.8 queda representada la máquina de estados.

3.3. Contribuciones

En esta sección se presentan las soluciones a las observaciones realizadas por los sinodales durante la presentación del trabajo terminal 1.

3.3.1. El juego

El juego definida por la RAE es una actividad recreativa o de competición sometida a reglas por el entretenimiento. Sin embargo más que ello es parte fundamental para el desarrollo y aprendizaje de cualquier individuo. Esta actividad contribuye a la maduración, potencia cognitiva, desarrollo emocional, vehículo emocional que contribuye para aprender nuevas habilidades y conceptos a través de su propia experiencia.

El juego refleja la percepción de sí mismos, de otras personas y del mundo que nos rodea. Por ello mismo cuenta con 5 grandes ventajas:

- El juego otorga placer y felicidad.
- En el juego no se tiene miedo al error.
- Fomenta la creatividad.
- Práctica de creación de estrategias y colaboración.
- El juego es el aprendizaje natural de las personas.

Teorías del aprendizaje

Es el estudio del aprendizaje que concierne al proceso por el que ocurre según el libro "Teorías del aprendizaje" **libro Teo Apr.** Pues se necesita comprender algunas suposiciones generales de las teorías que sustentan el aprendizaje humano y de la forma en la que se construyen sus principios.

Las teorías más reconocidas sobre el aprendizaje son:

- Gestalt: reestructuración perceptual.
- Piaget: Constructivismo genético.
- Vygotsky: Teoría sociocultural.
- Ausbel: Teoría del aprendizaje significativo.
- Bruner: Teoría cognitiva.

Aquella más cercana al juego para el trabajo a realizar es la teoría de Bruner. Pues el jugador que contempla es epistémico social, inserto en una cultura y estructurado por un lenguaje. La inteligencia está relacionada con 3 etapas de desarrollo para conocer: ejecución, impresión o imagen y significado simbólico. La evaluación está enfocada al estudio integral de los procesos cognoscitivos y los cambios que se originan.

Los videojuegos como medio de comunicación

Los videojuegos gracias a sus características de alcance masivo y presentación interactiva al usuario, son considerados parte de las TIC (tecnologías de la información y comunicación). Estos son más atractivos e influyentes dado que se enfoca a el ocio y entretenimiento de las personas. Es así como podemos ver incluso a los videojuegos usados como publicidad, puede ser de manera implícita donde se muestre marcas o productos dentro de un escenario o situación del juego o explícita donde el mismo juego presenta a la marca mostrando sus cualidades y ventajas (en la mayoría de los casos de forma exagerada). Además podemos ahora combinar la expansión que nos da el internet junto con la diversión de un videojuego, posibilitando a los jugadores la capacidad de promover los productos que han probado y enseñarlo a los demás jugadores.

Serious games

Aquí podemos aprovechar la creación de un serious game, pues son los juegos digitales con una finalidad explícita para el aprendizaje más allá del entretenimiento sin ser pensados en la diversión. Tienen su interés en el desarrollo de las competencias, mediante actividades interactivas basadas en el juego. Contribuyen al desarrollo de la coordinación ojo-mano, agudeza visual, reacción, atención múltiple, aptitud relacional, motivación, tolerancia a la frustración, toma de riesgos, resolución de problemas y toma de decisiones, así como la reflexión estratégica, la creatividad, cooperación y sentido de innovación marqués, 2010, p.276. Así mismo el jugador mejora el desempeño y se adentra a la experimentación, dada una situación simulada en la realidad virtual sin tener que enfrentar los riesgos de la realidad.

La gamificación y game-based learning son herramientas que persiguen el mismo objetivo de atraer y hacer practicar experiencias para memorizar y retener contenidos. Pueden usarse como ayuda para crear un serious game.

La gamificación es el uso de elementos de juego y técnicas de diseño para potenciar la motivación y compromiso de los jugadores. Mientras game-based learning se refiere al área cognitiva y apariencia donde debe crearse una experiencia de aprendizaje positiva.

En el siguiente cuadro **gabale** establecemos las diferencias más destacadas en ambas técnicas.

Cuadro 3.1: Diferencias entre gamificación y game-based learning

Gamificación	Game-based learning
Incluir los mecanismos de los juegos a situaciones de aprendizaje	Usar los juegos para crear una experiencia
Existen como motivadores puntos de experiencia, logros e incentivos	La experiencia va dirigida al pensamiento
Enriquece la ambientación y simulación del aprendizaje	Ambientación y simulación controladas

Motivos para jugar

El área de interés para el desarrollo del trabajo es la gamificación, para ello la parte importante a conocer son los diferentes motivantes que tiene una persona al jugar.

Para determinar el perfil motivacional se tomará la rueda de motivos "3.9 definida por Valderrama **avalde**, donde se define motivos de aproximación; aquellas personas sociales y buscan la convivencia y motivos de evitación; aquellas personas que prefieren la seguridad y estancia individual.

Es así como tenemos en contra partes diferentes motivos dependiendo del jugador objetivo, que son la búsqueda de:

Figura 3.9: Rueda de motivos de Beatris Valderrama



- Logros o hedonismo
- Exploración o seguridad
- Contribución o conservación
- Autonomía o afiliación
- Poder o cooperación

3.3.2. Modelo de negocios en un videojuego

Para sustentar un proyecto o producto económico se debe tener claro un modelo de negocios. En el mundo de los videojuegos no existe la excepción, pero también debe considerarse que existen formas muy diferentes de adquirir el ingreso.

Incluso el mismo juego puede estar involucrado en un ingreso directo del servicio.

Mercado global

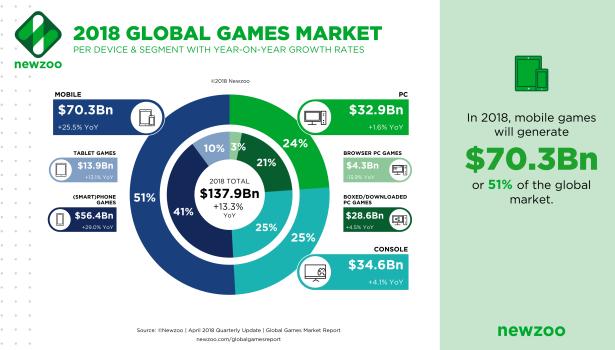
Se reporta segun Newzoo que 2.3 billones de jugadores en todo el mundo gastarán \$ 137.9 billones en juegos en 2018. Esto representa un aumento de + 13.3 % en comparación con el año anterior, o \$ 16.2 billones. Los ingresos por juegos digitales tomarán el 91 % del mercado global con \$ 125.3 mil millones, como podemos ver en la **fig:merglo**

Por primera vez, más de la mitad de todos los ingresos del juego provendrán del segmento móvil como vemos en la imagenasdasd. Los teléfonos inteligentes representarán el 80 % de esto, o \$ 56.4 mil millones, con el 20 % restante proveniente de tabletas.

Presentación al cliente

Un videojuego como cualquier software al momento de ser vendido al cliente puede encontrarse en dos presentaciones, una versión física o solo digital. En la siguiente tabla **fidi** se muestran las diferencias más destacables.

Figura 3.10: Mercado global al primer trimestre del año 2018



Cuadro 3.2: Tabla comparativa de un producto físico o digital

	Físico	Digital
Coste de desarrollo	sí	sí
Coste de producción	sí	no
Coste de envío	sí	no
Riesgo de sobra/infra producir inventario	sí	no
Facturación por unidad vendida	mucho mayor	mucho menor
Unidad vendida costea soporte	generalmente sí	imposible
Porcentaje del precio de venta que percibe la empresa	30 %-40 %	70 %
Tiempo de cobro	90 días o más	30 días
Capacidad de llegar al público con marketing	caro	difícil

Aún así cabe mencionar que este es un aspecto general que involucra a cualquier software.

Formas de ingreso

Dentro de los videojuegos existen modelos de negocio que han ido cambiando a lo largo de los años y muchas de las veces depende del tipo del juego. Pero podemos definir las siguientes conforme lo visto y consumido en los últimos 5 años a la fecha del proyecto a presentar.

Ingredientes de monetización

Los modelos de negocio anteriores pueden ser combinables con otros "ingredientes" de monetización para acrecentarlos ingresos.

- Dinero virtual: Es el medio de intercambio que utiliza un videojuego para poder formalizar las compras dentro de él. A menudo se suele diferenciar el virtual currency (dinero virtual que se consigue por las propias mecánicas del juego y con abundancia) y el hard currency (dinero virtual premium que se consigue con dinero real o con acciones muy concretas y con mucha escasez).
- Bienes virtuales: Son objetos intangibles que son comprados e intercambiados que sólo tienen sentido dentro del juego, muchas veces estos son comprados con dinero virtual.
- Publicidad y patrocinio: Anuncios o productos presentados en el juego para darse a conocer.

Cuadro 3.3: Tabla comparativa de ventajas y desventajas de modelos de negocios de videojuegos

Nombre	Descripción
Pay-to-play	Se debe pagar contenido y uso del videojuego
Free-to-play	Ofrece gratis contenido y uso del videojuego en su totalidad, se monetiza con publicidad y compras dentro del juego.
Freemium	Ofrece gratis el uso del videojuego pero no se accede a todo su contenido, establece jerarquías y niveles para el contenido.
Suscripción	Se debe pagar el contenido y uso del videojuego pero con limitaciones.

Cuadro 3.4: My caption

Rama	Profesión	Salario con <3años de exp.	Salario con 3-6años
	Programador	\$71,855 USD	\$79,877 USD
Programadores e ingenieros	Programador principal		\$94,877 USD
	Director técnico		
	Animador	\$50,000 USD	\$55,547USD
Artistas y animadores	Artista principal		\$71,029 USD
	Director de arte		
Diseñadores de juego	Diseñador de juego	\$53,000 USD	\$65,516 USD
	Director creativo		\$68,654 USD
	Productor asociado		\$59,079 USD
Productores	Lider de proyecto		\$73,500 USD
	Productor ejecutivo		
Profesional de audio	Director de sonido		
Testers	Tester		\$38,833 USD
	Lider de control de calidad		\$60,417 USD
	Marketing		\$73,500 USD
Negocios y administración	CEO		
	Gerente ejecutivo		

- Bonificaciones y servicios virtuales: Son aceleradores de juego o servicios que mejoran el desempeño o facilitan en el juego.
- DLC (downloadable content): Es un contenido de descarga digital exclusivo y adicional de un videojuego que se vende por separado y posterior al lanzamiento de este. Suele lanzarse para alargar la longevidad del videojuego y para aprovechar su éxito comercial. Su adquisición no tiene sentido sin tener antes el videojuego ya que es un producto complementario y dependiente a él.

3.3.3. Costo de hacer un videojuego

Salarios

Para realizar un videojuego se necesita de diferentes profesiones para llevarlo a cabo. En el siguiente cuadro se mostrará la profesión y salario a recibir en la industria del videojuego en una empresa ya establecida al año 2014 según la encuesta con una relación definida en experiencia.

3.3.4. Vicio en el videojuego

Un videojuego se está convirtiendo en una adicción cuando la ansiedad se sobrepone al placer del juego y como síntoma se tiene la urgencia de jugar sin pensar en las consecuencias. El vicio se establece con cierta facilidad, ya que el juego ofrece un entorno de inmersión al combinar los aspectos interactivos e identificarse con el personaje o situación.

En la siguiente tabla se muestra las opiniones más comunes a favor y en contra de los videojuegos sin incluir mitos o falsedades, todas estas afirmaciones son demostrables o no han sido posible desmentirlas con hechos o pruebas concretas.

Cuadro 3.5: Opiniones comunes a favor y en contra de los videojuegos

A favor	En contra
Entretienen	Provocan adicción
Ejercitan la coordinación oculo-manual	Promueven conductas violentas
Estimulan la capacidad de lógica y reflexión	Aíslan socialmente
Ayudan a concentrar la atención	Limitan la imaginación
Son un potencial muy adecuado para distintas aplicaciones sociales	Restan tiempo de otras actividades

Síntomas de vicio

A continuación se muestran síntomas identificables visualmente que ya representan una adicción al juego:

- El jugador parece estar absorto al juego, sin atender cuando lo llaman.
- Siente demasiada tensión, incluso aprieta las mandíbulas cuando juega.
- No aparta la vista de la pantalla.
- Empieza a perder interés por otras actividades que practicaba.
- Trastornos del sueño.
- Distanciamiento de familia y amigos.
- No respeta los horarios estipulados.

Carcaterísticas de un videojuego con potencial al vicio

Los videojuegos (en especial los free-to-play) contienen características de vicio como: la necesidad de concluir “tareas incompletas”, síntomas de abstinencia y la posibilidad de jugarlo en todo momento. En este apartado solo mencionaremos algunas de ellas y al menos las más visibles en muchos videojuegos.

- El efecto Zeigarnik: Se tiene incomodidad de las personas por tener “tareas incompletas”, en el caso de un videojuego se genera la necesidad por terminar el juego. El juego a su vez puede contemplar el nivel de porcentaje completado de un juego, provocando al jugador dicho síntoma haciéndolo jugar hasta su completado.
- Síntoma de abstinencia: En donde se priva o dificulta la posibilidad de realizar una actividad a una persona. En un juego tenemos como se establece turnos de juego, para recuperarlos se establece un límite de tiempo de espera. Usualmente estos tiempos de espera son de aproximadamente media hora o múltiplos de ella, pues psicológicamente este tiempo es lo que soporta una persona con una “tarea incompleta” en mente.
- La competencia: Que consiste en una disputa entre personas que aspiran a un mismo objetivo o a la superioridad en algo. Así, en un juego, ayudado en la mayoría de las veces por las redes sociales se puede compartir y comparar el avance entre los jugadores.

Causas del vicio ajenas al videojuego

Entre los jóvenes entre 13 y 18 años especialmente existen muchas más causas para caer en el vicio de un videojuego, pueden ser factores psicológicos, emocionales, del entorno en el que se desarrollan y más. Incluso muchos de los factores siguientes encajan en otros tipos de adicciones.

- Atención inexistente de los padres.
- No hay límites establecidos por la familia.
- Los valores no están asentados.
- Utilización de los videojuegos como "niñera".
- El joven no tiene sentido de pertenencia o no es aceptado en los grupos sociales que interacciona.
- Necesidad de escape de la realidad a un medio virtual.
- Establece mayor libertad expresión (o intenciones verdaderas) dentro del juego, ya sean positivas o negativas.
- Que la persona padezca alguna enfermedad que imposibilite realizar otras actividades .
- Trastornos psicológicos como depresión, impulsividad o ansiedad.
- Situación ante la solución de problemas y toma de decisiones.
- Falta de control emocional.

Prevención del vicio

Como toda adicción existen formas de prevenir llegar a ella. Dadas las situaciones, causas, factores dentro del vicio del videojuego y con lectura de técnicas pedagógicas, las propuestas de solución se dan como sigue:

- Establecer un horario de juego:
 - Se puede poner una alarma que avise al jugador que ha estado jugando demasiado tiempo.
 - Se puede programar a cierto tiempo jugado un bloqueo.
- Complementariamente a la programación de un horario de juego, deberán establecerse qué actividades se llevarán a cabo en los momentos en los que no se va a jugar:
 - Poder agregar al juego notas recordatorias de las actividades por hacer.
 - Alarmas personalizadas dentro del juego.
- Evitar los juegos online, al menos hasta que se tenga una organización del tiempo libre que impida dedicar mucho tiempo a dichos videojuegos.
 - Evitar ser un juego online.

- No instalar la consola ni el ordenador en la habitación.
 - El juego va a ser inaccesible a horas determinadas como la madrugada y noche.
- Los padres deben conocer los videojuegos.
 - Pedir forzosamente los correos de los tutores.
 - Mandar mensajes a los tutores con un registro en resumen de lo que se está jugando.
 - Habilitar una opción de bloqueo para los padres.
 - Bloqueo automático e informe con estadística dependiendo de frecuencia de uso y tiempo de uso.

3.3.5. Modelo de datos

La primera observación en atender fue el modelo de datos del juego, dicho modelo de datos se realizó utilizando un modelo entidad relación de base de datos (Ver Anexo 7.3) ya que al modelarse de esta forma hace escalable el juego si se deseará en algún futuro emplear una base de datos para mejorar el almacenamiento de datos y el manejo de más usuarios para ofrecer un modo online. El modelo de datos está basado en el modelo de clases y contiene únicamente a las clases actoras. Toda entidad actora se define como una especialización de una entidad base llamada GameObject, esta entidad está definida por como su identificador y por otras entidades como GameObjectPosition, Level, Tag, AnimationMachine, entre otros.

3.3.6. Estrategias para combatir la adicción entre los usuarios

La segunda observación sobre la que se trabajo fue como disminuir la adicción del jugador al videojuego Yolotl. Esta observación dio lugar a una investigación sobre la adicción a los videojuegos ya que antes de proponer alguna solución se debía conocer cómo se definía, las causas y las consecuencias de la adicción al videojuego. Al final de la investigación se pudieron formular tres posibles soluciones para evitar la adicción del jugador; sin embargo, dado que este tópico no estaba en la planeación original del proyecto y por las implicaciones que llevaban cada una de las soluciones se decidió únicamente describir las soluciones y sus implicaciones sin desarrollar ninguna de las tres. A continuación, se describen a manera de resumen las soluciones (nuevamente si se dese a profundizar en la investigación realizada y las soluciones se puede consultar el Anexo 7.4):

- **Notificación de confirmación para continuar la partida.** Esta solución propone que el juego solicite la confirmación del usuario para continuar una vez que éste ha detectado que el jugador ha estado jugando durante un tiempo prolongado como una hora.
- **Control paterno.** El juego le envía un formulario al tutor del jugador por medio de un correo electrónico. En este formulario el tutor podrá decidir cuento tiempo al día la aplicación podrá estar abierta.
- **Sistema de vidas.** El jugador tiene una cantidad de vidas limitadas. Cada vez que el jugador ingresa a un nivel o muere dentro de uno y reinicia la partida se gasta una vida. Para recuperar vidas el jugador deberá de esperar un determinado tiempo.

3.3.7. Modelo de Negocios

3.4. Trabajo realizado durante trabajo terminal 1

En esta sección se habla a manera de resumen el trabajo realizado durante el periodo correspondiente a trabajo terminal 1. La división de esta sección queda organizada en dos subsecciones: una para la etapa de preproducción y otra para los dos primeros *sprints* de la etapa de producción.

3.4.1. Presentado en TT1

A continuación se muestra lo presentado en TT1, que representa la investigación, análisis y prototipos del proyecto.

Contexto

Primero nos encontramos a determinar que es la cultura; aquella que define la identidad de un individuo por sus creencias religiosas, de pensamiento, sentimentales y sociales. Mientras que la historia son todos los sucesos pasados dentro de un espacio específico. Entonces la cultura histórica es aquellos aspectos arquitectónicos, de pensamiento, éticas y morales, grupos de pertenencia y convivencia. Dentro de la hultura histórica encontramos que el 41 % de los mexicanos no asisten a eventos culturales y presentan una gran desinformación de ella.

Luego nos encontramos que existen formas de presentar cualquier tema o actividad dentro de los juegos. A partir de eso investigamos sobre los videojuegos, juegos que se presentan a través de un medio visual o auditivo en el que existe interacción por diferentes dispositivos de entrada. Dentro de los videojuegos hay clasificaciones por contenido, que son las utilizadas para determinar el tipo de público al que va dirigido, además de que es una clasificación conveniente para limitar compra y venta. Sin embargo existen muchas más clasificaciones, en donde el tipo de contenido, dispositivo u objetivo es su pertenencia.

Viabilidad

Con esta información ya en mano, buscamos sobre la situación actual de comercio de los videojuegos. Pasando primero por los ingresos a nivel mundial y en que dispositivos en el año 2017; a nivel mundial se tiene un ingreso de \$108.9 mil millones de dólares, el 42 % está en los dispositivos móviles, tanto teléfonos inteligentes como tabletas, el resto queda en la computadora y consolas. Así México queda en el doceavo lugar de consumo de videojuegos a nivel mundial, con un ingreso percibido de \$1.4 mil millones de dólares en una población de 130 millones de mexicanos. De la población en México el 20 % juega videojuegos, de ese porcentaje 45 % juega en el celular y 40 % juega de una a dos veces por semana.

Análisis

Una vez contemplada la información, se decide que el proyecto se haga en un dispositivo móvil android, dado que el 86 % de los usuarios de dispositivos móviles tienen android y como versión mínima 5.0 lolipop con su uso de 32 % de las personas.

La metodología a usar será Hundle, que consiste en un parecido a la metología scrum solo que está adaptada a la creación y desarrollo de videojuegos, donde se establece un proceso general de preproducción, producción y postmortem. Destacando aquí que se hicieron algunos ajustes dentro

de la parte de preproducción agregando y modificando apartados dado que eran necesarios aclarar antes de empezar con el proyecto.

Luego se pasó a definir las herramientas de desarrollo, donde se escogió debido a su flexibilidad multiplataforma el motor de juego Unity y como herramientas de dibujo corel draw x5 y photoshop.

Como arquitectura a usar se eligió modelo vista controlador, donde se dividirá el código en controladores, actores y auxiliares, estos últimos ayudarán a los controladores y actores en situaciones específicas donde no se encuentre en ninguna de estas características.

Progresión y prototipos

Ya realizadas las investigaciones y análisis previo para el proyecto, se estableció la progresión que iba a tener el juego, junto con definir las interfaces que contendría y su interacción entre ellas. Se estableció la mecánica del juego donde se definía los botones de acción del personaje, el espacio de características del personaje y un apartado para determinar otros factores como items o vida del enemigo.

Ya en los prototipos se pasó a el maquetado de los niveles uno y dos, determinando personajes y eventos, así como prototipos de uso de la herramienta Unity. Al final dando como resultado dos prototipos contenido el nivel uno y dos y un prototipo de uso de la herramienta.

3.4.2. Etapa de Preproducción

Esta etapa corresponde a la planeación análisis y diseño del juego. Como lo indica la metodología *Huddle*, para esta etapa se trabaja en el desarrollo del documento de diseño del juego. Esta etapa queda del desarrollo queda dividida en cuatro *sprints*.

Primer Sprint Huddle de Preproducción

Antes de iniciar el diseño del juego se realiza un trabajo de investigación sobre la cultura azteca. Esta investigación abarca:

- **La sociedad mexica:** su historia tradiciones y clases sociales.
- **Mitología mexica:** Dioses, mito de los cinco soles, mito de la creación del hombre del maíz, el Mictlán.
- **Historia de la Malinche:** Historia del personaje antes y después de la llegada de los españoles.

Durante la etapa de investigación se selecciona la información histórica que sera relevante y útil para la narrativa del juego y el diseño de su jugabilidad. Para la investigación histórica de esta etapa se consultan libros, códices, páginas de Internet, artículos de investigación e incluso se visitan museos como el templo mayor.

Segundo Sprint Huddle de Preproducción

En este *sprint* se redactan las primeras secciones del documento de diseño del juego *Yolotl*. Se inicia con la idea concepto y con el tema del juego. De igual forma se selecciona un nombre para el juego a desarrollar: *Yolotl*. Para algunos juegos la mecánica es la primera es ser definida; no obstante, por la naturaleza del juego como herramienta de transmisión de cultura, *Yolotl* nace con

su historia. La historia de *Yolotl* pasa por diferentes etapas de diseño; siendo modificada gradualmente, pero manteniendo algunos elementos clave como la lucha contra la divinidad.

En la etapa del concepto también se define la plataforma para la que será el juego: dispositivos móviles con sistema operativo Android 5.2. Por su parte se decide utilizar un motor de juego como herramienta de desarrollo, pues esto permite centrarse en el diseño e implementación de aquellos elementos que diferencien a *Yolotl* del resto de juegos, tal como su mecánica, sus personajes, etc. Luego de investigar sobre los motores de juegos disponibles, se elige Unity 3D como ambiente de desarrollo.

Una vez teniendo la idea concepto se define la visión del juego y sus mecánicas. En cuestión de las mecánicas el enfoque por el que se opta es el de mantener el juego con mecánicas simples y familiares para aquellos jugadores que ya habían tenido alguna experiencia con algún juego de plataformas, sin descartar algunos detalles que le dieran identidad al juego en cuanto a su jugabilidad. Paralelamente a la preproducción, se inicia el desarrollo de un primer demo con el fin de familiarizarse con la herramienta de Unity3D, este demo incluye las mecánicas más simples del juego.

Con la historia, la visión y la mecánica definidas se procede a puntualizar los estados del juego, diseñar las interfaces gráficas de navegación y de interacción con el personaje. Para ver la versión final de las interfaces se puede consultar anexo 7.1.

Tercer Sprint Huddle de Preproducción

En el tercer Sprint se definen la cantidad de niveles y en qué consiste cada uno, de igual forma se establecen los objetivos de cada nivel, la recompensa a obtener una vez completado el mismo, los enemigos a vencer y las cinemáticas que fungen como transiciones entre niveles.

Al mismo tiempo que se diseñan los niveles, se detallan los personajes tanto a nivel narrativo como a nivel de jugabilidad, definiendo habilidades para los enemigos, los niveles en los que parecerían y sus acciones dentro de la historia. Para esta parte se trata de obtener la mayor fidelidad posible a los mitos y códices. En el anexo 7.2 se habla a mayor detalle sobre el diseño de los personajes.

Cuarto Sprint Huddle de Preproducción

En el cuarto sprint se termina de escribir el argumento del juego, de esta destapa se obtiene el guión literario del juego. En este *sprint* también se definen elementos de ambientación para el juego tales como la música de fondo, los efectos de sonido y los efectos especiales.

De igual forma, en este *sprint* se especifican las armas de los personajes, los ítems; quedando diseñados tanto a nivel de comportamiento como a nivel visual. Al igual que con los personajes se busca que las armas, tanto en comportamiento como en diseño, se mantengan lo más fiel posible a los mitos y leyendas de donde se basaron.

Con el cuarto *sprint* se finaliza la etapa de preproducción, obteniendo así un documento de diseño lo suficientemente detallado como para iniciar el diseño del juego a nivel de ingeniera.

3.4.3. Etapa de producción

En esta sección se habla del trabajo realizado durante los dos primeros *sprints* de esta etapa, ya que fueron desarrollados durante los meses correspondientes al trabajo terminal 1. Todos los *sprints* del la etapa de producción posteriores al segundo *sprint* son abordados en la sección 4.

Primer Sprint Huddle de Producción.

En este *sprint* se realiza un análisis del documento de diseño, en consecuencia de este análisis se se diseña el videojuego en materia de las clases que lo componen y el modelo bajo el que funcionaría el juego a nivel de programación.

Haciendo uso del paradigma orientado a objetos se propone emplear tres tipos de clases:

- **Actores:** Son las clases que modelan a los enemigos, los ítems, los coleccionables, los check-points y al jugador.
- **Controladores:** Son las clases encargadas de gestionar la partida y la navegación entre interfaces. Estas clases desencadenan eventos conforme a las acciones de las clases actoras. Estas clases también son las encargadas de verificar que se cumplan las reglas de los niveles.
- **Auxiliares:** Estas clases ayudan al funcionamiento de los actores y los controladores. Estas clases también se encargan de vincular datos con las clases controladoras como efectos de sonido, música, datos para la progresión entre niveles.

El modelo planteado permite reutilizar parte del demo generado durante la etapa de preproducción. Por lo que en este *sprint* se inicia la integración del código del primer demo con el comportamiento modelado por las clases definidas en el párrafo anterior.

En el primer *sprint* de Producción también se crean los *sprites* del primer nivel utilizando la herramienta de modelado en *3D Blender*. En la figura 3.11 se pueden observar algunos de los modelos creados. Al finalizar este *sprint* se determina la no viabilidad del modelado en 3D de los *sprites* por cuestiones de tiempos; en consecuencia, se descarta este método para generar los *sprites* y se inicia el desarrollo de los *sprites* a partir de otras técnicas de animación más tradicionales.

Segundo Sprint Huddle de Producción.

En este *sprint* se inicia el desarrollo de los *sprites* con *Adobe Photoshop* y *Corel Draw*. A la par se inicia la maquetación de la etapa de selva del nivel uno. En este sprint se logran terminar todos los *sprites* referentes al primer nivel del juego tales como:

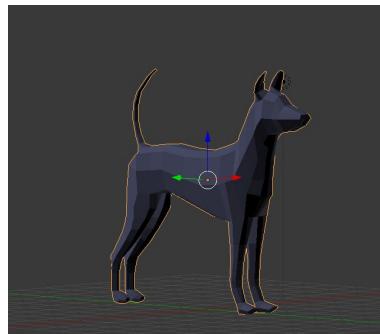
- Objetos de fondo: Arbustos, árboles, jarrones y cajas.
- Imagen de fondo: Fondo de la selva, la ciudad y el menú principal.
- Ciudadanos del mercado: Comerciantes, nobles y esclavos.
- *Xólotl* en su forma *xoloitzcuintle*: Bloques de animación para correr y normal.
- *Malinalli* sin la caracola: Bloques de animación correr, saltar y normal.

Una vez terminados los *sprites* referentes al nivel uno estos se integran al código permitiendo tener un segundo prototipo con la siguiente funcionalidad:

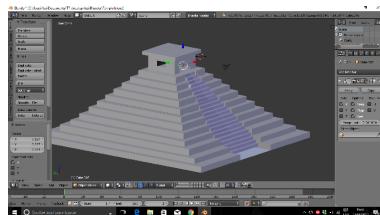
- Control de personaje por medio de la GUI.
- Transiciones entre interfaces.
- Personaje segurable que aparece en el primer nivel funcional.
- Funcionamiento básico del controlador de diálogos.



(a) Modelo de *Malinalli* generado en *Blender*.



(b) Modelo de *Xólotl* generado en *Blender*.



(c) Modelo de un templo generado en *Blender*.



(d) Modelo de una mujer comer- ciante generado en *Blender*.

Figura 3.11: Modelos de personajes y objetos crados en *Blender* (Autoria propia).

Capítulo 4

Trabajo realizado

4.1. Etapa de producción de *Huddle*

4.1.1. Detalles del proyecto

Aquí se mostrará del proyecto el título, estudio, género, plataforma, fecha de inicio, fecha de término, lo planeado, en desarrollo, sin planear y terminado.

Título

Yolotl

Estudio

ESCOM - IPN

Plataforma

Dispositivos móviles android

Fecha de inicio

1 Agosto 2017

Fecha de término

1 Junio 2018

Planeado

El 80 %

Desarrollo

El 10 %

Planear

El 20 %

Terminado

El 0 %

4.1.2. Sprint plan 1**SprintID**

01

Inicio

1 Agosto 2017

Fin

31 Agosto 2017

Meta

Nivel 1

Porcentaje

El 16 %

FeatureID

01

Nombre

N1

Estado

Planeado

SprintID

01

Comentarios

-

4.1.3. Sprint backlog 1

Sprint backlog

01

Tareas

6

Tendencia

0

Esfuerzo restante

56

Tendencia actual

-

Progreso ideal

10

Tareas restantes

10

Nombre de la tarea

n1

FeatureID

01

Miembro

Rocío

Rol

Desarrollo

Estado

Planeado

Esfuerzo

10

4.1.4. Buglist**BugID**

01

Descripción

Doble salto infinito

Descripción técnica

El gameobject no realiza la función detectar colisión de el "piso"

Autor

Velez

Estado

Revisión

FeatureID

01

4.1.5. Task Slips 1**FeatureID**

1

Nombre

Maqueta

Tarea

Realizar maqueta del nivel completo

Miembro

Rocío

Esfuerzo estimado

1

Terminado

sí

Restante

0

FeatureID

1

Nombre

Enemigos

Tarea

Realizar el comportamiento de los enemigos o cualquier otra acción necesaria dentro del nivel

Miembro

Rocio

Esfuerzo estimado

3

Terminado

sí

Restante

0

FeatureID

1

Nombre

Arte 1

Tarea

Crear vista de los personajes

Miembro

Rocío

Esfuerzo estimado

2

Terminado

sí

Restante

0

FeatureID

1

Nombre

Arte 2

Tarea

Crear el diseño de los obstáculos

Miembro

Rocío

Esfuerzo estimado

2

Terminado

sí

Restante

0

FeatureID

1

Nombre

Arte 3

Tarea

Crear el diseño de los items

Miembro

Rocio

Esfuerzo estimado

2

Terminado

sí

Restante

0

4.1.6. Sprint plan 3**SprintID**

03

Inicio

25 Septiembre 2017

Fin

24 Diciembre 2017

Meta

Nivel 2

Porcentaje

El 16 %

FeatureID

03

Nombre

N3

Estado

Planeado

SprintID

03

Comentarios

-

4.1.7. Sprint backlog 3**Sprint backlog**

03

Tareas

14

Tendencia

0

Esfuerzo restante

56

Tendencia actual

-

Progreso ideal

14

Tareas restantes

14

Nombre de la tarea

n3

FeatureID

03

Miembro

Rocío

Rol

Desarrollo

Estado

Planeado

Esfuerzo

14

4.1.8. Task Slips 3**FeatureID**

3

Nombre

Maqueta

Tarea

Realizar maqueta del nivel completo

Miembro

Rocío

Esfuerzo estimado

1

Terminado

si

Restante

0

FeatureID

3

Nombre

Enemigos

Tarea

Realizar el comportamiento de los enemigos o cualquier otra acción necesaria dentro del nivel

Miembro

Rocio

Esfuerzo estimado

3

Terminado

sí

Restante

0

FeatureID

3

Nombre

Arte 1

Tarea

Crear vista de los personajes

Miembro

Rocío

Esfuerzo estimado

2

Terminado

sí

Restante

0

FeatureID

3

Nombre

Arte 2

Tarea

Crear el diseño de los obstáculos

Miembro

Rocío

Esfuerzo estimado

2

Terminado

Sí

Restante

0

FeatureID

3

Nombre

Arte 3

Tarea

Crear el diseño de los items

Miembro

Rocio

Esfuerzo estimado

2

Terminado

Sí

Restante

0

FeatureID

3

Nombre

Boss 1

Tarea

Diseñar la máquina de estados del enemigo principal del nivel

Miembro

Rocio

Esfuerzo estimado

2

Terminado

sí

Restante

0

FeatureID

3

Nombre

Boss 2

Tarea

Crear la máquina de estados del enemigo principal del nivel

Miembro

Rocio

Esfuerzo estimado

2

Terminado

sí

Restante

0

4.1.9. Cuarto *sprint* de producción

Una vez definidas las nuevas estrategias de trabajo y que se atendieron las observaciones de los sinodales se procedió a realizar la maquetación de los niveles pares restantes y los *sprites* faltantes. Al término de este *sprint* se obtienen más de 100 *sprites* y las maquetas de los niveles pares.

Creación de las maquetas de los niveles pares restantes

Para la generación de las maquetas se sigue usando la plantilla creada durante la creación del primer demo del juego (ver figura 4.1). Para la creación de la maqueta también se crea un documento con todos los componentes de un nivel como son los enemigos, los ítems, los puntos de guardado, las plataformas y los obstáculos. Este documento se imprime y los objetos se recortan para ser pegados como estampas en las plantillas de diseño de las maquetas. En promedio la maqueta de cada nivel no excede de las 15 plantillas, sin embargo hay algunas que exceden este número como la maqueta del cuarto nivel, la cual por su naturaleza de laberinto terminó por ser más extensa que el resto. Por su parte los niveles correspondientes a los jefes de los niveles no sobrepasan de las tres plantillas, siendo la maqueta del jefe del sexto nivel la más pequeña de todas. En el anexo 7.5 se pueden consultar las maquetas de los niveles pares.

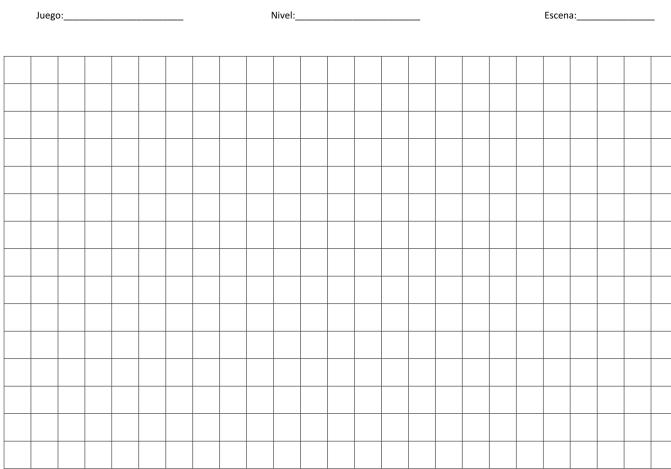


Figura 4.1: Plantilla para la creación de niveles.

Creación de los *sprites* faltantes

Lo siguiente a realizarse durante el cuarto *sprint* fueron los *sprites*, durante las modificaciones que se definieron en Trabajo Terminal 1 fue la utilización de un *software* de animación en dos dimensiones para generar los *sprites* restantes; sin embargo, el cambio de *software* para generar los *sprites* fue descartado, esto debido a que se adquirió una nueva tableta digitalizadora que agilizó

la creación de *sprites*. Para Trabajo Terminal 2 se dibujaron y digitalizaron más de 100 *sprites*. Para mejorar la experiencia visual del jugador se animaron *sprites* que en los primeros demos eran estáticos como es el caso de los fantasmas del segundo nivel de la sección de plataformas (ver figura 4.2).

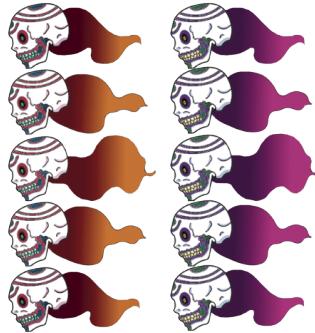


Figura 4.2: Bloques de animación para el enemigo de tipo fantasma.

Otros cambios en cuanto el aspecto visual del juego fue la integración de nuevos *sprites* para el personaje jugable, los nuevos sprites incluyen la caracola que *Malinalli* (ver figura 4.3) emplea para atacar y que se obtiene al final del primer nivel de la sección de selva, estos *sprites* para *Malinalli* son utilizados únicamente en los niveles posteriores al primer nivel para darle sentido a la narrativa; para el segundo nivel se hizo algo parecido, los *sprites* del personaje jugable fueron sustituidos por *Malinalli* montando un ajolote (ver figura 4.4), este cambio se hizo para que lo que el jugador vea dentro del nivel sea coherente con la narrativa propuesta y se mejore la inmersión del juego.

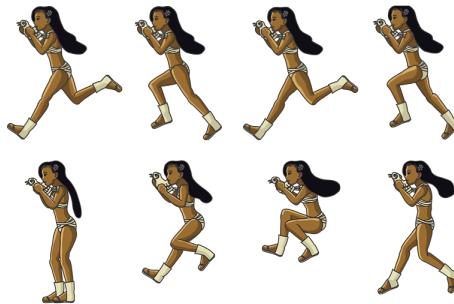


Figura 4.3: Bloques de animación para *Malinalli* posterior a que ella obtiene la caracola.

En lo que se refiere a los Jefes de cada nivel, no solo se crearon sus respectivos *sprites*, también fue necesario la creación de los *sprites* referentes a sus ataques, para el caso particular de *Mictlantecuhtli* se dibujaron 30 *sprites* tanto para la animación del personaje como para la animación de sus respectivos ataques (ver figura 4.5). Para el diseño de la interfaz gráfica de usuario (*GUI* por sus siglas en inglés) se emplearon *sprites* de las páginas *Kenney.nl* y *Game Art 2D*. Es importante aclarar que la creación de *sprites* pudo haber sido sustituida utilizando paquetes de *sprites* que existen

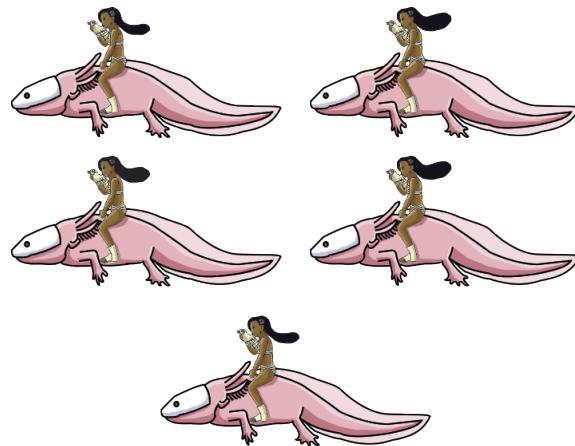


Figura 4.4: Bloques de animación para *Malinalli* montando al ajolote del segundo nivel del juego.

en la red y que son de licencia libre; sin embargo, con la creación de *sprites* propios para el juego se consigue crearle una identidad visual propia al juego, esto permite que el jugador se identifique con mayor facilidad con el personaje y tenga una mejor asociación con el mundo y la historia que se le presenta dentro del juego [cita]. Si se desea ver a profundidad los sprites que se crearon se puede consultar el anexo 7.2.



Figura 4.5: Bloques de animación para *Mictlan tecuhtli*, jefe final del juego.

Cierre del sprint

Al terminar este *sprint* se obtienen todas las maquetas de los niveles pares y los *sprites* a utilizar de los mismos. Este *sprint* se considera completado ya que se generaron todos los elementos que se tenían planeado. Como ultimo paso para este *sprint*, se realiza el empaquetado de todos los *sprites* creados durante el *sprint*.

4.1.10. Sprint plan 5

SprintID

05

Inicio

12 Marzo 2018

Fin

18 Marzo 2018

Meta

Nivel 5

Porcentaje

El 16 %

4.1.11. Feature log 5**FeatureID**

05

Nombre

N5

Estado

Planeado

SprintID

05

Comentarios

-

4.1.12. Sprint backlog 5**Sprint backlog**

05

Tareas

14

Tendencia

0

Esfuerzo restante

56

Tendencia actual

-

Progreso ideal

14

Tareas restantes

14

Nombre de la tarea

n5

FeatureID

05

Miembro

Rocío

Rol

Desarrollo

Estado

Planeado

Esfuerzo

14

4.1.13. Task Slips 5**FeatureID**

5

Nombre

Maqueta

Tarea

Realizar maqueta del nivel completo

Miembro

Rocío

Esfuerzo estimado

1

Terminado

sí

Restante

0

FeatureID

5

Nombre

Enemigos

Tarea

Realizar el comportamiento de los enemigos o cualquier otra acción necesaria dentro del nivel

Miembro

Rocio

Esfuerzo estimado

5

Terminado

sí

Restante

0

FeatureID

5

Nombre

Arte 1

Tarea

Crear vista de los personajes

Miembro

Rocío

Esfuerzo estimado

2

Terminado

sí

Restante

0

FeatureID

5

Nombre

Arte 2

Tarea

Crear el diseño de los obstáculos

Miembro

Rocío

Esfuerzo estimado

2

Terminado

sí

Restante

0

FeatureID

5

Nombre

Arte 3

Tarea

Crear el diseño de los items

Miembro

Rocio

Esfuerzo estimado

2

Terminado

Sí

Restante

0

FeatureID

5

Nombre

Boss 1

Tarea

Diseñar la máquina de estados del enemigo principal del nivel

Miembro

Rocio

Esfuerzo estimado

2

Terminado

Sí

Restante

0

FeatureID

5

Nombre

Boss 2

Tarea

Crear la máquina de estados del enemigo principal del nivel

Miembro

Rocio

Esfuerzo estimado

2

Terminado

sí

Restante

0

4.1.14. Sexto *sprint* de producción

En este *sprint* se tiene como objetivo la implementación de todos los actores del juego tanto a nivel de código como en la creación de los *GameObjects* para generar su posterior *Assets*. En esta sección no se hablan de aquellos actores que se hayan implementado desde Trabajo terminal 1 a no ser que su funcionalidad se haya rediseñado.

Implementando los enemigos normales del juego

Las primeras clases actoras en ser programadas fueron las correspondientes a los enemigos normales, estas clases se programaron a la par que la clase *Player*. Si bien la clase *Player* ya estaba programada desde los primeros prototipos, esta clase no contaba con toda su funcionalidad implementada y la funcionalidad faltante tuvo que ser implementada a la par que otras clases para verificar el correcto funcionamiento en la interacción de clases como la de los enemigos y los *ítems*.

Al igual que con la clase *Player*, existían enemigos desde los primeros prototipos; no obstante, su funcionalidad tuvo que ser reimplementada a fin de ofrecer un desempeño que optimizará recursos y agregará nuevas funcionalidades. A continuación, se listan los cambios que presentan los enemigos de séptimo *sprint* en relación de los enemigos del primer prototipo:

- **Áreas de acción:** En el primer prototipo los enemigos ejecutaban sus patrones de movimiento y ataque sin importar que éstos se encontraran visibles para el jugador o no. Los enemigos del sexto *sprint* cuentan con áreas de acción definida, lo que hace que sus patrones de movimientos y ataques solo se ejecuten si el jugador entra a estas áreas activas. Esto permite que el dispositivo no gaste recursos en objetos que no se encuentran visibles para el jugador (ver figura).

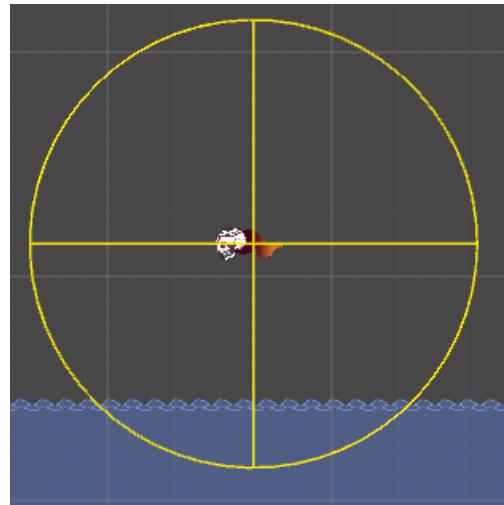


Figura 4.6: Ejemplo del area de acción del enemigo *RedGhost*

- **Cantidad de vida:** En el primer prototipo todos los enemigos eran derrotados por un único disparo, esto limitaba el factor de reto del juego al no ofrecer enemigos más resistentes al ataque del jugador. En el fin de ofrecer una nueva capa de complejidad a los enemigos se agrega a la clase *Enemy* el atributo *maxHealth* y *healthAmount*, estos atributos son los encargados de almacenar la máxima cantidad de vida que un enemigo puede tener y la vida actual de dicho enemigo (ver figura 4.7). La cantidad de vida sólo se actualiza cuando el enemigo es atacado por el jugador o cuando se reinicia el nivel. Para la actualización de la vida del enemigo se utiliza el comando *Clamp* de la clase *Math*, este comando permite especificar rangos con valores máximos y mínimos del resultado de operaciones, esto con la finalidad de que la vida actual del enemigo nunca sea cero y nunca sobrepase a su máximo de vida (ver figura 4.8).
- **Cantidad de daño:** Al igual que con la cantidad de vida, los enemigos del primer prototipo inflingían la misma cantidad de daño sin importar su tipo; por lo que para tener enemigos más y menos fuertes se agrega el atributo *damageAmount*. Un enemigo puede infringir daño al jugador cada vez que toca al jugador o cuando dispara un ataque. Cada vez que un enemigo o un disparo enemigo choca con el jugador, el objeto enemigo manda a llamar el método *SetHealth* del *Player* y le envía como parámetro el valor de su atributo *damageAmount*, seguido de un segundo método de la clase *Player* llamado *EnemyNockBack*, este método es el encargado de la animación que indica que el jugador ha recibido daño (ver figura 4.9).
- **Girar horizontalmente:** En el primer prototipo el enemigo era incapaz de girar sus *Sprite* y su ataque una vez que el jugador lo sobrepasaba como se ve en la figura 4.10. Utilizando la posición del enemigo y la posición del jugador dentro del área activa, el enemigo puede voltear su sprite y su ataque con base al valor de la distancia entre éste y el jugador:

```

public class Enemy : MonoBehaviour {
    [Tooltip("Float value, than represents the max health value")]
    public float maxHealth;
    private float healthAmount;
    [Tooltip("Float value, than represents the max damage value")]
    public float damageAmount;
    private bool isAlive;

    // Use this for initialization
    void Start () {
        healthAmount = maxHealth;
        //Debug.Log ("MaxHealth enemy" + maxHealth);
        //Debug.Log ("MaxDamage Enemy" + damageAmount);
        isAlive = true;
    }
}

```

Figura 4.7: Nuevos atributos para la clase *Enemy*.

```

public void SetHealth(float value){
    healthAmount = Mathf.Clamp (healthAmount-value, 0f, maxHealth);
    if (healthAmount == 0f){
        isAlive = false;
        SFXCtrl.instance.ShowEnemyExplosion (transform.position);
    }
}

```

Figura 4.8: Actualización de la cantidad de vida de la clase *Enemy*.

- Si la posición del enemigo es mayor que las del jugador, el enemigo mantiene su orientación inicial.
- Si la posición del jugador es mayor que la del enemigo, el enemigo se volteá.

Voltear un Sprite no representa mayor problema en código; sin embargo, el voltear un sprite cuyo colisionador no es simétrico como el de la figura 4.11. Esto puede representar un problema cuando se tiene que detectar colisiones, tal y como ocurre con los enemigos de tipo RedGost y PurpleGost; para evitar alterar la detección de colisiones se crea una nueva clase auxiliar llamada FixerCollider, cuyo objetivo es ajustar la posición del colisionador una vez que el personaje se gira como se puede observar en la figura 4.11.

- **Trigger Collider:** En el primer prototipo el colisionador del enemigo tenía una configuración del tipo sólido lo que ocasionaba que cuando el enemigo chocara con otro enemigo o con algún ataque enemigo este se estancara o fuera empujado por el objeto contra el que chocaba (ver figura). Para corregir este comportamiento se configuro el colisionador como uno de tipo trigger (ver figura 4.12).
- **Rigidbody2D:** Para evitar el comportamiento mencionado en el *Trigger Collider* también fue necesario modificar la configuración del componente *Rigidbody2D*, este componente pasa de estar en modo *Dynamic* a modo *Kinematic* lo que permite evitar que el objeto de juego reaccione conforme a las leyes físicas comunes.

Para implementar cada uno de los patrones de movimiento de los enemigos es necesario utilizar posiciones auxiliares que indiquen el límite del movimiento del personaje, salvo en la clase Vulture ya que este explota al hacer contacto con el jugador. En la figura 4.14 se puede observar el patron de movimiento del enemigo de tipo jaguar expresado en código. Este comportamiento consiste en un movimiento recto horizontal de un punto A a un punto B y de regreso, haciendo una pausa en el

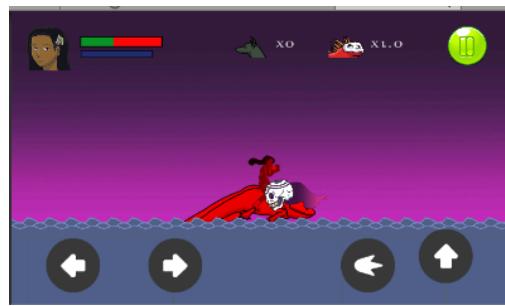


Figura 4.9: Ejecución de los métodos *SetHealth* y *EnemyNockBack* de la clase *Player*, los cuales actualizan la cantidad de vida del jugador y muestran la animación de que el jugador ha recibido daño.

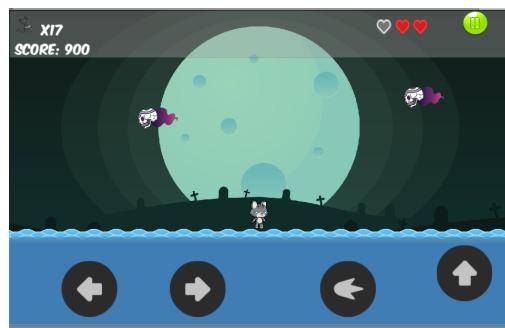


Figura 4.10: En los primeros prototipos el enemigo es incapaz de girar su sprite y su ataque una vez que el jugador se coloca tras de éste.

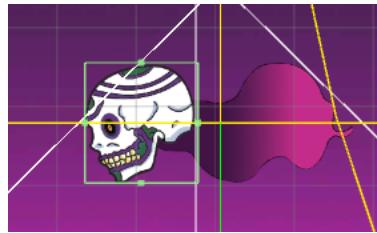
movimiento cada vez que el jaguar ha alcanzado cualquiera de los puntos A o B (ver figura).

Para resaltar la muerte de un enemigo se agrega un efecto especial de explosión acompañado de un efecto de sonido para la explosión del personaje. Para esta funcionalidad se implementa la clase SFXCtrl y AudioCtrl para manejar los efectos de especiales y el sonido respectivamente, siendo estos los primeros controladores en ser implementados.

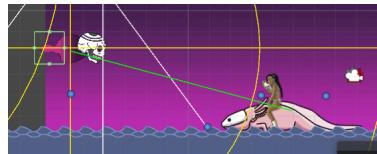
Implementando los enemigos jefes del juego

Para la implementación de los jefes se reutilizan las configuraciones de los enemigos normales referente a componentes como *Rigidbody2D* y el *Colisionador*, el uso de la clase *Enemy* para el manejo de vida y el uso de los efectos de sonido y de explosiones para la muerte del jefe.

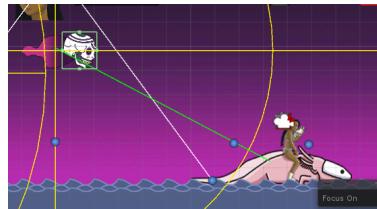
La lógica teórica tras los jefes del juego esta inspirada por el jefe *Roxas* (ver figura 4.16) del juego *Kingdom Hearts 2 Final Mix*. Dentro de *Kingdom Hearts 2 Final Mix*, *Roxas* es uno de los jefes que requiere mayor habilidad de juego para ser derrotado, ya que a diferencia del resto de los jefes de *Kingdom Hearts 2 Final Mix*, el patrón de ataque de *Roxas* es totalmente aleatorio. Es decir, el jugador puede saber en qué consiste cada uno de los ataques de este jefe, pero desconoce el orden en el que estos serán ejecutados, salvo por algunos ataques que están condicionados a una secuencia de ataque anterior. Con los jefes del juego *Yolotl* sucede algo parecido, el jugador puede llegar a conocer los tipos de ataque que posee un jefe determinado pero la secuencia de ejecución de los ataques está programada para que sea aleatoria, lo que puede generar experiencias de juego



(a) Ejemplo de un colisionador no simétrico respecto al *sprite*.



(b) Al girar el sprite de manera horizontal la posición del colisionador no se modifica



(c) Posición del colisionador modificada al emplear la clase *FixerCollider*.

Figura 4.11: Comportamiento del colisionador antes y después de la implementación d ela clase *FixerCollider*.

muy sencillas o bastante retadoras para el jugador. El anterior comportamiento se logra simulando una máquina de estados con un arreglo de tipo booleano llamado *whatCanDo*, en el cual solo un índice puede tener el valor verdadero cada vez que se actualiza el estado y dependiendo del valor del índice del valor verdadero será el ataque que ejecutará el enemigo. Después de cada ataque el enemigo espera un tiempo determinado antes de asignar el siguiente y ejecutarlo. Para ayudar al lector a comprender el funcionamiento de los jefes se explica nuevamente usando como ejemplo al jefe *Itzpapálotl* del nivel cuatro. El jefe *Itzpapálotl* cuenta con cuatro acciones:

- ***WaitForAction***: Espera un tiempo determinado y asigna un nuevo índice valor verdadero del arreglo de valores booleanos. Se activa si *whatCanDo[0]* es verdadero.
- ***shotFire***: Dispara cuatro esferas de fuego que siguen al jugador y en caso de no chocar con este después de un tiempo se destruyen. Se activa si *whatCanDo[1]* es verdadero.
- ***useShell***: Invoca un círculo de fuego que protege a *Itzpapálotl* de cualquier daño, el escudo de fuego también puede infringir daño al jugador si hace contacto con éste. Se activa si *whatCanDo[2]* es verdadero.
- ***CreateButterflies***: Invoca mariposas en tres puntos del campo, las mariposas también infringen daño al jugador y desaparecen después de un tiempo. Se activa si *whatCanDo[3]* es verdadero.

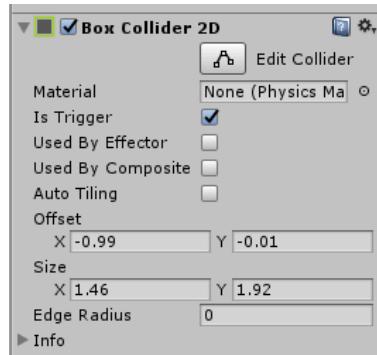


Figura 4.12: Configuración actual del colisionador de los enemigos.

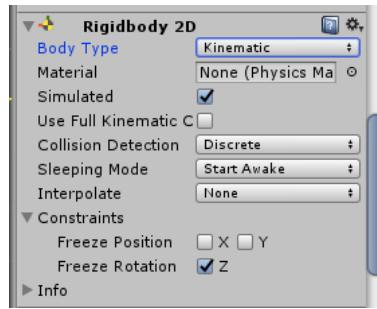


Figura 4.13: Configuración actual del componente *Rigidbody2D* de los enemigos.

Al inicializarse el jefe *Itzpapálotl* *whatCanDo[0]* es igual a cero. Por lo que *Itzpapálotl* ejecuta *waitForAction*, al terminar la ejecución de *waitForAction*, *whatCanDo[0]* es igual a falso y un nuevo índice tiene ahora el valor verdadero. Supóngase ahora *whatCanDo[2]* es verdadero. *Itzpapálotl* ejecuta *useShell*, al terminar su ejecución asigna *whatCanDo[2]* como falso y asigna a *whatCanDo[0]* como verdadero. Nuevamente *Itzpapálotl* espera unos segundos y actualiza *whatCanDo*. Por la naturaleza aleatoria de la actualización, *whatCanDo[2]* puede ser nuevamente verdadero o lo puede ser cualquier otro índice exceptuando al 0 o a un número mayor que el índice máximo del arreglo. En la figura se muestra la verificación de los valores de *whatCanDo* antes de la ejecución de cualquiera de los ataques que tienen asignados. En la figura 4.17 se muestra un ejemplo en código de la máquina de estados del jefe *Itzpapálotl*.

Por la forma en la que fue diseñado el comportamiento de la máquina de estados, el nivel de dificultad que presente el jefe esta dado en función de dos variables: *damageAmount* y *timeBetweenAttacks*, correspondientes a la cantidad de daño que el jefe puede infringir en el jugador y al tiempo que se espera para actualizar los valores de *waitForAction*. A mayor cantidad de daño y menor tiempo de espera entre ataques, mayor será la dificultad para derrotar al enemigo.

Implementando los ataques enemigos del juego

Dentro del juego existen seis tipos de ataques enemigos:

- **Disparos con una trayectoria definida:** Este tipo de disparo sigue una trayectoria recta horizontal como se ve en la figura 4.18. Para evitar la saturación de objetos dentro del juego, todos los disparos de este tipo se destruyen después de un tiempo. Para implementar este tipo de ataque se crea un *GameObject* y se le agregan los siguientes componentes:

```

IEnumerator TurnLeft(float originalSpeed){
    anim.SetInt("state", 0);
    //Debug.Log (anim.GetInteger ("state"));
    yield return new WaitForSeconds (minDelay);
    anim.SetInt ("state", 1);
    sr.flipX = false;
    speedPatrol = -originalSpeed;
    canTurn = true;
}

IEnumerator TurnRight(float originalSpeed){
    anim.SetInt ("state", 0);
    //Debug.Log (anim.GetInteger ("state"));
    yield return new WaitForSeconds (minDelay);
    anim.SetInt ("state", 1);
    sr.flipX = true;
    speedPatrol = -originalSpeed;
    canTurn = true;
}

void SetStartDirection(){
    if (speedPatrol > 0) {
        sr.flipX = true;
    }else if (speedPatrol < 0) {
        sr.flipX = false;
    }
}

public void MovePatrol(){
    Vector2 temp = rb.velocity;
    temp.x = speedPatrol;
    rb.velocity = temp;
}

```

Figura 4.14: Patrón de movimiento del enemigo tipo Jaguar expresado en código.



Figura 4.15: Patrón de movimiento del enemigo tipo Jaguar expresado en su comportamiento visual.

- **Collisionador:** El colisionador permite detectar si este ataque hace contacto con el *Player* o con el suelo del nivel, en el primer caso se infringe daño al *Player* y se destruye el *GameObject*, en el segundo el *GameObject* solo se destruye.
- **Rigidbody2D:** El *rigidbody2D* se configura con la opción *kinematic* para evitar que el movimiento del disparo sea afectado por la gravedad. Este componente permite en el código agregar una velocidad al objeto.
- **DestroyWithDelay:** Componente creado por medio de la clase del mismo nombre, esta clase destruye al *GameObject* que la contiene después de una cantidad determinada de tiempo.
- **EnemyBullet:** Esta clase controla en la velocidad y dirección del movimiento del disparo, también tiene como atributo el daño que causa la bala y gestiona las colisiones del objeto.

Este tipo de disparo es empleado por los enemigos de tipo *RedGost*, *Tepeyóllotl* y por *Mictlan-tecuhtli*.

- **Disparos que siguen al jugador:** Este tipo de disparo sigue un comportamiento y configuración parecida al anterior con la diferencia que en este tipo el disparo seguirá al jugador hasta impactarse contra éste o destruirse después de un tiempo si no colisiona contra el jugador. Este comportamiento requiere que el disparo tenga una referencia a la posición del jugador para moverse hacia él, en la figura 4.19 se puede ver la implementación de dicho comportamiento.



Figura 4.16: Roxas es el jefe más retador de *Kingdom Hearts 2 Final Mix*. Dentro de *Kingdom Hearts 2 Final Mix*. [Imagen] (2014) Recuperado de: https://images.khinsider.com/2014%20Uploads/05/Screenshots%205-30/KHII_battle_03_EN%20copy_1401446206.jpg

```
// Update is called once per frame
void Update () {
    /*
     * Check first if is visible if it's make the change between actions
     * otherwise fire
    */
    if (enemy.GetIsAlive ()) {
        if (whatCanDo [0]) {
            StartCoroutine (WaitForAction ());
            //Debug.Log ("Wait");
        } else if (whatCanDo [1]) {
            FireBullet (1f);
            //Debug.Log ("fire");
        } else if (whatCanDo [2]) {
            UseFireShell ();
            //Debug.Log ("shell");
        } else if (whatCanDo [3]) {
            CreateButterflies ();
            //Debug.Log ("shell");
        }
        Move ();
    } else {
        HanddleDeath ();
    }
}
```

Figura 4.17: Ejemplo de la máquina de estados del enemigo jefe.

to en código. Este ataque es utilizado por los enemigos de tipo *Mictlantecuhtli*, *Tepeyóllotl*, *Itzpapálotl*, *Xochitonal* y *Tlazolteolt*. Para todos estos enemigos el disparo tiene el mismo efecto que es el de infringir daño en el jugador; sin embargo, en el tipo de *Tlazolteolt* este tipo de disparo también puede disminuir la cantidad de *Tonalli* del Player.

- **Escudo de defensa que desaparece después de un tiempo:** Este ataque es efectuado por *Itzpapálotl*. Al invocarse este escudo el enemigo no se ve afectado por los ataques del jugador. Este escudo no puede ser destruido y desaparece después de un tiempo que se invocó; este comportamiento se logra utilizando el método *Invoke*, este método permite ejecutar métodos después de una cantidad de segundos; por lo que la desactivación se consigue mandando a llamar al metodo responsable de esto con *Invoke* y asignando una cantidad determinada de segundos de espera (Ver figura). Infringe daño al jugador al hacer contacto con él.
- **Escudo de defensa que debe de ser destruido para desaparecer:** Ataque utilizado por *Tlazolteolt*. Este escudo puede ser destruido por disparos del jugador y no desaparece al cabo de un tiempo. Al igual que el anterior protege al enemigo de los ataques del jugador e infringe

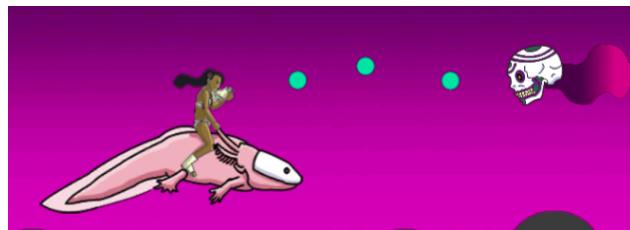


Figura 4.18: Ejemplo de un disparo de una trayectoria definida.

```
public void Move (){
    Vector3 target = player.transform.position;
    float fixedSpeed = speed * Time.deltaTime;
    transform.position = Vector3.MoveTowards (transform.position, target, fixedSpeed);
}
```

Figura 4.19: Implementación en código del comportamiento del disparo que sigue al jugador.

daño si el jugador hace contacto con éste. Este tipo de escudo no cuenta con un método que lo desactive, en su lugar tiene una cantidad de vida determinada y que al llegar a cero, por efecto de los ataques del jugador, desaparece y reaparece hasta que el enemigo jefe lo vuelve a convocar.

- **Objetos que aparecen en posiciones cuya aparición tiene un tiempo de duración:** Este ataque es empleado por *Itzpapálotl* y *Mictlantecuhtli*. Cuando se activa provoca que se creen instancias del *GameObject* que contiene la clase *Butterfly*. Esta clase genera un movimiento vertical ascendente e infringe daño al jugador al hacer contacto con esta. La creación de estos *GameObjects* se mantiene activa por un periodo de tiempo y después desactiva. Este funcionamiento se logra de una manera similar al comportamiento del escudo de defensa que desaparece después de un tiempo utilizando el método *Invoke*. En la figura se puede observar la implementación en código de este tipo de ataque.
- **Objetos que aparecen en posiciones de manera periódica:** Este ataque genera una lluvia de huesos o de piedras que le infringen daño al jugador una vez hacen contacto con este de lo contrario se destruyen al hacer contacto con el suelo. Para la creación de los objetos se utiliza el metodo *Invoke* el cual controla la cantidad de *GameObject* que se crean. Este ataque es utilizado por *Mictlantecuhtli* y *Tepeyóllotl*.

Implementando los obstáculos

Una de las características de un juego de plataformas es la existencia de diferentes obstáculos que el jugador debe de superar por medio de saltos **Ref_JuegoDisenio**. En *Yolotl* se diseñaron e implementaron diferentes tipos de obstáculos para ofrecer una variedad de retos al jugador, a continuación, se describe cada uno de ellos y cómo fue que fueron implementados en el juego:

- **Plataforma que se mueve:** Es uno de los elementos más comunes de los juegos de plataforma, este obstáculo consiste en una superficie de que se mueve de una posición a otra, ver figura 4.20. dentro del juego se creó la clase *MovingPlatform* para este tipo de obstáculo. *MovingPlatform* tiene por atributos las posiciones a las que se moverá, velocidad a la que se moverá. Para su movimiento la clase hace uso de cuatro vectores de posiciones *pos01*, *pos02*, *startPos* y *nextPos*. *Pos01* y *pos02* son las posiciones límite que alcanzará la plataforma, *startPos* es la posición hacia la que la plataforma inicie su movimiento inicial y *nextPos*

es la siguiente posición a la que se irá la plataforma una vez que haya alcanzado un límite. Manejar el comportamiento de las plataformas móviles con este sistema de posiciones permite que la plataforma pueda tener movimiento horizontal, vertical o diagonal sin la necesidad de reescribir código. Al igual que con los enemigos las plataformas de este tipo tienen un radio de área activa para evitar que su comportamiento se ejecute si no están visibles al jugador. Asignar el movimiento de la plataforma no es suficiente para su correcto funcionamiento, ya que cuando el movimiento de la plataforma es horizontal, ésta se desplaza sin el personaje ya que por sí misma no es capaz de asignarle un movimiento al jugador, por tal motivo fue necesario crear una nueva etiqueta para las plataformas llamada *Platform* y asignar dos nuevos parámetros en las colisiones al jugador una para cuando entra en contacto con el colisionador de la plataforma y otra cuando sale. Cuando el jugador entra en contacto con el colisionador de la plataforma se le asigna un parentesco con la posición de la plataforma, lo que le permite seguir el movimiento de la plataforma, este parentesco se rompe cuando el jugador sale de la plataforma, en la figura 4.21 se puede ver esto en código. Adicionalmente, se utilizó el comando *OnDrawGizmos* para dibujar la trayectoria de la plataforma a fin de facilitar la configuración de las plataformas móviles en la construcción de los niveles, ver figura 4.22.

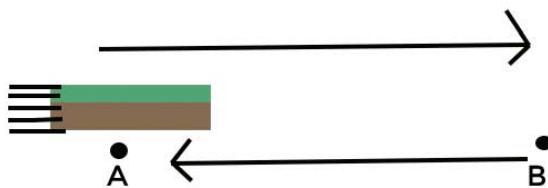


Figura 4.20: Comportamiento de la plataforma que se mueve de manera visual.

```

void OnTriggerEnter2D(Collider2D other){
    if (other.gameObject.CompareTag ("Platform")) {
        player.SetIsJumping (false);
        player.transform.parent = other.gameObject.transform;
    }

    if (other.gameObject.CompareTag ("Ground")) {
        player.SetIsJumping (false);
        //player.SetIsGrounded (true);
        //Debug.Log(player.GetIsJumping());
    }
}

void OnTriggerExit2D(Collider2D other){
    if (other.gameObject.CompareTag ("Platform")) {
        player.transform.parent = null;
    }
}

```

Figura 4.21: Gestión de la respuesta ante diferentes colisiones para garantizar el comportamiento de una plataforma que se mueve de manera horizontal.

- **Plataforma que cae:** Este tipo de plataforma se cae después de que el jugador se posiciona sobre ella. Para evitar que la plataforma caiga instantáneamente una vez que el jugador ha caído sobre ella, un tiempo de retraso se le asigna a la caída.

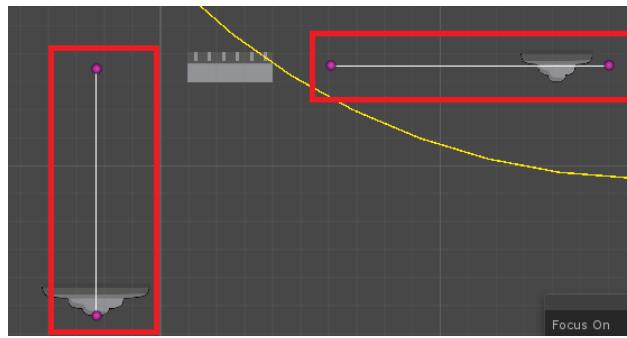


Figura 4.22: Uso del método *OnDrawGizmos* para visualizar la trayectoria de una plataforma que se mueve.

- **Plataforma con más de dos posiciones de control:** esta plataforma puede seguir patrones complejos movimiento como círculos, rectángulos o cuadrados. Su funcionamiento es similar a la plataforma que se mueve con la diferencia de que soporta más de dos posiciones de control; para esto se vale de un arreglo de posiciones en donde el atributo de *nextPos* recorre todo el arreglo de posiciones y al llegar al último elemento del arreglo se le asigna el valor del primer elemento reiniciando el recorrido, ver figura .
- **Viento:** Ese obstáculo crea una corriente de viento que empuja al jugador hacia el vacío, ver figura . Para crear este obstáculo se crean tres clases: *PushingObstacle*, *WindCreator* y *WindHelper*. La primera controla el movimiento del viento a crear. La segunda crea el viento por periodos de tiempo dejando un tiempo de inactividad para que el jugador pueda pasar y la tercera activa al creador de viento cuando el jugador entra en el área activa del obstáculo, cada clase esta instanciada en un *GameObject* diferente. En un principio solo existían las clases *PushingObstacle* y *WindCreator*, lo que provoca que el creador de viento cree viento aun cuando el obstáculo no es visible para el jugador; esto ocasiona la creación de muchos *GameObjects* innecesarios para el viento, por tal motivo se crea la clase *WindHelper* para controlar la activación del creador de viento. Para definir los periodos de creación de viento y de pausa de viento es necesario probar diferentes valores para asignar los tiempos de creación y de pausa del viento. Luego de varias pruebas se definen los siguientes tres valores: 4 para el tiempo activo de creación, 8 para el tiempo de pausa de viento y 0.4 para la pausa entre creación de instancias de viento.
- **Estalagmitas:** Este obstáculo se cae e infringe daño en el jugador cuando éste pasa por debajo del obstáculo, ver figura . Para implementar este obstáculo se crean dos clases: *Stalagmite* y *StalagmiteCtrl*. La primera clase gestiona la caída del objeto y el daño que le infringe al jugador si choca con este o la destrucción del objeto en caso de que choque con el suelo. La segunda clase se encarga de indicarle a la clase *Stalagmite* que el jugador va a pasar bajo de ella para que inicie su caída. En la figura 4.23 se puede ver la configuración de ambos objetos dentro de un nivel.
- **Obstáculo que hace daño:** este obstáculo infringe daño al jugador cuando éste hace contacto con él y no puede ser destruido por el mismo. Este tipo de obstáculo se puede encontrar en el segundo nivel en la etapa de plataforma. Para su implementación se crea la clase *DamageObstacle* y esta gestiona el daño que infringe el obstáculo pudiendo generar obstáculos que causen más o menos daños que otros.

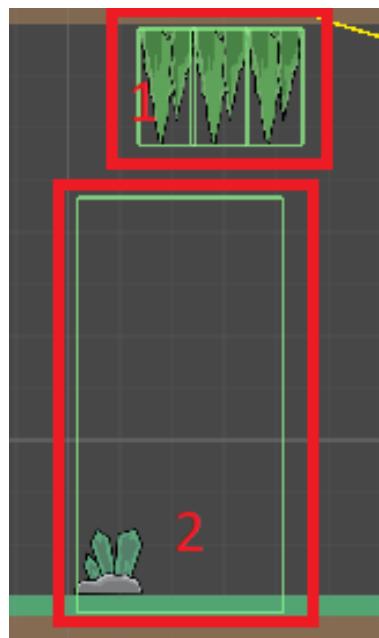


Figura 4.23: Configuración de los diferentes *GameObjects* que conforman el obstáculo estalagmita.

- **Xólotl en su forma Colibrí:** Este obstáculo tiene un comportamiento parecido a la plataforma con más de dos posiciones de control anteriormente descrita, con la diferencia de que su movimiento describe una línea y no un circuito cerrado; por otro lado, al morir el jugador este obstáculo regresa a su posición inicial. Este obstáculo aparece únicamente en el nivel 6, donde el jugador deberá cruzar distintos segmentos del mapa sobre este obstáculo y tendrá que vencer a los enemigos que vayan apareciendo para avanzar, ver figura .

Implementando los ítems y los objetos colecciónables

Las ultimas clases actoras en ser implementadas fueron los ítems, los objetos colecciónables y los puntos de guardado, ya que en el caso de los ítems y los puntos de guardado se necesitaba que el jugador sufriera daño, gastara *tonalli* o muriera ya fuera a causa de enemigos u obstáculos.

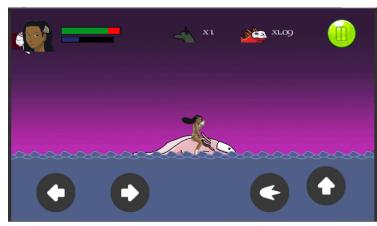
Dentro del juego existen dos tipos de Items: Los que recuperan cantidad de vida y los que recuperan cantidad de *tonalli*. Ambos tienen un funcionamiento similar: como atributos sus respectivas clases tienen una cantidad de lo que van a restaurar (sea *tonalli* o vitalidad), al hacer contacto con el jugador le incrementan dicho atributo en la cantidad que tienen asignada y se destruyen mostrando un efecto de brillos y un efecto de sonido.

En lo que se refiere a los objetos colecciónables dentro del juego se crea la clase *CollectableObjects*, clase encargada de destruir los objetos colecciónables una vez que el jugador los ha tocado, dejando la tarea de actualizar el marcador al jugador. Para poder lograr la actualización del atributo *score* del *Player* se asignaron etiquetas para los objetos colecciónables y dependiendo de dichas etiquetas se gestiona la actualización de los marcadores, esto debido a que la función de los objetos colecciónables depende directamente del nivel en el que se encuentre el jugador; por ejemplo, para los perros que aparecen en el segundo nivel, hacer contacto con uno de ellos no solo actualiza el marcador sino también incrementa el poder que tendrá el Jefe de este nivel, por lo tanto

la actualización visual del marcador deberá incluir cuantos perros se han tocado y en cuanto se ha incrementado el poder del jefe(ver figura 4.24); mientras que en el nivel 4, los coleccionables son llaves y su función es la de desbloquear la transición al siguiente nivel solo si se han juntado todas las llaves; visualmente esta actualización solo requiere de la actualización de un elemento en pantalla (ver figura 4.24). Para solucionar esto se crearon dos etiquetas para cada tipo de coleccionable: *CollectableDog* y *CollectableKey*. Dejando al gestor de colisiones de la clase *Player* verificar por medio de la etiqueta de qué tipo de coleccionable se trata e invoca al controlador de la interfaz gráfica de los marcadores del juego o HUB.



(a) El marcador referente a los perros se encuentra en cero, ya que el jugador no ha tocado ninguno de estos.



(b) Al tocar un perro el marcador referente al conteo de ellos se actualiza junto al del poder del jefe.

Figura 4.24: Interacción entre el jugador y los objetos coleccionables del nivel.

Implementación de los puntos de guardado

Los puntos de guardado o *checkpoints* son los actores encargados de hacer aparecer al jugador en una posición del nivel una vez que éste haya muerto, evitando que el jugador pierda su progreso dentro del nivel. Un *checkpoint* se activa cuando el jugador hace contacto con éste. La funcionalidad de este actor esta dada por la clase *CheckPoint*, esta clase tiene como atributos(ver figura 4.25):

- ***isActive***: Atributo booleano. Cuando es verdadero indica que el jugador ha pasado por el checkpoint. Solo puede haber un único checkpoint activo.
- ***CheckPointsList***: Arreglo de GameObjects que instancian la clase CheckPoint.

Cuando el jugador entra en contacto con el *checkpoint*, éste manda a llamar el método *ActivateCheckPoint*, este método toma todos los elementos que hay en *CheckPointsList* y vuelve su atributo *isActive* falso, después vuelve verdadero el atributo *isActive* del *checkpoint* que fue tocado por el jugador. Cuando el jugador muere se manda a llamar el método *GetActivePosition*, este metodo

busca el *checkPoint* activo y revive al jugador en esa posición. En la figura 4.26 se muestran los métodos de esta clase.

Existen dos tipos de *checkPoints*:

- ***CheckPoint vacío***: este *checkPoint* no tiene ningún sprite asignado y se encuentra al inicio del nivel. Esta activo por defecto y si el jugador muere sin antes tocar algún otro *checkPoint*, la posición de este *checkPoint* es donde aparecerá al revivir.
- ***CheckPoint normal***: Este *checkPoint* tiene un búho como *sprite* y una animación para cuando el jugador interactua con él.

```
public class Checkpoint : MonoBehaviour {
    private Animator anim;
    public bool isActive;
    public List<GameObject> checkPointsList;
    // Use this for initialization
    void Start () {
        checkPointsList = GameObject.FindObjectsWithTag ("Checkpoint");
        anim = GetComponent();
        checkPointsList = checkPointsList.FindAllObjectsWithTag ("Checkpoint");
    }
}
```

Figura 4.25: Atributos de la clase *Checkpoint*

```
private void ActivateCheckPoint(){
    foreach (GameObject cp in checkPointsList) {
        cp.GetComponent <Checkpoint> ().setIsActive (false);
    }
    isActive = true;
}

private void setIsActive(bool value){
    isActive = value;
}

private bool getIsActive(){
    return isActive;
}

public static Vector3 GetActivePosition(){
    Vector3 activePosition = new Vector3 (0f, 0f, 0f);
    if (checkPointsList != null) {
        foreach (GameObject cp in checkPointsList) {
            if(cp.GetComponent <Checkpoint> ().getIsActive()){
                activePosition = cp.transform.position;
                break;
            }
        }
    }
    return activePosition;
}

void OnTriggerEnter2D(Collider2D other){
    if (other.gameObject.CompareTag ("Player")){
        anim.SetInt ("state", 1);
        ActivateCheckPoint ();
    }
}
```

Figura 4.26: Métodos de la clase *Checkpoint*

Cierre del sprint

Al termino de este *sprint* se cuenta con todos los *Assets* de los actores organizados en diferentes carpetas como *Player*, *Obstacles*, *Enemies*, etc. Este *sprint* se cierra con éxito y deja todos los elementos listos para la construcción de las clases controladoras.

4.1.15. Sprint plan 7

SprintID

07

Inicio

29 Enero 2018

Fin

18 Febrero 2018

Meta

Nivel 7

Porcentaje

El 16 %

FeatureID

07

Nombre

N7

Estado

Planeado

SprintID

07

Comentarios

-

4.1.16. Sprint backlog 7**Sprint backlog**

07

Tareas

14

Tendencia

0

Esfuerzo restante

56

Tendencia actual

-

Progreso ideal

14

Tareas restantes

14

Nombre de la tarea

n7

FeatureID

07

Miembro

Rocío

Rol

Desarrollo

Estado

Planeado

Esfuerzo

14

4.1.17. Task Slips 9**FeatureID**

7

Nombre

Maqueta

Tarea

Realizar maqueta del nivel completo

Miembro

Rocío

Esfuerzo estimado

1

Terminado

sí

Restante

0

FeatureID

7

Nombre

Enemigos

Tarea

Realizar el comportamiento de los enemigos o cualquier otra acción necesaria dentro del nivel

Miembro

Rocio

Esfuerzo estimado

5

Terminado

sí

Restante

0

FeatureID

7

Nombre

Arte 1

Tarea

Crear vista de los personajes

Miembro

Rocío

Esfuerzo estimado

2

Terminado

sí

Restante

0

FeatureID

7

Nombre

Arte 2

Tarea

Crear el diseño de los obstáculos

Miembro

Rocío

Esfuerzo estimado

2

Terminado

sí

Restante

0

FeatureID

7

Nombre

Arte 3

Tarea

Crear el diseño de los items

Miembro

Rocio

Esfuerzo estimado

2

Terminado

Sí

Restante

0

FeatureID

7

Nombre

Boss 1

Tarea

Diseñar la máquina de estados del enemigo principal del nivel

Miembro

Rocio

Esfuerzo estimado

2

Terminado

Sí

Restante

0

FeatureID

7

Nombre

Boss 2

Tarea

Crear la máquina de estados del enemigo principal del nivel

Miembro

Rocio

Esfuerzo estimado

2

Terminado

sí

Restante

0

4.1.18. Octavo *sprint* de producción

En este *sprint* se implementa en código la funcionalidad de todos los elementos controladores del juego y sus auxiliares, exceptuando a los controladores de los niveles ya que éstos se desarrollarán en el décimo *sprint* dado que se requieren los niveles terminados para comprobar su funcionalidad.

Controlador de Audio

Este controlador es el encargado de gestionar los efectos de sonido del juego. Logra su funcionamiento con ayuda de la clase controladora *AudioCtrl* y de la clase auxiliar *PlayerAudio*.

La clase *PlayerAudio* está compuesta de los atributos del tipo *AudioClip*. Por convención los archivos de audio que *Unity* soporta pueden ser del tipo *MP3*, *OGG*, *WAV*, entre otros, en relación a tamaño del archivo y calidad de audio se opta por utilizar los archivos de tipo *OGG*. Para la importación de los archivos de audio se elige la configuración que se muestra en la figura 4.27, ya que ésta se disminuye el tiempo de carga de los audios y el espacio en memoria de almacenamiento que utilizan dentro de la aplicación. *PlayerAudio* contiene cinco pistas de audio correspondientes a los efectos de sonido cuando:

- El jugador hace contacto con un objeto coleccional, como una llave o un *xoloitzcuintle*.

- El jugador hace contacto con un ítem.
- El jugador dispara *tonalli*.
- Un enemigo es destruido.
- El jugador muere.

PlayerAudio se vale de la serialización para su funcionamiento (ver figura 4.28), esta clase funge como un organizador de elementos para su visualización en el *GameInspector*, como se ve en la figura 4.29.

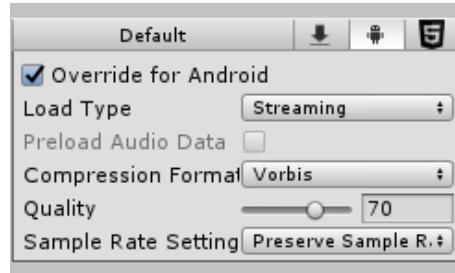


Figura 4.27: Configuración de la importación de los archivos de audio.

```
[Serializable]
public class PlayerAudio{
    public AudioClip CollectablePickup;
    public AudioClip ItemPickup;
    public AudioClip PlayerShoot;
    public AudioClip EnemyExplosion;
    public AudioClip PlayerDies;
}
```

Figura 4.28: Serialización de la clase *PlayerAudio*

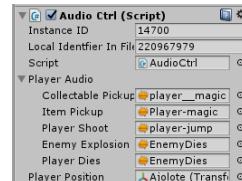


Figura 4.29: Atributos organizados en el *GameInspector* gracias a la serialización de la clase *PlayerAudio*.

Por su parte la clase *AudioCtrl* tiene como atributos la clase *PlayerAudio*, un vector de tres dimensiones para la posición del personaje que manda a llamar los métodos que invocan los audios y una instancia de la misma clase *PlayerAudio* para la perseverancia de datos. Los métodos que contiene la clase son para invocar determinado audio, para esa funcionalidad se utiliza el método *PlayClipAtPoint*, el cual necesita como atributos un objeto de tipo *AudioClip* y un vector de tres dimensiones. Cada método que invoca un efecto de audio es mandado a llamar en la parte del

código que implementa dicha funcionalidad, por ejemplo en el método *SetHealth* del la clase *Enemy* se agrega la linea de código que invoca el audio para la destrucción de un Enemigo en donde se maneja la muerte del enemigo. En la figura se pueden ver los atributos y los métodos de la clase *AudioCtrl*.

```
public class AudioCtrl : MonoBehaviour {
    public static AudioCtrl instance; // For calling the methods in this script
    public PlayerAudio playerAudio; // for accessing player audio effects
    public Transform playerPosition; //The player position is needed for using its audio effects

    void Start () {
        if (instance==null) {
            instance = this;
        }
    }

    public void PlayerShoot(Vector3 playerPos){
        AudioSource.PlayClipAtPoint (playerAudio.PlayerShoot, playerPos);
    }

    public void ItemPickup(Vector3 playerPos){
        AudioSource.PlayClipAtPoint (playerAudio.ItemPickup, playerPos);
    }

    public void CollectablePickup(Vector3 playerPos){
        AudioSource.PlayClipAtPoint (playerAudio.CollectablePickup, playerPos);
    }

    public void PlayerDies(Vector3 playerPos){
        AudioSource.PlayClipAtPoint (playerAudio.PlayerDies, playerPos);
    }

    public void EnemyDies(Vector3 enemyPos){
        AudioSource.PlayClipAtPoint (playerAudio.EnemyExplosion, enemyPos);
    }
}
```

Figura 4.30: Atributos y métodos de la clase *AudioCtrl*.

Controlador de Diálogos

Este controlador se emplea en la cinemáticas del juego. El controlador de Diálogos se vale de cuatro clases para su funcionamiento: *Dialogue*, *Dialogues*, *DialogueFile* y *DialogueCtrl*.

Las clases *Dialogue* y *Dialogues*, son clases de tipo auxiliar. Ambas clases son serializadas y el objetivo de ellas es el de organizar la información en el *GameInspector*. La clase *Dialogue* tiene como atributos un *string* para almacenar el nombre del personaje al que pertenece el dialogo y una lista de tipo *string* para los diálogos. Por su parte la clase *Dialogues* tiene como atributo una lista de tipo *Dialogue*. El objetivo de la clase *Dialogues* es la de almacenar todos los diálogos de la cinemática. En la figura 4.31 se puede ver la definición de las clases *Dialogue* y *Dialogues*.

La clase *DialogueFile* se encarga de la gestión de los diálogos, al contenerlos y activar la visualización de los mismos por medio de la clase *DialogueCtrl*.

La clase *DialogueCtrl* se encarga de la visualización de los diálogos dentro de la escena. Por tal motivo parte de sus atributos son *GameObjects* del tipo *Text* de la *UI*. La clase *DialogueCtrl* visualiza los diálogos tomando el contenido de la clase *Dialogue* y extrayendo elemento por elemento. De cada elemento extraído se obtiene su lista de diálogos y se almacena esta lista en un *Queue* y se muestra en pantalla cada dialogo. La transición entre diálogos se hace si el jugador oprime el botón de salto, ver figura 4.32. Una vez que el controlador ha mostrado todos los diálogos de un elemento. Los diálogos se visualizan por medio de una animación que muestra carácter por carácter, en la figura 4.33 se muestra el método encargado de mostrar esta animación.

```
[System.Serializable]
public class Dialogue{
    public string name;
    public List<string> dialogues;

    public override string ToString ()
    {
        return string.Format ("{0} : {1}", name, dialogues);
    }
}

[System.Serializable]
public class Dialogues{
    public List<Dialogue> dialogues;

    public void printDialogues(){
        foreach (Dialogue dialogue in dialogues) {
            Debug.Log (dialogue.name);
            Debug.Log (dialogue.dialogues [0]);
        }
    }

    public Dialogue GetDialogue(int index){
        return dialogues [index];
    }
}
```

Figura 4.31: Serialización de las clases *Dialogue* y *Dialogues*.



Figura 4.32: La transición entre diálogos se hace si el jugador oprime el botón de salto, este botón es el que se encuentra encerrado en rojo.

Controlador de los datos del juego

El controlador de datos del juego es el encargado de gestionar los datos relacionados al progreso del jugador, es decir los niveles desbloqueados, la cantidad de vida, la cantidad de *tonalli*, el marcador y la cantidad de daño que hacen los ataques del jugador. Para su funcionamiento, este controlador se vale de las clases *GameData* y *GameDataCtrl*.

La clase *GameData*, es una clase de tipo auxiliar y su función es la de serializar los datos que se van a almacenar en el archivo de los datos del juego, ver figura 4.34. Por su naturaleza esta clase solo contiene la definición de los atributos y sus *getters* y *setters*. Unity maneja un tipo de archivo binario que permite la codificación de los datos para evitar que estos sean modificados por el jugador.

La clase *GameDataCtrl* es la encargada de gestionar la creación del archivo, el cargado de la información y de salvar el progreso del jugador. Esta clase se debe de encontrar presente en todas las escenas del juego, ya que de los datos que ésta obtiene se inicializan los valores de la clase *Player* y se verifican de los niveles disponibles. Unity destruye todos los datos de una escena una

```

IEnumerator TypeSentences(string sentences){
    dialogueText.text = "";
    foreach(char letter in sentences.ToCharArray()){
        dialogueText.text += letter;
        yield return null;
    }
}

```

Figura 4.33: El método *TypeSentences* es el encargado de animar el texto de los diálogos al mostrarlos en pantalla

vez que sale de ésta y crea una nueva, por lo que en cada escena se debería crear una instancia de la clase *GameDataCrl*. Para evitar que se desperdicien recursos en el proceso de creación y destrucción de las instancias de la clase *GameDataCrl* se recurre al patrón de diseño *Singleton*. El patron de diseño *Singleton* garantiza que exista solo una instancia de una clase, lo que para el caso particular de *Unity* se traduce en persistencia de la información entre escenas. En la figura 4.35 se puede observar como se logra el comportamiento de un *singleton* al agregar el método *DontDestroyOnLoad*. El uso de un *sigleton* hace que se tenga una configuración especial para el *GameObject* que contenga a esta clase y garantizar que se tenga una única instancia. El *GameObject* que contenga a la clase *GameDataCtrl* debe de estar únicamente en la escena del menú principal y no debe de encontrarse en ninguna otra. Para probar el funcionamiento de los niveles del juego se puede tener tener un *GameObject* que contenga a la clase *GameDataCtrl* en dichas escenas pero es importante que cuando se construya la *apk* del juego, todos estos *GameObjects* se desactiven o se eliminen para evitar crear conflictos por múltiples instancias de la clase *GameDataCtrl*.

```

[Serializable]
public class GameData{
    private float healthAmount;
    private float tonalliAmount;
    private float damageTonalli;
    private int scoreAmount;
    private bool[] levels;
}

```

Figura 4.34: Los atributos de la clase *GameData* corresponden a los datos que se guardarán en el archivo del progreso del juego por lo que estos atributos se encuentran serializados

Controlador de fin de la partida

El controlador de fin de partida funciona a partir de la clase *GameOverCtrl* y de un *GameObject* de tipo *Panel*. La logica detrás este controlador es la siguiente, en el *Panel* se contienen los botones que tienen la funcionalidad de la interfaz que se muestra cuando el jugador ha muerto dentro del juego, ver figura 4.36. La interfaz de fin de partida esta compuesta de tres botones uno para reiniciar la partida, otro para volver al menú de selección de nivel y el tercero para salir de la aplicación. La clase *GameOverCtrl* se encarga de la funcionalidad de los botones y de hacer aparecer el *panel* cuando el jugador es derrotado. En la figura 4.37 se pueden observar los métodos de la clase *GameOverCtrl*.

```

void Awake(){
    if (instance == null) {
        instance = this;
        DontDestroyOnLoad (gameObject);
    } else {
        Destroy (gameObject);
    }
    bf = new BinaryFormatter ();
    dataFilePath = Application.persistentDataPath + "/game.dat";
    Debug.Log (dataFilePath);
    data = new GameData ();
    //data = new GameData ();
}

```

Figura 4.35: Inicialización de la clase *GameDataCtrl* para garantizar el patrón de diseño *Singleton*.



Figura 4.36: Interfaz que aparece cuando el jugador ha muerto dentro del juego para indicar que ha perdido.

Controlador de la barra de vida y de *tonalli* del jugador

Este controlador se encarga de actualizar gráficamente la cantidad de vida y *tonalli* con la que cuenta el jugador. Para la funcionalidad de este controlador se implementa un *Panel GameObject* para que contenga las barras de vida y tonalli del jugador del jugador. Las barras de vida y de tonalli se crean usando *UI GameObjects* de tipo *Image*. Para la barra de vida se emplean tres *UI GameObjects* de tipo *Image*:

- *Health*: de color verde, que representa la cantidad de vida del jugador.
- *Damage*: De color rojo, representa la cantidad de daño que recibe el jugador, es del mismo tamaño que *Health*.
- *BarHealth*: De color negra, es ligeramente de mayor tamaño que las dos anteriores y funge como borde para la barra de vida. Ver figura 4.38.

La actualización de la barra de vida se consigue redimensionando el tamaño de *Health*, Unity cuenta con el método *localScale* que permite ajustar la dimensión de un *GameObject*. *LocalScale* requiere como argumento un vector de dos dimensiones para indicarle si se hará un ajuste horizontal o vertical del objeto, este método toma el 1 como la máxima dimensión del objeto por lo que si se quisiera que el la barra de vida se redujera a la mitad, se tendría que enviar como argumento al método un vector de (0,5, 1). Para el valor en *x* de la escala a la que se va a redimensionar *Health* basta con dividir la cantidad actual de vida entre la cantidad máxima, es decir *health/maxHealth*.

```

public void GotoLevelSelection(){
    loader.setSceneName ("00_LevelSelector");
    loader.changeScene ();
}

public void Retry(){
    player.PausedForReviving();
}

public void setActive(bool value){
    gameoverPanel.SetActive (value);
}

public void quitApplication(){
    Application.Quit ();
}

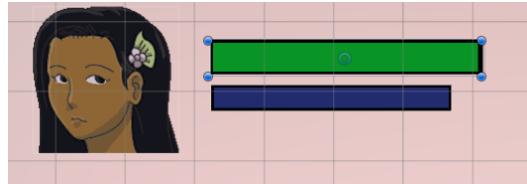
public void RestartSinceBeginning(){
    loader.setSceneName (SceneName);
    loader.changeScene ();
}

public void GoToNextScene(){
    loader.setSceneName (nextSceneName);
    loader.changeScene ();
}

```

Figura 4.37: Métodos de la clase GameOverCtrl

Para actualizar la barra de tonalli se requiere del mismo método. La clase *HealthBar* es la encargada de gestionar el redimensionamiento de las barras de vida y de tonalli. Sus métodos *UpDateHealthBar* y *UpDateTonallithBar* son llamados en la clase *Player* en los métodos *SetHealt* y *SetTonalli*, métodos encargados de actualizar la cantidad de vida y de *tonalli* respectivamente.

Figura 4.38: Barras de vida y *tonalli*.

Controlador del panel de marcadores

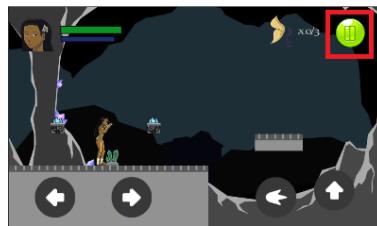
El controlador del panel de marcadores es el encargado de actualizar gráficamente cuantos objetos colecciónables ha juntado el jugador dentro de un nivel, ya sean llaves o *xoloitzcuintles*, ver figura 4.39. Para implementar la actualización gráfica de los marcadores se crea un *panel GameObject* para contener los iconos de los objetos colecciónables y el texto que proporciona la información de los mismos. La clase *HUBCtrl* es encargada de actualizar el texto referente a los marcadores; para tal función se vale de dos métodos:

- ***UpDateDogScore***: responsable de actualizar el conteo de *xoloitzcuintles* y del poder de *Xochitonal* en el segundo nivel.
- ***UpDateKeys***: encargado de actualizar la cantidad de llaves colectadas en el cuarto nivel.

Estos métodos son mandados a llamar por la clase *Player* en su método *OnTriggerEnter2D*, ya que este es el que gestiona las reacciones del jugador y del juego ante las colisiones del jugador.



Figura 4.39: Panel del marcador correspondiente al nivel 4.



(a) Botón de pausa ubicado en la esquina superior izquierda.



(b) Interfaz de partida pausada.

Figura 4.40: Componentes en *GameObject* para la funcionalidad de pausar partida. La funcionalidad de los botones queda dada por : 1, Botón para salir de la aplicación; 2, botón para volver al menú de selección de nivel; 3, botón para quitar la pausa del juego.

Controlador de pausa en el juego

Este controlador permite pausar el nivel que se esta jugando al oprimir el botón de pausa que se encuentra en la esquina superior derecha de la pantalla, ver figura 4.40. Para implementar esta funcionalidad se agrega el botón de *pausa* al panel de marcadores y se crea un nuevo panel parecido al de fin de partida con la diferencia que en lugar del botón de reintentar partida se pone el botón de quitar pausa del juego, ver figura 4.40. Dado que dos de los botones tienen la misma funcionalidad que el panel de fin de partida, los botones para salir de la aplicación y para ir al menu de selección de nivel toman la funcionalidad del *GameOverCtrl*; mientras que el botón de pausa y de botón para quitar la pausa del juego toman su funcionalidad de la clase *PauseCtrl*. La funcionalidad de *PauseCtrl* se basa en el flujo del tiempo de la partida. *Unity* permite controlar la velocidad de la partida con el atributo *timeScale* de la clase *Time*, donde el valor uno es el flujo normal del tiempo y cero es detener la partida. *PauseCtrl* utiliza dos métodos en donde controla la velocidad del juego y desaparece o hace visible el panel de juego pausado; el primer método, llamado *SetGamePaused*, es el encargado de hacer que la velocidad del juego sea igual a cero y de hacer aparecer el panel de juego pausado; el segundo método, llamado *SetGameActive* regresa el juego a su flujo de tiempo habitual al asignarle un valor de uno a *timeScale* y desactiva el panel de juego pausado. En la figura 4.41 se pueden ver ambos métodos en código.

Controlador de efectos especiales

Este controlador muestra dos efectos especiales: una explosion cuando un enemigo o el jugador son derrotados y un destello cuando el jugador toca un ítem o un objeto coleccionable. Para generar el controlador se utilizan dos clases:

- **SFX:** Clase auxiliar que implementa la serialización para agrupar los *assets* que generan los efectos, ver figura. **SFXCtrl:** Clase controladora encargada de instanciar los efectos dentro de la escena dado un vector de posición de tres dimensiones.

```

public void SetGamePaused(){
    pausedPanel.SetActive (true);
    Time.timeScale = 0;
}

public void SetGameActive(){
    pausedPanel.SetActive (false);
    Time.timeScale = 1f;
}

```

Figura 4.41: La pausa se logra con los métodos *SetGamePaused* y *SetGameActive* encargados de controlar el valor del atributo *timeScale* de la clase *Time*.

Como sucede con el controlador de efectos de sonido, los métodos que generan los efectos especiales de la clase *SFXCtrl* son mandados a llamar dentro de otras clases como *Player* o *Enemy* cuando el jugador o el enemigo muere y el el gestor de colisiones de la clase *Player*.

Cierre del sprint

Este *sprint* se finaliza sin contratiempos y logrando el objetivo del mismo, que es generar los controladores comunes de los niveles. Por tal motivo, al finalizar el octavo *sprint* se cuentan con los controladores y actores suficientes como para construir los niveles restantes.

Para iniciar la construcción de los niveles se crea una escena base que contenga todos los controladores y al personaje jugable totalmente configurado con sus clases auxiliares, ver figura 4.42. Se decide crear esta escena para no invertir tiempo en la creación y configuración de los elementos comunes de los niveles. Con esta escena creada se dejan preparadas todas las herramientas para el siguiente *sprint*.

4.1.19. Sprint plan 9

SprintID

09

Inicio

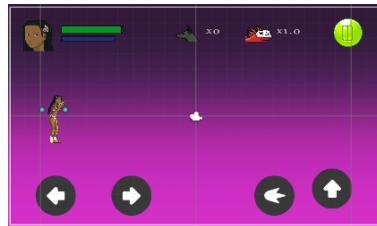
19 Marzo 2018

Fin

16 Abril 2018

Meta

Nivel 9



(a) Vista de la escena base del juego.



(b) Vista de los objetos que contiene la escena base del juego desde la pestaña de jerarquía.

Figura 4.42: Vista de la escena base que contiene todos los controladores y al personaje jugable configurado.

Porcentaje

El 16 %

4.1.20. Feature log 9**FeatureID**

09

Nombre

N9

Estado

Planeado

SprintID

09

Comentarios

-

4.1.21. Sprint backlog 9**Sprint backlog**

09

Tareas

14

Tendencia

0

Esfuerzo restante

56

Tendencia actual

-

Progreso ideal

14

Tareas restantes

14

Nombre de la tarea

n9

FeatureID

09

Miembro

Rocío

Rol

Desarrollo

Estado

Planeado

Esfuerzo

14

4.1.22. Task Slips 9**FeatureID**

9

Nombre

Maqueta

Tarea

Realizar maqueta del nivel completo

Miembro

Rocío

Esfuerzo estimado

1

Terminado

sí

Restante

0

FeatureID

9

Nombre

Enemigos

Tarea

Realizar el comportamiento de los enemigos o cualquier otra acción necesaria dentro del nivel

Miembro

Rocio

Esfuerzo estimado

5

Terminado

sí

Restante

0

FeatureID

9

Nombre

Arte 1

Tarea

Crear vista de los personajes

Miembro

Rocío

Esfuerzo estimado

2

Terminado

sí

Restante

0

FeatureID

9

Nombre

Arte 2

Tarea

Crear el diseño de los obstáculos

Miembro

Rocío

Esfuerzo estimado

2

Terminado

sí

Restante

0

FeatureID

9

Nombre

Arte 3

Tarea

Crear el diseño de los items

Miembro

Rocio

Esfuerzo estimado

2

Terminado

sí

Restante

0

FeatureID

9

Nombre

Boss 1

Tarea

Diseñar la máquina de estados del enemigo principal del nivel

Miembro

Rocio

Esfuerzo estimado

2

Terminado

sí

Restante

0

FeatureID

9

Nombre

Boss 2

Tarea

Crear la máquina de estados del enemigo principal del nivel

Miembro

Rocio

Esfuerzo estimado

2

Terminado

sí

Restante

0

4.1.23. Décimo *sprint* de producción

El objetivo del décimo *sprint* es integrar los actores y controladores del juego para la creación de niveles completos y funcionales. De igual forma se implementa la funcionalidad faltante del menú principal y se construye el menú de selección de nivel.

Al cierre del octavo *sprint* se generó una escena base para construir el resto de los niveles faltantes; por lo que en las siguientes secciones de este *sprint* se abordan los pasos generales para la construcción de un nivel en su totalidad. Estos pasos se repiten para la creación de los niveles pero por la extensión del documento y para facilitar la lectura solo se hablan de los pasos para un solo nivel pero es importante que se considere que fue más de un nivel el que se construyó en este *sprint*.

Construcción del escenario

Lo primero que se hace durante la creación de un escenario es la creación del suelo. En los primeros prototipos se emplearon dos métodos diferentes para esta tarea:

- **Lever Builder:** esta herramienta permite maquetar niveles. Cuenta con tres modos:
 - Este modo sustituye el icono *Print Mode* del cursor por una mano y un cuadrado. Este permite agregar *sprites* oprimiendo el botón derecho y arrastrando el cursor como si se tratara de un pincel. Para poder agregar *sprites* es necesario agregarlos anteriormente como componentes, ver figura 4.43.
 - **Collider Mode:** Utilizando el cursor de cuadro, se seleccionan los componentes a los que se desea agregar un colisionador. En este modo también se puede configurar el tipo de colisionador que se desea agregar y la orientación del mismo (vertical u horizontal), ver figura 4.44.

- **Selection Mode:** *Level Builder* gestiona los *sprites* a partir de generar parentescos con objetos base que agrupan los *sprites*; por ejemplo, los *sprites* referentes al suelo son hijos de un *GameObject* llamado *Ground*. En este modo se puede cambiar el objeto padre de los componentes seleccionados, duplicarlos, encontrarlos en la pestaña de jerarquía, etc.

El principal problema que se tuvo con esta herramienta fue que no agilizaba el maquetado de los niveles significativamente, llegando incluso a complicar más la creación del suelo si por algún motivo se debía de modificar el parentesco entre los componentes del suelo. Otro inconveniente que presentaba esta herramienta es que requería de configuraciones extras para generar la *APK* del juego.

- **Acomodo manual de los Sprites:** Esto consistía en acomodar los *sprites* de manera manual los *sprites* dentro de una escena. Esto no solo era tardado y laborioso sino que también no proporcionaba una maquetación tan exacta ya que en teléfonos de alta resolución se podían apreciar espacios entre los *sprites* que componen el suelo.

La alternativa que se utiliza para estos dos métodos en la creación del escenario es utilizar un nuevo tipo de *GameObject* llamado: *Tilemap*. Este objeto crea una grilla virtual sobre la que se arrastraran los *sprites* que se deseen agregar. El tamaño de los cuadros que componen la grilla puede ser configurado modificando los valores del atributo *Tile Anchor*, para este proyecto el valor para *X* y *Y* es de 0,5. Para poder agregar *sprites* a la grilla es necesario crear primero un *Tile Palette*. Para crear un *Tile Palette* se dirige el cursor a la pestaña de *Tile Palette* y se da click en la opción de *Create new palette*, ver figura . Una vez hecho esto se va a abrir un cuadro de dialogo en donde se le asigna un nombre al *palette* creado, para este proyecto el *palette* tiene por nombre *Ground*, ver figura . Elegido el nombre, se da click en el botón de *create* y se abre una nueva pestaña en la que se debe seleccionar la locación donde se va a guardar el *palette*, en este caso se elige una carpeta llamada *Tiles*. Como paso final se deben de agregar *sprites* al *palette*, esto se logra arrastrando los *sprites* que se desean agragar a la grilla de la pestaña *Tile Pilette*, ver figura. Con estos pasos completados se procede a pintar el nivel. *Unity* ofrece diferentes herramientas para llenar la grilla, tales como: el pincel que permite poner un *sprite* cuadro por cuadro; *Filled Box* que permite llenar áreas enteras; el borrador, que permite eliminar *sprites*, etc. En la figura se muestra el uso de la herramienta *Filled Box* para dibujar una zona del suelo del nivel. Tomando la maqueta del nivel como base, se pinta todo el suelo.

Es posible asignar un colisionador como componente a un *GameObject* de tipo *Tilemap*. Primero se agrega el componente *Rigidbody 2D* y se configura como del tipo *Kinematic*. Luego se agrega el componente *Composite Collider 2D*. Finalmente se agrega el componente *Tilemap Collider 2D*, en este componente se activa la casilla de *Used By Composite*. Con esta configuración *Unity* genera los colisionadores de manera automática según las superficies utilizadas por los *sprites* que hayan sido puestos en la grilla de la escena; lo anterior permite que el colisionador se actualice y adapte según se vayan agregando o quitando elementos a la grilla. En la figura se puede observar el colisonador de un nivel generado con estos componentes.

Terminado el suelo del nivel, se agregan los actores del nivel, es decir los enemigos, puntos de guardado, obstáculos, ítems y objetos colecciónables, en caso de que el tenga. Todos estos componentes únicamente se arrastran a la escena ya que en los *sprints* anteriores fueron configurados para ser *assets*. Una vez agregados a la escena, lo que queda es configurar el tamaño del area activa para

las plataformas de algunos objetos, la cantidad de daño que los enemigos y algunos objetos pueden infringir y la cantidad de vida y *tonalli* que los ítems pueden restaurar.

Finalmente se agregan alguno sprites que corresponden a objetos de fondo. Por ejemplo en el nivel cuatro en la zona de plataformas hay antorchas que son objetos meramente decorativos, ver figura .

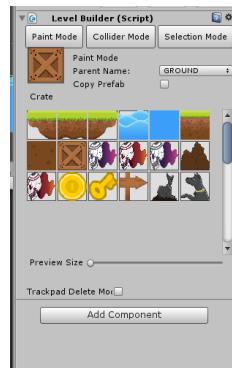


Figura 4.43: Pestaña del *Print Mode* de la herramienta *Level Builder*.

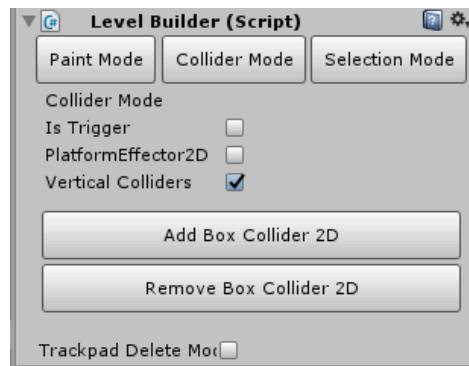


Figura 4.44: Pestaña del *Collider Mode* de la herramienta *Level Builder*.

Configuración de la cámara

Para lograr que la cámara siga al jugador se utiliza un *asset* llamado *Cinemachine*. En los prototipos anteriores se había empleado una clase llamada *CameraCrl*, esta clase permitía que la cámara siguiera al jugador pero no lograba marcar límites para el movimiento de la cámara, lo que provocaba que la cámara mostrara espacios vacíos. *Cinemachine* permite solucionar ese problema de una manera bastante sencilla pero primeramente se va a iniciar por hablar sobre su integración en el proyecto.

Cinemachine es una herramienta desarrollada por *Unity*; no obstante, no viene por defecto con la versión 2017 del software y es necesario descargarla desde la tienda de *assets*, ver figura . Una vez que se descarga se debe de desempaquetar, ver figura . Para verificar que se ha integrado esta herramienta al proyecto en la barra de tareas principal debe de aparecer la opción *Cinemachine*, ver figura ; de igual forma en las carpetas del proyecto debe de mostrarse una carpeta con el mismo



Figura 4.45: Encerradas por recuadros rojos se pueden ver las antorchas que decoran el nivel cuatro en la zona de plataforma.

nombre, ver figura.

Para crear una cámara con *cinemachine*, se debe dar click en la opción de *cinemachine* de la barra principal de tareas y seleccionar la opción de *Create 2D camera*, ver figura 4.46. Lo anterior va a crear una cámara virtual en la pestaña de jerarquía, ver figura . Para vincular la cámara virtual con la cámara principal se debe de agregar el componente de *Cinemachine Brain* a la cámara principal. Lo que resta es configurar la cámara virtual para que siga al jugador. Se arrastra el *GameObject* del jugador de la pestaña de jerarquía al atributo de *Follow* de la cámara virtual; esto le indica a la cámara cual va a ser el objeto de debe de seguir. Al seleccionar la cámara virtual, se puede observar en la vista de *Game* que la pantalla tiene dos zonas una azul y otra roja; la zona azul delimita el área en la que el jugador puede moverse sin que la cámara lo siga y una vez que se aproxima a la zona roja, la cámara comienza a seguirlo, ver figura 4.48. El tamaño de estas zonas puede ser modificado con los valores de los atributos *Screen X*, *Screen Y*, *Dead Zone Width* y *Dead Zone Heidth*, ver figura 4.49.

Para finalizar la configuración de la cámara se crea el área que contendrá la cámara, es decir el área de la que la cámara no puede salir. Para tener un area de contención, primero se debe de crear un *GameObject* vacío nuevo, para este proyecto este *GameObject* se llama *LimitCamera*. A *LimitCamera* se le agrega el componente *Polygon Collider*, se configura el colisionador como de tipo *Trigger* y se modifica su forma para que se adapte al mapeado del nivel, en la figura se muestra un ejemplo de la forma que se le dio a este *GameObject*. Finalmente se agrega el componente *Cinemachine Confiner* a la cámara virtual y se arrastra a *LimitCamera* al atributo *Bounding Shape 2D*, ver figura 4.50. Con esto se logra que los límites de *LimitCamera* sean los límites de la cámara.

Creando el controlador del nivel

El controlador del nivel es aquel que se encarga de verificar que el jugador haya cumplido con el objetivo del nivel y en caso de que lo haya cumplido el controlador debe de desbloquear el nivel siguiente y guardar el progreso del jugador. En total existen cuatro tipos de controladores de nivel:

- El controlador que solo verifica que el jugador haya llegado al final del nivel.
- El controlador que verifica que el jugador haya llegado al final del nivel y haya recolectado una cantidad determinada de objetos colecciónables pero no guarda el marcador.

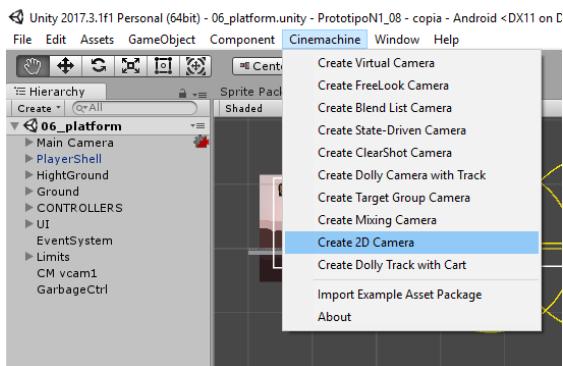


Figura 4.46: Creación de una cámara de tipo *cinemachine*.

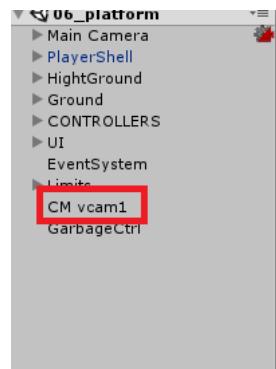


Figura 4.47: Cámara virtual desde la pestaña de jerarquía.

- El controlador que verifica que el jugador haya llegado al final del nivel y guarda el valor del marcador.
- El controlador que verifica que el enemigo de tipo jefe haya sido eliminado.

El primer tipo de controlador es empleado en la zona de plataforma de los niveles seis y ocho. El segundo se emplea en la zona de plataforma del nivel cuatro. El tercer tipo se emplea en la zona de plataforma del segundo nivel. Mientras que el último controlador se utiliza para las zonas de jefe de todos los niveles.

En lo que se refiere a la progresión del personaje, es decir al aumento de la vida, cantidad de *tonalli* y cantidad de daño de *Malinalli*, en la figura se muestra la progresión del personaje una vez que el jugador ha terminado un nivel determinado.

Creación de las cinemáticas del juego

En este *sprint* se crean las cinemáticas de los niveles pares. Las cinemáticas del juego son aquellas animaciones que fungen como transiciones entre escenas, su objetivo es el de contar la historia del juego.

Una escena de tipo cinemática tiene la siguiente estructura visual(ver figura 4.51):

- Una imagen que contiene los diálogos.



Figura 4.48: La zona azul corresponde al área que en la que el jugador puede moverse dentro del juego sin que la cámara lo siga.

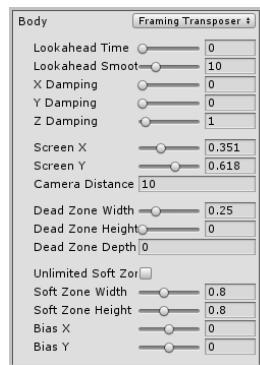


Figura 4.49: Atributos que se pueden modificar para cambiar las dimensiones de la zona azul de la cámara virtual.

- Un objeto de texto para el nombre del personaje.
- Un objeto de texto para el dialogo.
- Un objeto de texto para indicarle al jugador que debe de oprimir el botón de salto para ver el siguiente dialogo.
- Botones que controlan al jugador en los niveles.
- *Sprites* de los personajes que aparecen en esa escena.

Para la creación de la cinemáticas se emplean los diálogos del guión literario desarrollado durante el trabajo terminal 1 y las imágenes correspondientes al diseño de personajes también realizadas durante trabajo terminal 1.

Las clases encargada del controlar el flujo de los diálogos en la cinemática son las clases *DialogueCtrl* y *DialogueFile*. Como se menciona en la sección 4.1.18, estas clases se encargan de la visualización y transición de diálogos.

Agregando la funcionalidad faltante al menú principal

La funcionalidad faltante del menú principal es la verificación de la existencia del archivo que contiene los datos de la partida, en caso de que no exista el archivo crearlo, en caso de que exista

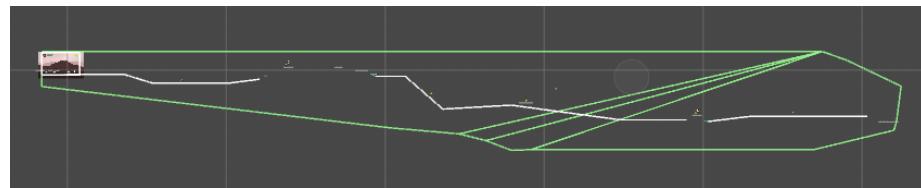


Figura 4.50: Configuración de la forma de *LimitCamera*.

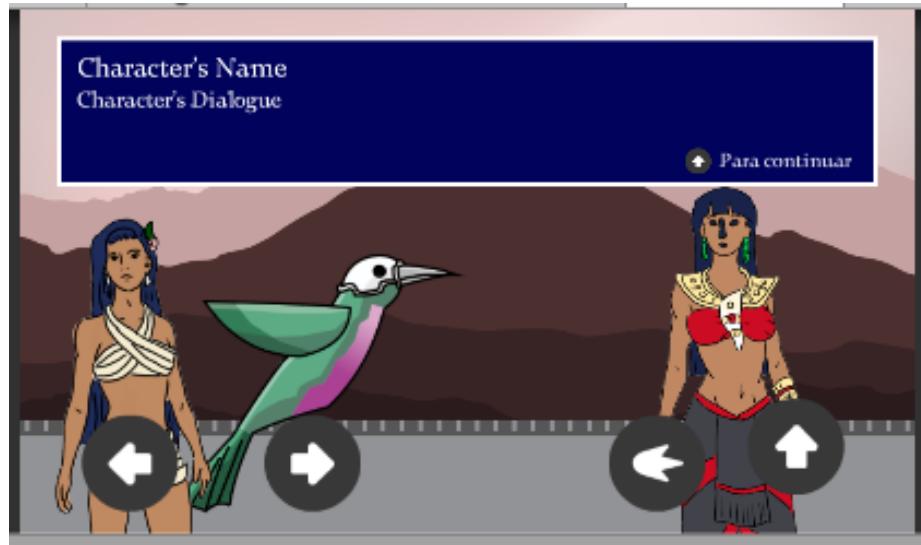


Figura 4.51: Estructura visual de una escena de tipo cinematográfica.

cargar la información y la aparición de los mensajes de confirmación. En los primeros prototipos el menú principal dirigía al jugador y no permitía que el jugador guardará su progreso.

Para conseguir el funcionamiento ya descrito, se crea la clase *MenuCtrl* y se agrega un *GameObject* de tipo *Panel* como el que se ve en la figura 4.52, este panel tiene esa estructura a fin de ser utilizado tanto para el mensaje de advertencia para crear una nueva partida como para el mensaje que notifica que no existe una partida previamente guardada. Cuando el jugador oprime el botón de *Empezar partida*, se manda a llamar al método *OpenWarningWindow*, este método habilita el panel del mensaje y modifica el texto que se muestra en el panel por el: “Advertencia: Se eliminarán todos los datos previamente guardados. ¿Desea crear una nueva partida?”. Si el jugador oprime el botón de aceptar el *MenuCtrl* mandará a llamar el método *CreateNewGameData* de la clase *GameDataCtrl* y cargara la escena del menú de selección de partida. Si el jugador oprime el botón de cancelar, el *panel* de advertencia se cerrará. En caso de que el jugador quiera cargar su partida, éste debe de oprimir el botón de *Cargar partida*, lo que manda a llamar el método *LoadGame* de la clase *MenuCtrl*; el método *LoadGame* verifica si existe el archivo con los datos de la partida, su existe se cargan los datos y se redirige a la pantalla de selección de nivel, en caso contrario se habilita el *panel* con el aviso de que no se encontró ningún archivo con los datos del juego y se le pide al jugador iniciar una nueva partida.

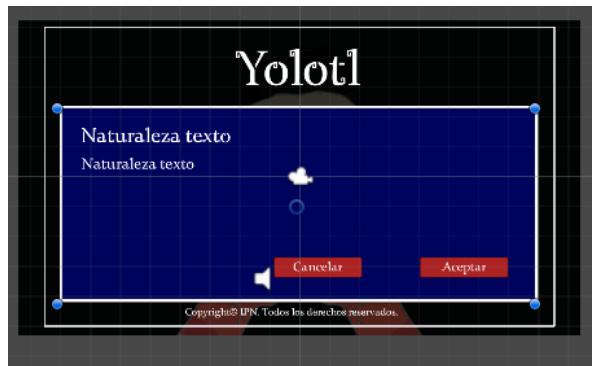


Figura 4.52: *Panel* agregado a la interfaz del menú principal a fin de mostrar los mensajes de confirmación de crear nueva partida o para notificar que no existe una partida previamente guardada.

Creación de la funcionalidad del menú de selección de nivel

El menú de selección de nivel, es el encargado de permitirle al jugador seleccionar que nivel desea jugar. El jugador solamente puede acceder a los niveles que haya desbloqueado.

Para implementar este menú se crea primeramente la interfaz gráfica. Esta se logra con tres botones (dos para navegar y uno para confirmar que nivel se desea seleccionar), tres imágenes (dos para el cuadro de texto donde se muestra la información del nivel y una para la imagen del nivel) y dos objetos de texto (para el título del nivel y la descripción del nivel). En la figura 4.53 se muestra la interfaz del menú de selección de nivel.

Para dar la impresión al jugador de que se encuentra ante una navegación de tipo carrusel, la clase *SelectionManuCtrl*(encargada de la funcionalidad del menú de selección de nivel) se vale de recorrer tres arreglos para modificar el contenido de la interfaz. Los tres arreglos son los siguientes:

- ***LevelInformation***: Arreglo de instancias de la clase auxiliar *LevelDescriptor*, *LevelDescriptor* tiene como atributos dos *strings* correspondientes al nombre del nivel y a la descripción del nivel.
- ***levelImage***: Arreglo de *sprites*, los *sprites* de este arreglo corresponden a las imágenes de los niveles.
- ***levelName***: Este arreglo de *strings* corresponde al nombre de las escenas de los niveles para ser cargadas.

SelectionManuCtrl se vale de dos métodos: *GoNext* y *GoPrevious* para navegar a través de los arreglos y mostrar la información de los niveles si estos han sido desbloqueados por el jugador, de lo contrario muestran por defecto el mensaje de “No disponible”, ver figura 4.54.

Cierre del sprint

Al cierre de este *sprint* se cuenta con los niveles terminados y funcionales así como los menús del juego. Por lo que se deja todo preparado para la realización de pruebas.

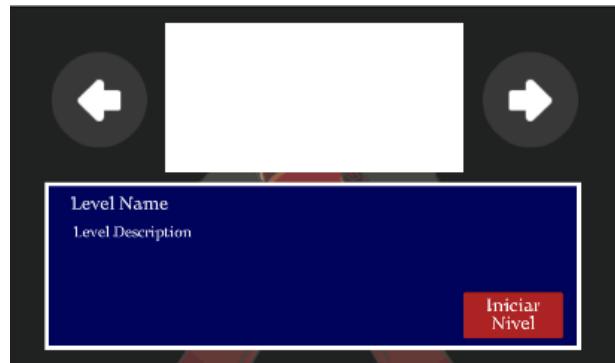


Figura 4.53: Diseño del menú de selección de nivel



(a) Vista del menú de selección de nivel cuando el nivel no ha sido desbloqueado



(b) Vista del menú de selección de nivel cuando el nivel ha sido desbloqueado.

Figura 4.54: Vistas del menú de selección de nivel dependiendo de si el nivel esta o no disponible.

Capítulo 5

Resultados obtenidos

5.1. Pruebas

5.1.1. Sprite packer

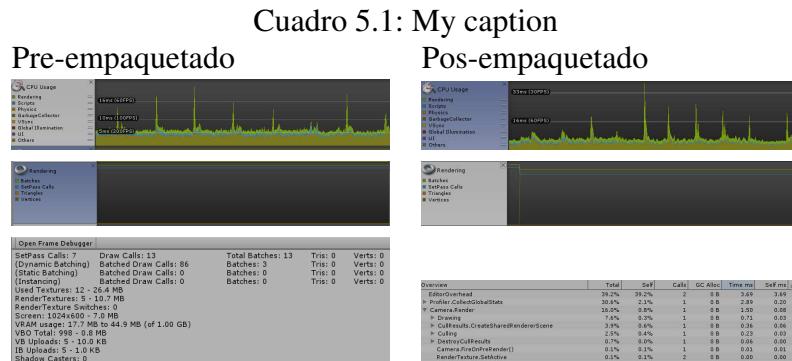
El objetivo de esta prueba es la optimización de carga de imágenes al crear paquetes de sprites, en unity conocido como sprite packer. Se usará pruebas automáticas con la herramienta profiles de Unity.

Una parte significativa de una textura de sprite a menudo será ocupado por el espacio vacío entre los elementos gráficos y este espacio va a resultar en memoria de video gastada en el tiempo de ejecución. Para un rendimiento óptimo, lo mejor es empacar gráficas de varias texturas de sprite muy juntas dentro de una misma textura conocida como un atlas. Unity proporcionar una utilidad Sprite Packer para automatizar el proceso de generar atlases de las texturas de sprite individuales.

Al final podemos determinar que si existe un menor porcentaje de carga en imágenes debido al empaquetado. En este caso el empaquetado fué en sprites estáticos, una mayor percepción de carga menor se puede observar al tener un empaquetado de animaciones, recordando que un movimiento en una imagen puede contener un número determinado de cuadros por segundo. Se deduce que si ha habido una optimización.

5.1.2. Mecánicas del juego

En esta parte se incluirán las pruebas de los componentes jugables que forman parte del juego. Como se desarrollan e interactúan a lo largo del juego.



Salto del jugador

El objetivo de la prueba consiste en verificar la condición inicial del salto, en donde debe realizarse un doble salto. Esta es una prueba funcional del juego, la herramienta a utilizar es el mismo programa de Unity, dentro del computador. Al aplicar la prueba se detecta un salto ilimitado de veces en el jugador, debido a que el valor de verdadero o falso para permitir un salto extra no esta siendo detectado por el método de salto. Se hace un reporte final del fallo de salto extra.

Plataforma en movimiento

El objetivo de la prueba es comprobar que funcione la plataforma con movimiento tanto horizontal como vertical como se ha determinado. La herramienta a utilizar sigue siendo dentro del computador Unity. Como prueba dentro del funcionamiento del juego el jugador se posiciona sobre la plataforma, la plataforma se desplaza de forma adecuada y en la dirección establecida, tanto el movimiento horizontal como el movimiento vertical. Pero se detecta que el desplazamiento es independiente al personaje, por lo que el jugador debe manualmente seguir la trayectoria de la plataforma, en este caso particular, la plataforma horizontal. Mientras que en el caso de la plataforma vertical se detecta una disminución de velocidad al realizar una ascensión y un aumento de velocidad al descender. Se concluye que los componentes funcionan de manera independiente en cuanto a su movimiento, la solución a proponer consistiría en un código por parte de la plataforma en movimiento que cuando detecte al jugador le aplique la misma velocidad que este realice sumándosela a la que pudiera el jugador tener en ese momento y esto sería solo al contacto de la plataforma.

5.1.3. Dinámicas del juego

En esta parte las pruebas van dirigidas a las situaciones que crea el jugador con lo que se le ha permitido dentro del videojuego.

Salto bajo

El objetivo de la prueba consiste en entender la percepción del usuario ante las acciones que realiza el personaje dentro del juego. La herramienta a utilizar es en el computador en el programa Unity. El jugador se presenta en diferentes escenarios de los niveles para que pueda moverse a voluntad propia dentro de lo que establece el juego. Las situaciones a las que se presenta el jugador abarcan saltar en plataformas estáticas, plataformas móviles, camino horizontal y camino ascendente. Al final el usuario reporta que en su propia percepción el salto o elevación del personaje es menor al que le gustaría o en algunas ocasiones al que necesita. Las soluciones a proponer consisten en aumentar la variable que controla el movimiento ascendente del jugador o la variable que controla la gravedad proporcionada en el juego.

Instrucciones del juego

El objetivo de la prueba consiste en entender la percepción del usuario ante las acciones que realiza dentro del nivel introductorio del juego. La herramienta a utilizar es en el computador en el programa Unity. Al usuario se le presenta el nivel introductorio del juego para que termine con este mismo. Al final reporta que por prueba y error ha sabido el funcionamiento de los botones que se le presenta, según a opinión del usuario le gustaría dentro del juego una demostración o explicación de los mismos botones y lo que hacen. Al final se propone como solución dentro de los mismos

diálogos explicar el funcionamiento de los botones, o el realizar un video explicativo de los botones o indicarle al usuario que realice actividades determinadas para la prueba de los botones.

Desbloqueo de personas

El objetivo de la prueba consiste en entender la percepción del usuario ante las acciones que realiza dentro del nivel introductorio del juego. La herramienta a utilizar es en el computador en el programa Unity. Al usuario se le presenta como situación seguir avanzando al siguiente nivel dentro del nivel introductorio, donde en una parte el jugador debe desbloquear al menos cinco diálogos. El jugador reporta que aunque sea el mismo diálogo que active una y otra vez, este se contabiliza sin ningun problema. La solución a proponer es colocar una variable de verdadero y falso para indicar si el diálogo ha sido activado y así no contabilizarlo nuevamente.

5.1.4. Estética del juego

En esta parte las pruebas van dirigidas a la respuesta del jugador ante lo que se le presenta directamente como interacción dentro del juego, ya sea auditiva o visualmente.

Ilustraciones de items

El objetivo de la prueba consiste en entender la percepción del usuario ante lo que se le muestra a lo largo del juego. La herramienta a utilizar es en el computador en el programa Unity. Dentro de los niveles impares el jugador reporta un poco confuso a primera vista el item que incrementa el tonalli, al no reconocer que objeto lo representaba. El objeto a querer mostrar es una flor de vainilla, al mostrar la imagen por separado al jugador y presentarlo en un tamaño más grande el usuario reporta como confuso las ramas u hojas de la imagen a lado de la flor. Como propuesta de solución se tiene la eliminación de las ramas u hojas de la imagen, dejando solo la flor visible o hacer mucho más grande la imagen presente en el juego.

Menú principal

El objetivo de la prueba consiste en entender la percepción del usuario ante lo que se le muestra en el menú principal. La herramienta a utilizar es en el computador en el programa Unity.

5.1.5. Prueba unitaria

Las primera prueba unitaria fue sobre los actores. En esta sección se describe como se realiza la prueba y como se solucionan los errores encontrados a partir de ella.

Objetivo de la prueba

Verificar el funcionamiento lógico de los componentes del nivel y definir los valores a algunos atributos para el funcionamiento correcto de algunos actores.

Herramientas utilizadas durante la prueba

Unity.

Aplicación de la prueba

Para esta prueba se evaluá el comportamiento de los actores antes de su integración a los niveles y como estos interactúan con el jugador. Unity permite ver los valores que adquieren los atributos durante su ejecución, ver figura 5.1. Así que para esta prueba basta con ejecutar la escena base y observar como responden los actores.

Primeramente se revisa que los enemigos y las plataformas sigan sus patrones de movimiento definidos. Después se verifica que los enemigos, obstáculos e ítems afecten la cantidad de vida del jugador y en algunos casos su cantidad de *tonalli*. En el caso de los jefes se verifica que sus ataques no se vean interrumpidos por nuevos ataques de la maquina de estados. En el caso particular de obstáculos como *WindCreator* se definen los intervalos de tiempo para su correcto funcionamiento.

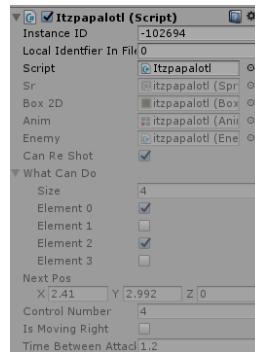


Figura 5.1: Unity permite ver los valores de los tributos de las clases en ejecución.

Conclusiones de la prueba

En esta prueba se observan diferentes problemas en el comportamiento de los actores que se solucionan, a continuación se mencionan los errores encontrados y como se solucionaron:

- El marcador se actualiza al doble cuando el jugador cae sobre un objeto colecciónable: Este error resulta producto de utilizar un *GameObject* auxiliar para la detección de las colisiones del suelo. El error se soluciona fácilmente al agregar un componente de tipo rigidbody 2D al *GameObject* auxiliar para la detección de las colisiones del suelo.
- Los ítems restauran el doble de vida cuando el jugador cae sobre ellos: Este error es generado por las mismas causas que el de los objetos colecciónables así que al solucionar el de los objetos colecciónables se soluciona éste.
- Los ataques de los jefes generados por corutinas se empalman con otros ataques o interrumpen los que ya se están ejecutando: Esto se soluciona al detener todas las corutinas generadas por el jefe cuando se ejecuta un ataque.

5.1.6. Prueba de integración

Esta prueba se realiza una vez se integraron los actores y controladores a los niveles.

Objetivo de la prueba

Verificar el funcionamiento lógico de los componentes del nivel al ser integrados para formar un nivel entero.

Herramientas utilizadas durante la prueba

Unity.

Aplicación de la prueba

Para realizar esta prueba es necesario jugar los niveles para observar que el comportamiento de los controladores y los actores se ejecute correctamente al integrarse con otros actores. En esta prueba también se ajustan las áreas activas de las plataformas a fin de que su funcionamiento no se detenga si se alejan mucho del jugador al realizar su recorrido.

Conclusiones de la prueba

Al finalizar esta prueba se pudo verificar que los controladores funcionan de manera correcta; sin embargo, es necesario realizar ajustes referentes a los tiempos de transiciones entre escenas y los valores de las áreas activas de varias plataformas y obstáculos ya que con sus valores iniciales algunas plataformas se detenían al realizar su recorrido dado que el jugador se salía de su área activa y se volvía inalcanzable. En cuanto al obstáculo de *WindCreator* se ajustó el tamaño del área activa garantizando que el obstáculo se encuentre activo cuando el jugador llegue a donde se encuentra éste.

5.1.7. Prueba de sistema

Esta prueba se realiza una vez se integraron los actores y controladores a los niveles.

Objetivo de la prueba

Verificar el flujo de la navegación del juego.

Herramientas utilizadas durante la prueba

Unity.

Aplicación de la prueba

Esta prueba inicia desde la escena de menú principal en donde se verifica que el controlador del menú realiza las validaciones correspondientes antes de crear o cargar una partida; de igual forma se verifica que aparezcan los mensajes de confirmación a cada caso, sea el de confirmación de la nueva partida o el que notifica que no hay datos previamente guardados.

La siguiente escena a probar es el menú de selección de nivel. En este se verifica que la información mostrada por la interfaz corresponda al nivel que se desea acceder. Después, se verifica que en efecto el juego no permita acceder a niveles que aun no se desbloquean.

Para finalizar la prueba se verifica que se realicen las transiciones entre niveles y cinemáticas. De igual forma se prueba la funcionalidad de botones de navegación de los niveles referentes al panel de pausa, fin de partida y nivel completado.

Conclusiones de la prueba

Al finalizar esta prueba se pudo confirmar que las transiciones entre escenas se realiza de manera correcta; salvo en algunos casos pero fue debido a que el nombre de la escena a la que se debería redirigir no estaba escrito correctamente o no coincidía con el nombre de la escena a la que debía ir. Con esto se puede concluir que se cumple el mapa de navegación que se propuso en el documento de diseño realizado en trabajo terminal 1.

5.1.8. Prueba de rendimiento

Esta prueba se realiza una vez que hechas las modificaciones como producto de las pruebas unitarias, de integración y de sistema.

Objetivo de la prueba

Verificar el uso del GPU.

Herramientas utilizadas durante la prueba

Profiler de Unity.

Aplicación de la prueba

Esta prueba inicia desde el menú principal y con la herramienta profiler se observa el desempeño del GPU de la maquina al simular el juego. La ventaja de utilizar *Profiler* es que indica que elementos de la escena son los que están consumiendo un determinado porcentaje del GPU. En las figuras 5.2, 5.3 y 5.4 se muestran los resultados de la herramienta *Profiler*.

Conclusiones de la prueba

Al observar el desglose del uso del GPU en las diferentes escenas que se probaron, se identifica al *EditorOverHead* como uno de los principales consumidores de recursos; investigando en la documentación de *Unity*, se detecta que este elemento es producto de un error de rendimiento en la versión 2017 pero que se puede solucionar al descargar uno de los parches que *Unity* proporciona desde su sitio web.

Con las pruebas de *profiler* se puede concluir que el juego tiene un buen rendimiento en cuanto a uso de recursos puesto que no presenta caídas dramáticas en cuanto a desempeño.

5.1.9. Prueba de rendimiento

Esta prueba se realiza una vez que hechas las modificaciones como producto de las pruebas unitarias, de integración y de sistema.



(a) Vista general del uso del GPU.

	Total	Self	Calls	GC Alloc	Time ms	Self ms	△
Profiler.CollectGlobalStats	70.1%	0.7%	1	0 B	11.78	0.12	
Profiler.CollectMemoryAllocationStats	59.3%	59.3%	1	0 B	9.97	9.97	
Profiler.CollectAudioStats	9.8%	9.8%	1	0 B	1.65	1.65	
Profiler.CollectDrawStats	0.1%	0.1%	1	0 B	0.02	0.02	
EditorOverhead	23.1%	23.1%	2	0 B	3.88	3.88	
Camera.Render	2.3%	0.3%	1	0 B	0.38	0.06	
PostLateUpdate.UpdateAudio	0.4%	0.0%	1	0 B	0.07	0.00	

(b) Desglose de los porcentajes del uso del GPU

```
Textures: 2186 / 151.5 MB
Meshes: 205 / 1.0 MB
Materials: 45 / 60.0 KB
AnimationClips: 29 / 148.0 KB
AudioClips: 0 / 234.0 KB
Assets: 3383
GameObjects in Scene: 54
Total Objects in Scene: 523
Total Object Count: 3906
GC Allocations per Frame: 150 / 5.0 KB
```

(c) Desglose de los porcentajes del uso de memoria

Figura 5.2: Resultados de la herramienta *profiler* al analizar el menú de selección.

(a) Vista general del uso del GPU.

```
Textures: 2191 / 118.1 MB
Meshes: 32 / 384.0 KB
Materials: 45 / 60.0 KB
AnimationClips: 29 / 148.0 KB
AudioClips: 0 / 0 B
Assets: 3444
GameObjects in Scene: 80
Total Objects in Scene: 454
Total Object Count: 3898
GC Allocations per Frame: 150 / 5.0 KB
```

(b) Desglose de los porcentajes del uso de memoria

Figura 5.3: Resultados de la herramienta *profiler* al analizar una cinemática.



(a) Vista general del uso del GPU.

Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
EditorOverhead	37.8%	37.8%	2	0 B	4.05	4.05
Profiler.CollectGlobalStats	34.6%	1.4%	1	0 B	3.71	0.15
Profiler.CollectAudioStats	22.8%	22.8%	1	0 B	2.45	2.45
Profiler.CollectMemoryAllocationStats	9.6%	9.6%	1	0 B	1.03	1.03
Profiler.CollectDrawStats	0.6%	0.6%	1	0 B	0.07	0.07
GUI:Repaint	6.0%	1.5%	1	1.1 KB	0.64	0.17
Camera:Render	5.5%	0.7%	1	0 B	0.59	0.07

(b) Desglose de los porcentajes del uso del GPU

Used Total: 202.9 MB Unity: 95.5 MB Mono: 18.9 MB GfxDriver: 90.3 MB FMOD: 1.6 MB Video: 0 B Profiler: 17.1 MB
Reserved Total: 377.7 MB Unity: 263.4 MB Mono: 25.1 MB GfxDriver: 90.3 MB FMOD: 1.6 MB Video: 0 B Profiler: 24.0 MB
Total System Memory Usage: 0.93 GB

Textures: 2204 / 117.7 MB
Meshes: 158 / 1.0 MB
Materials: 48 / 63.0 KB
AnimationClips: 29 / 148.0 KB
AudioClips: 8 / 217.0 KB
Assets: 3632
GameObjects in Scene: 222

(c) Desglose de los porcentajes del uso de memoria

Figura 5.4: Resultados de la herramienta *profiler* al analizar un nivel.

Objetivo de la prueba

Verificar el uso del GPU, el nivel de bateria del telefono que utiliza y la temperatura del telefono mientras la aplicación este funcionando.

Herramientas utilizadas durante la prueba

Opciones de desarrollador del telefono Huawei TAG-L13 y *Battery Doctor*.

Aplicación de la prueba

Para esta prueba se debe de instalar la apk del juego en el dispositivo, activar las opciones de desarrollador e instalar la aplicación *Battery Doctor*. Una vez hecho esto se juega el juego y se mide el desempeño desde el teléfono. En las figura 5.5 se muestra el uso del GPU en distintos momentos de la partida.

Conclusiones de la prueba

De esta medición del desempeño del GPU se puede observar que la aplicación utiliza un mínimo del 9 % del GPU y hasta un máximo del casi el 30 %. Por su parte la temperatura del teléfono no se ve afectada manera significativa y en promedio utiliza un 10 % de la batería. Estas cifras son buenas si se considera que otras aplicaciones como *Messenger* de *Facebook* llega a utilizar el 50.6 % del GPU y casi el 60 % de la batería del teléfono, ver figura .



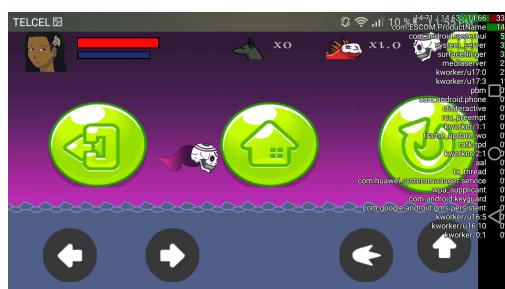
(a) Uso del GPU desde el menú principal.



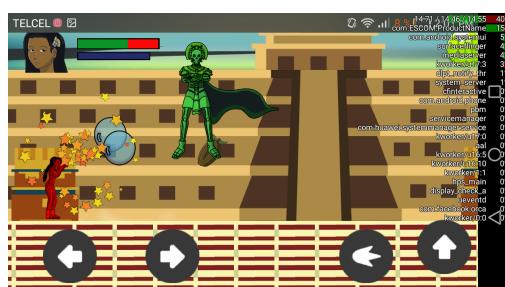
(b) Uso del GPU desde el menú de selección de nivel.



(c) Uso del GPU desde una cinematográfica.



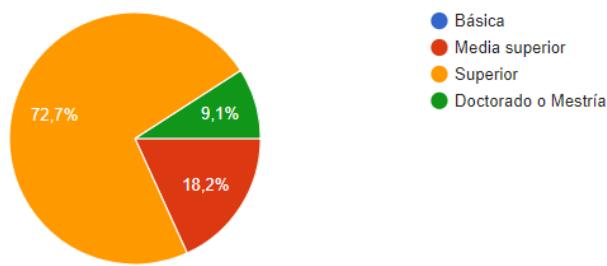
(d) Uso del GPU desde un nivel de plataforma.



(e) Uso del GPU desde un nivel de jefe.

Figura 5.5: Resultados del rendimiento del GPU del dispositivo Huawei

Figura 5.6: Gráfica de nivel de estudios



5.1.10. Cuestionario 4

Pre-juego

¿Cuál es tu edad?

Respuesta: El público en su mayoría que ha respondido la encuesta se encuentra entre los 21 y 25 años de edad y algunas personas entre los 30 y 40 años.

¿Cuál es tú nivel de estudios?

- Básica
- Media superior
- Superior
- Doctorado o maestría

¿Qué es lo primero que piensas cuando te dicen cultura de México?

- Comida
- Arquitectura o lugares
- Vestimenta
- Música
- Idioma

¿Por qué motivos visitas lugares históricos?

- Escuela
- Trabajo
- Evento especial
- Interés propio
- Otro?

Figura 5.7: Gráfica de cultura de México

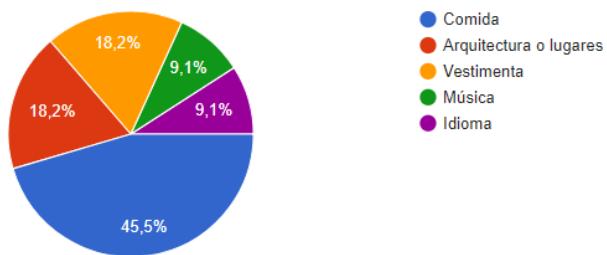
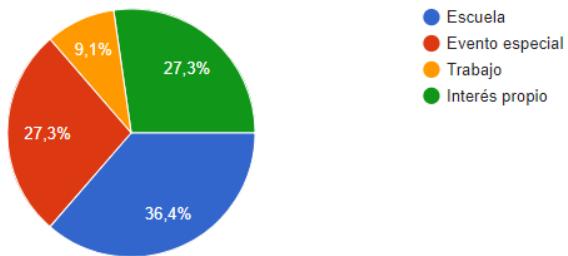


Figura 5.8: Gráfica de lugares históricos



¿Sabes sobre la cultura prehispánica en México?

- Sí
- No
- Creo

¿Qué sentimiento tienes?

- Alegría
- Enojo
- Tristeza

Figura 5.9: Gráfica de cultura prehispánica

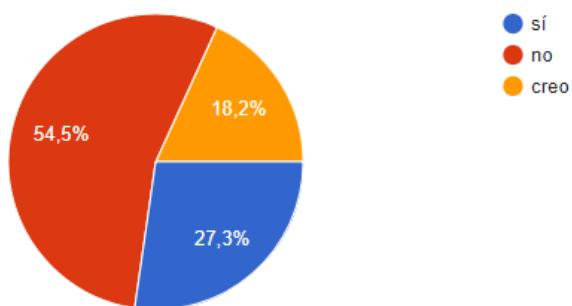
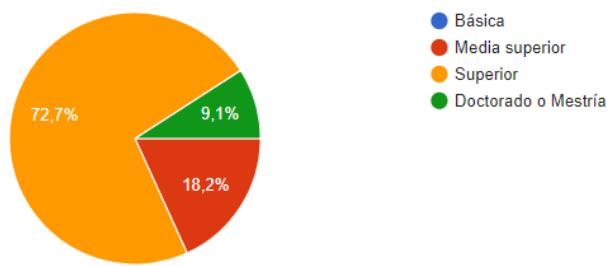


Figura 5.10: Gráfica de sentimiento



- Curiosidad
- Fatiga
- Otro?

Pirámide emocional. ¿En qué escalón te encuentras ahora? La emoción que tenían los encuestados en ese momento era un estado de seguridad o tranquilidad.

Post-juego

changing checkbox style locally ¿Hay algún elemento que no reconozcas o sepas que es?

- Items
- Personajes
- Fondo
- Texto (palabra)
- Lugar
- Tiempo
- Forma
- Otro?

Respuesta: Los encuestados en su mayoría no reconocían diferentes formas o imágenes de objetos presentados en el juego.

¿Cómo califica la velocidad de reacción de los botones?

- Mala
- Regular
- Buena

¿Qué te ha gustado MENOS del juego?

- Juego

Figura 5.11: Gráfica de velocidad de botones

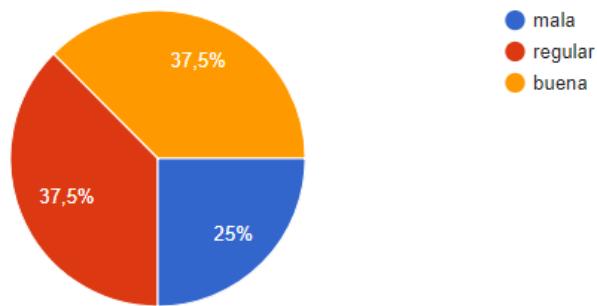
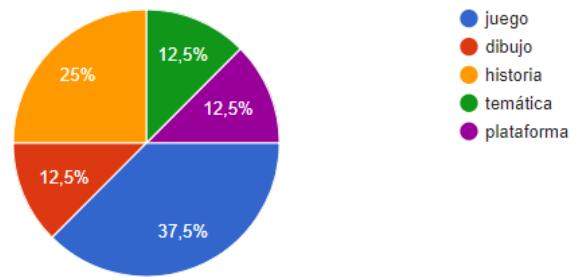


Figura 5.12: Gráfica de menos gusto



- Dibujo
- Historia
- Temática
- Plataforma

¿Qué te ha gustado MÁS del juego?

- Juego
- Dibujo
- Historia
- Temática
- Plataforma

¿Qué dificultad encontraste al jugar? Respuesta: Los encuestados en su mayoría contestaron sobre los movimientos o acciones que se podían realizar dentro del juego, que a su percepción parecían veces impredecibles.

¿Qué cambiarías del juego? Respuesta: Los usuarios en su mayoría mencionan sobre una mayor interacción en el juego sobre el personaje a otras acciones o eventos.

¿Qué sentimiento tienes?

- Alegría

Figura 5.13: Gráfica de más gusto

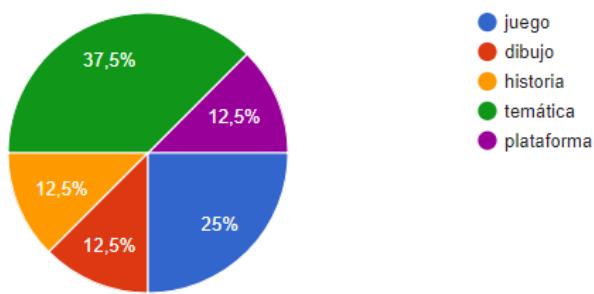
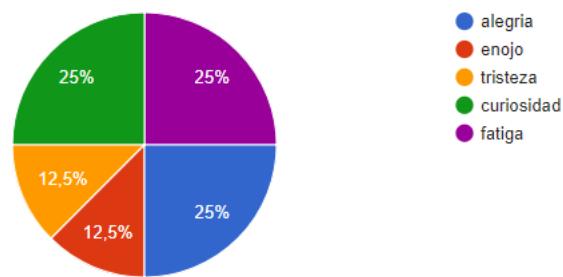


Figura 5.14: Gráfica de sentimiento



- Enojo
- Tristeza
- Curiosidad
- Fatiga
- Otro?

Pirámide emocional. ¿En que escalón te encuestas ahora? Respuesta: La mayoría de los jugadores seguía en el estado de seguridad o tranquilidad después del juego y seguidos de ellos algunos presentaban una emoción de autorrealización.

5.2. Niveles terminados

Capítulo 6

Conclusiones

Capítulo 7

Anexos

En este capítulo se encuentran todos los anexos que se mencionaron en los capítulos anteriores.

7.1. Interfaces

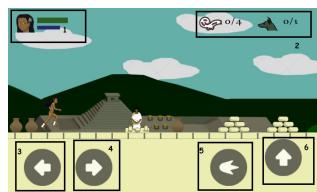


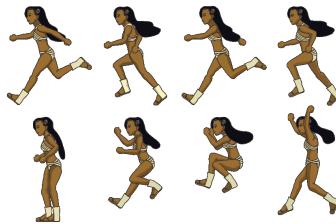
Figura 7.1: a nice plot

7.2. Diseño de Personajes

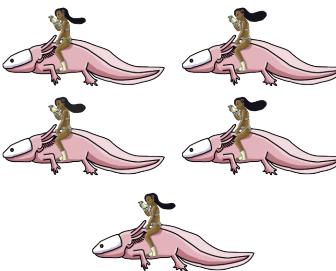
7.3. Modelo de Datos

7.4. Control de adicción en el jugador

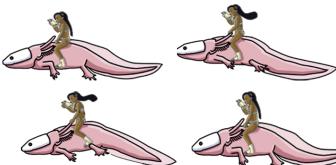
7.5. Maquetas de niveles



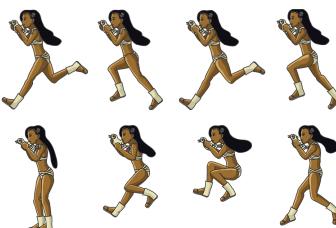
(a) Sprites Malinalli para el primer nivel.



(b) Sprites Malinalli de nado para el segundo nivel.



(c) Sprites Malinalli de salto para el segundo nivel.



(d) Sprites Malinalli para los niveles posteriores al segundo nivel.

Figura 7.2: Sprites del personaje jugable (Autoria propia)

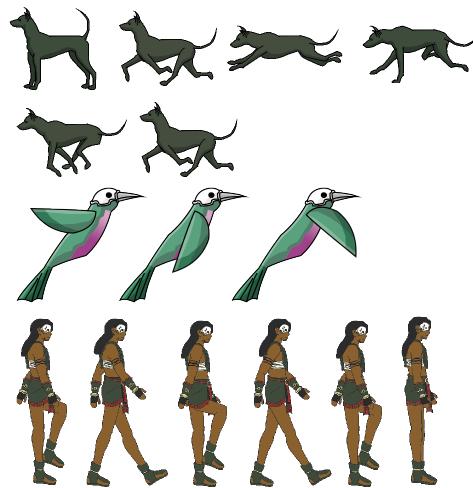


Figura 7.3: Sprites para las diferentes formas que toma Xólotl a lo largo del juego.



Figura 7.4: Sprites para los enemigos normales del juego.

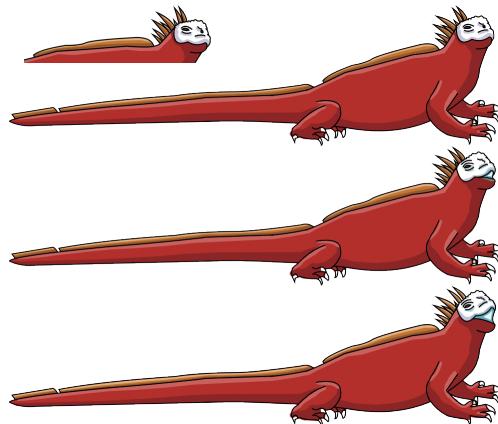


Figura 7.5: Sprites de Xochitonal.



Figura 7.6: Sprites de Itzpapálotl.

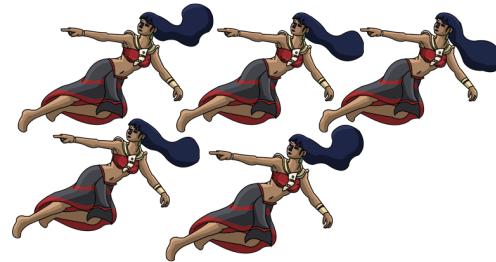


Figura 7.7: Sprites de Tlazolteotl.

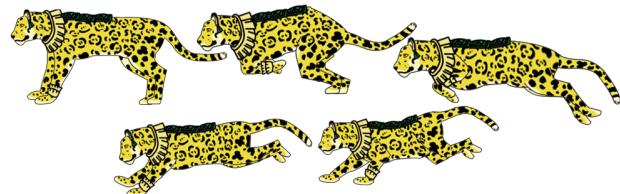


Figura 7.8: Sprites de Tepeyollotl.



Figura 7.9: Sprites de Mictlantecuhtli.

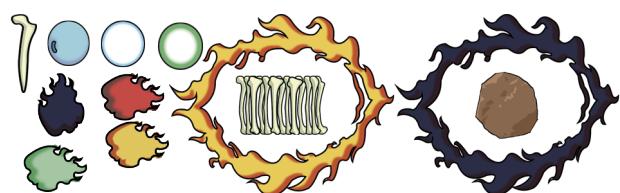


Figura 7.10: Sprites de los ataques de los personajes.