



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

²yolotl.escom@gmail.com

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

Advertencia	II
1. Introducción	1
2. Antecedentes	2
2.1. Propuesta	2
2.1.1. Planteamiento del problema	2
2.1.2. Marco Teórico	3
2.1.3. Planteamiento de la solución	7
2.2. Ajustes	7
2.2.1. Corrección del enfoque de la solución	7
2.2.2. Nueva división de trabajo	7
2.2.3. Actualizando el motor de juego	8
2.3. Contribuciones	9
2.3.1. Modelo de datos	9
2.3.2. Estrategias para combatir la adicción entre los usuarios	10
2.3.3. Modelo de Negocios	10
2.4. Trabajo realizado durante trabajo terminal 1	10
2.4.1. Etapa de Preproducción	11
2.4.2. Etapa de producción	12
3. Trabajo realizado	15
3.1. Etapa de producción de <i>Huddle</i>	15
3.1.1. Cuarto <i>sprint</i> de producción	15
3.1.2. Sexto <i>sprint</i> de producción	16
4. Resultados obtenidos	30
4.1. Prueba	30
4.1.1. Objetivo de la prueba	30
4.1.2. Herramientas utilizadas durante la prueba	30
4.1.3. Aplicación de la prueba	30
4.1.4. Conclusiones de la prueba	30
5. Conclusiones	31

6. Anexos	34
6.1. Interfaces	34
6.2. Diseño de Personajes	34
6.3. Modelo de Datos	34
6.4. Control de adicción en el jugador	34
6.5. Maquetas de niveles	34

Capítulo 1

Introducción

Capítulo 2

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.

2.1. 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.

2.1.1. 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 [1]. 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[2].

2.1.2. 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"[3].

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, advertising y advergaming*" se menciona que existen seis características comunes en los videojuegos: Interactividad, entretenimiento, jugabilidad, simulación \virtualidad, inmersión y multiplataformidad[4]; 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[5].
- **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"[6].
- **Jugabilidad:** en el libro "*Marketing y videojuegos: Product placement, in-game, advertising 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"[4].
- **Simulación \Virtualidad:** La simulación "se trata de una representación a medida cuyo objetivo nos permite interactuar y relacionarnos con lo representado según nuestros intereses"[4].
- **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 [7]. 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.[8]

Los videojuegos pueden ser clasificados con base a su jugabilidad, en el libro "*Juego. Historia, Teoría y Práctica del Diseño Conceptual de Videojuegos*"[9] 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[9]. 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 [10]. 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 redefinición en el proyecto y por ende un aumento en los costos de producción del sistema [11]. 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* [12]. Cada sprint se compone de las siguientes fases:
 - **Concepto.**
 - **Especulación.**
 - **Exploración.**
 - **Revisión.**
 - **Cierre**[13].
 - **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[14].
 - **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 [15]. *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 [16]; 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[17]”. 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 [17].

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 [18].

2.1.3. 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 videojuegos percibiendo un ingreso de 1.4 mil millones de dólares en esta industria. A su vez México cuenta con 49.2 millones de jugadores[19].
- **El auge de los juegos para teléfonos móviles:** En el 2017 la industria del videojuego tuvo ganancias de 108.9 mil millones de dólares 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[19].
- **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 [20].
- **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 [21].

2.2. Ajustes

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

2.2.1. Corrección del enfoque de la solución

2.2.2. Nueva división de trabajo

Antes del inicio del tercer *sprint* y teniendo como base la experiencia de desarrollo los anteriores prototipos, queda claro que se necesita diseñar una nueva estrategia que permita agilizar el desarrollo del juego sin comprometer la calidad del mismo. 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 2.1 .

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

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 2.1: Asignacion de tareas

2.2.3. Actualizando el motor de juego

Paralelamente a la nueva asignación de tareas, fue liberada 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 arrastran diferentes *Sprites* que se hayan importado previamente al tilemap (ver figura 2.2). 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 2.4). *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 *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

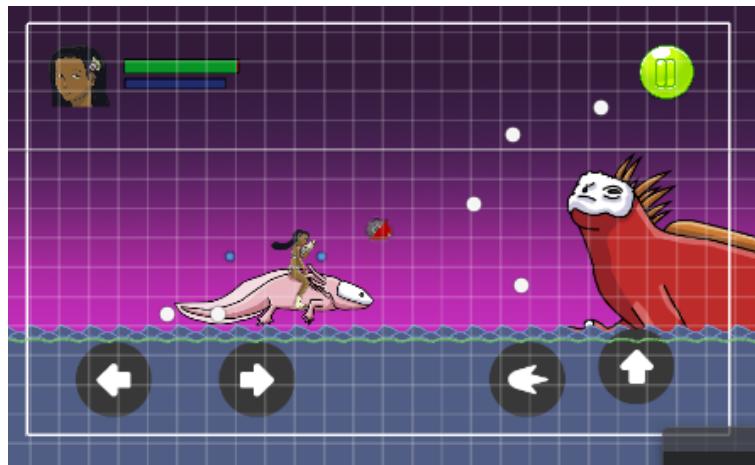


Figura 2.2: Vista de la escena cuando se tiene un *GameObject* de tipo *Tilemaps* para la construcción de niveles

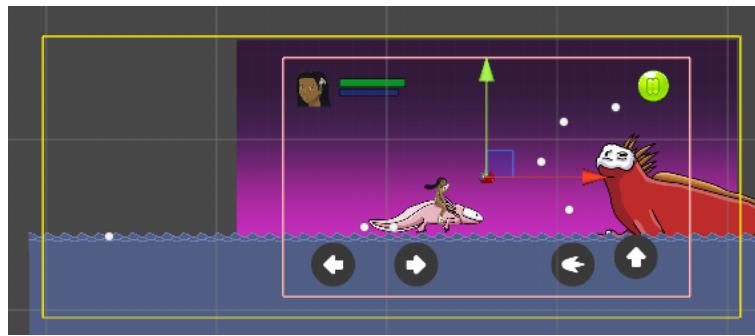


Figura 2.3: Vista de la escena cuando se tiene un *GameObject* de tipo *Tilemaps* para la construcción de niveles

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.

2.3. Contribuciones

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

2.3.1. 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 6.3) 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

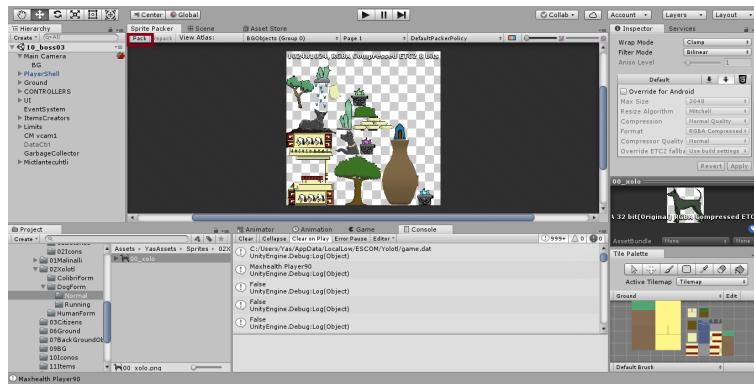


Figura 2.4: Vista de la pestaña del *Sprite Packer*.

entidad está definida por como su identificador y por otras entidades como `GameObjectPosition`, `Level`, `Tag`, `AnimationMachine`, entre otros.

2.3.2. 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 conllevaban 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 6.4):

- **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.

2.3.3. Modelo de Negocios

2.4. 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.

2.4.1. 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 6.1.

Tercer Sprint Huddle de Preproducción

En el tercer Sprint se definen la cantidad de niveles y en que 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 6.2 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 guion 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 ingeniería.

2.4.2. 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 3.

Primer Sprint Huddle de Producción.

En este *sprint* se realiza un análisis del documento de diseño, en consecuencia de este análisis 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 colecionables, 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 tanto 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 2.5 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:

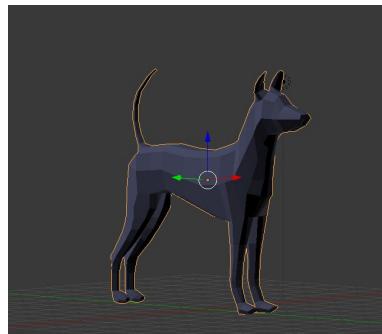
- Objetos de fondo: Arbustos, árboles, jarrones y cajas.
- Imagen de fondo: Fondo de la selva, la ciudad y el menú principal.
- Ciudadanos del mercado: Comerciantes, nobles y esclavos.
- *Xólotl* en su forma *xoloitzcuintle*: Bloques de animación para correr y normal.
- *Malinalli* sin la caracola: Bloques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

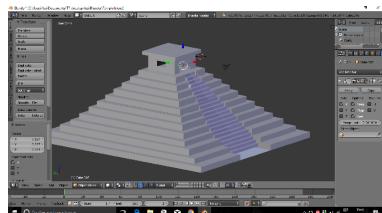
- Control de personaje por medio de la GUI.
- Transiciones entre interfaces.
- Personaje seguible que aparece en el primer nivel funcional.
- Funcionamiento básico del controlador de diálogos.



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

Capítulo 3

Trabajo realizado

3.1. Etapa de producción de *Huddle*

3.1.1. 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 3.1). 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 6.5 se pueden consultar las maquetas de los niveles pares.

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 3.2).

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 3.3) emplea para atacar y que se obtiene al final del primer nivel de la sección de selva, estos *sprites* para *Malinalli*

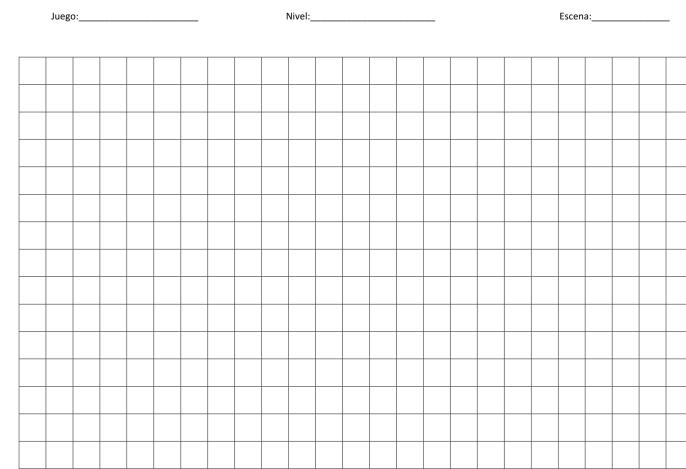


Figura 3.1: Plantilla para la creación de niveles.

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 3.4), 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.

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 ataque (ver figura 3.5). 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 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 6.2.

3.1.2. Sexto *sprint* de producción

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 *items*.

Es verdad que ya 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 sé-

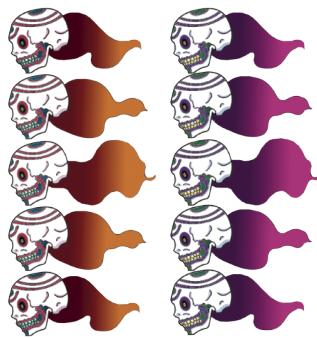


Figura 3.2: Bloques de animación para el enemigo de tipo fantasma.

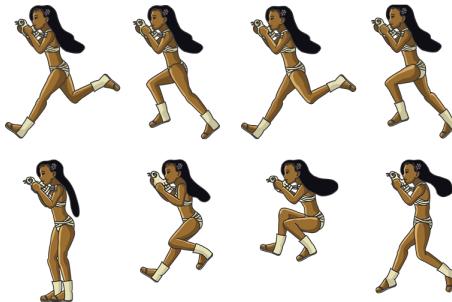


Figura 3.3: Bloques de animación para *Malinalli* posterior a que ella obtiene la caracola.

timo *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).
- **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 3.7). 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

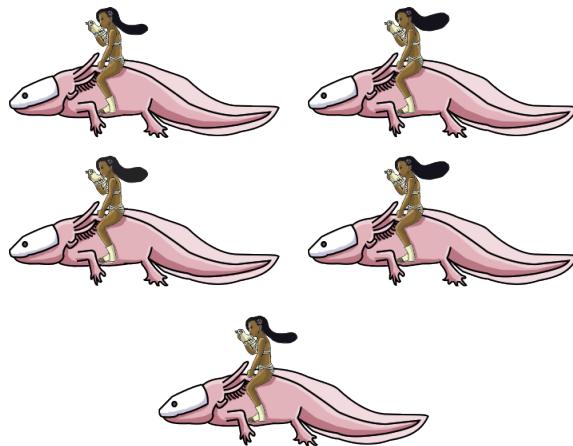


Figura 3.4: Bloques de animación para *Malinalli* montando al ajolote del segundo nivel del juego.



Figura 3.5: Bloques de animación para *Mictlantecuhtli*, jefe final del juego.

que la vida actual del enemigo nunca sea cero y nunca sobrepase a su máximo de vida (ver figura 3.8).

- **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 jugado 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 3.9).
- **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 3.10. 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:
 - 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 volteará.

Voltear un *Sprite* no representa mayor problema en código; sin embargo, el voltear un *sprite*

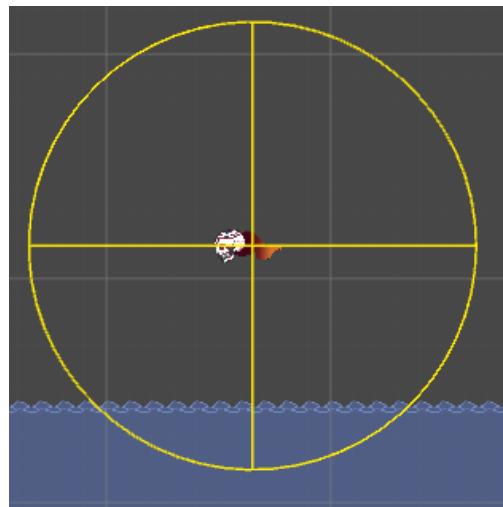


Figura 3.6: Ejemplo del area de acción del enemigo *RedGhost*

```
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 3.7: Nuevos atributos para la clase *Enemy*.

cuyo colisionador no es simétrico como el de la figura 3.11. 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 3.11.

- **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 3.12).
- **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

```

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

```

Figura 3.8: Actualización de la cantidad de vida de la clase *Enemy*.



Figura 3.9: 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.

ya que este explota al hacer contacto con el jugador. En la figura 3.14 se puede observar el patrón 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 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 3.16) 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 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

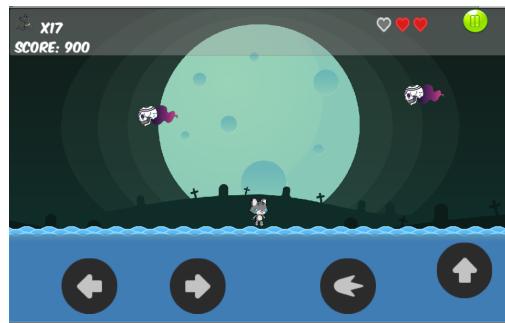


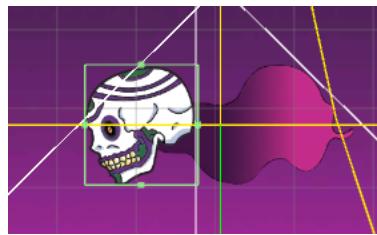
Figura 3.10: En los primeros prototipos el enemigo es incapaz de girar su sprite y su ataque una vez que el jugador se coloca tras de éste.

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:

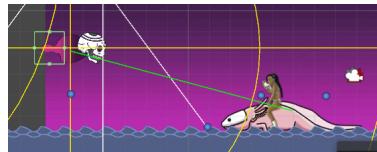
- ***WaitForAction***: Espera un tiempo determinado y asigna un nuevo índice valor verdadero del arreglo de valores booleanos. Se activa si *whatCanDo[0]* es verdadero.
- ***shotFire***: Dispara cuatro esferas de fuego que siguen al jugador y en caso de no chocar con este después de un tiempo se destruyen. Se activa si *whatCanDo[1]* es verdadero.
- ***useShell***: Invoca un circulo 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.

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 3.17 se muestra un ejemplo en código de la maquina de estados del jefe *Itzpapálotl*.

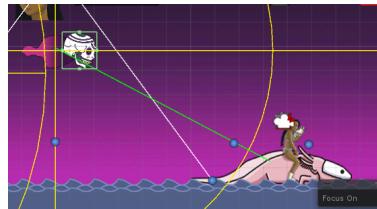
Por la forma en la que fue diseñado el comportamiento de la máquina de estados, el nivel de dificultad que presente el jefe esta dado en función de dos variables: *damageAmount* y *timeBetweenAttacks*, correspondientes a la cantidad de daño que el jefe puede infringir en el jugador y al tiempo que se 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.



(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 3.11: Comportamiento del colisionador antes y después de la implementación de la clase *FixerCollider*.

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 3.18. 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:
 - **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 se vea afectado por la gravedad. Este componente permite en el código agregarle 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

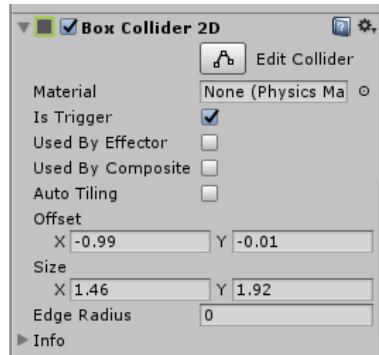


Figura 3.12: Configuración actual del colisionador de los enemigos.

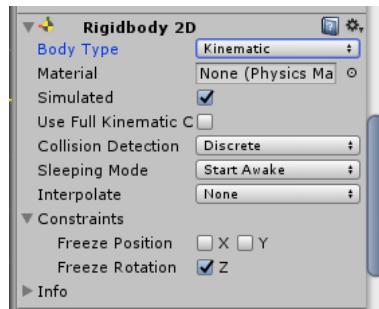


Figura 3.13: Configuración actual del componente *Rigidbody2D* de los enemigos.

del objeto.

Este tipo de disparo es empleado por los enemigos de tipo *RedGost*, *Tepeyóllotl* y por *Mictlantecuhtli*.

- **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 3.19 se puede ver la implementación de disco comportamiento 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 al 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 método 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

```

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



Figura 3.15: Patrón de movimiento del enemigo tipo Jaguar expresado en su comportamiento visual.

de un tiempo. Al igual que el anterior protege al enemigo de los ataques del jugador e infringe 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 vuelva 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*.



Figura 3.16: 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 3.17: Ejemplo de la máquina de estados del enemigo jefe.

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[9]. 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 3.20. 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 permi-

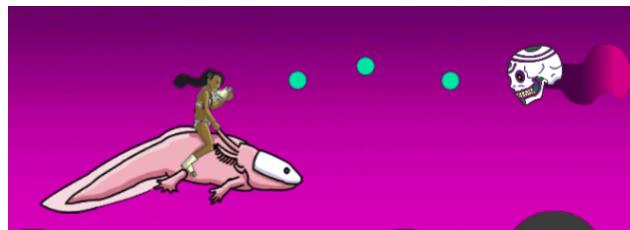


Figura 3.18: Ejemplo de un disparo de una trayectoria definida.

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

Figura 3.19: Implementación en código del comportamiento del disparo que sigue al jugador.

te 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 3.21 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 3.22.

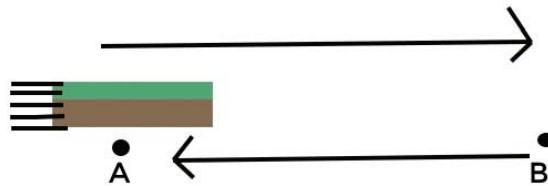


Figura 3.20: Comportamiento de la plataforma que se mueve de manera visual.

- **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.
- **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

```

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

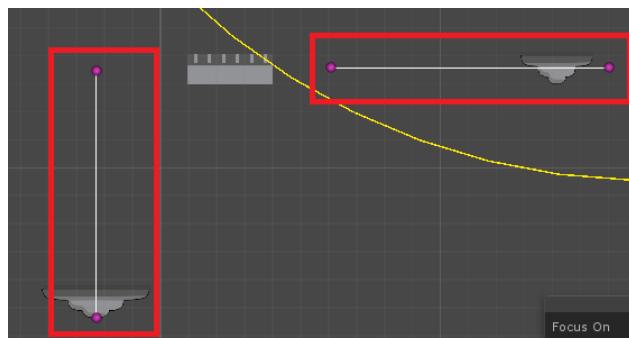


Figura 3.22: Uso del método *OnDrawGizmos* para visualizar la trayectoria de una plataforma que se mueve.

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 3.23 se puede ver la configuración de ambos objetos dentro de un nivel.

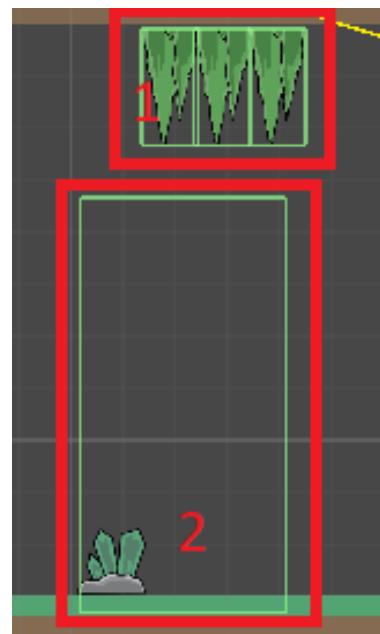


Figura 3.23: Configuración de los diferentes *GameObjects* que conforman el obstáculo estalagmita.

- **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.
- **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 .

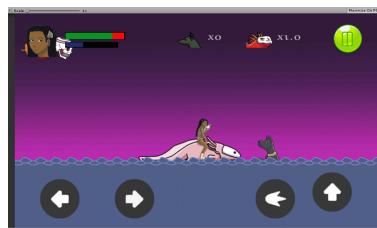
Implementando los ítems y los objetos colecciónables

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.

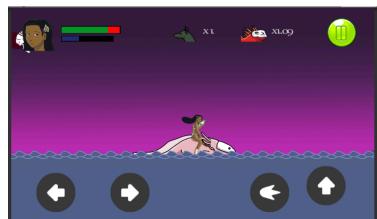
Dentro del juego existen dos tipos de Items: Los que recuperan cantidad de vida y los que recuperan cantidad de *tonalli*. Ambos tienen un funcionamiento similar: como atributos sus

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 3.24); mientras que en el nivel 4, los colecciónables 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). Para solucionar esto se crearon dos etiquetas para cada tipo de colecciónable: *CollectableDog* y *CollectableKey*. Dejando al gestor de colisiones de la clase *Player* verificar por medio de la etiqueta de qué tipo de colecciónable se trata e invoca al controlador de la interfaz gráfica de los marcadores del juego o HUB, ver figura .



(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 3.24: Interacción entre el jugador y los objetos colecciónables del nivel.

Implementación de los puntos de guardado

Capítulo 4

Resultados obtenidos

4.1. Prueba

- 4.1.1. Objetivo de la prueba**
- 4.1.2. Herramientas utilizadas durante la prueba**
- 4.1.3. Aplicación de la prueba**
- 4.1.4. Conclusiones de la prueba**

Capítulo 5

Conclusiones

Bibliografía

- [1] GrupoFormula, “Encuesta unam revela mexicanos saben poco de la constitución.” [Online]. Available: <http://www.radioformula.com.mx/notas.asp?Idn=660351&idFC=2017#>
- [2] Parametría, “Desconocen de qué país se independizó méxico.” [Online]. Available: http://www.parametria.com.mx/carta_parametrica.php?cp=4803
- [3] G. Carricay. ¿qué son los videojuegos? [Online]. Available: <https://medium.com/grupo-carricay/qu%C3%A9-son-los-videojuegos-d640dc6aa84>
- [4] J. M. Pereño, Marketing y videojuegos: Product placement, in-game, adevertising y advergaming. ESIC, 2010.
- [5] J. Steuer, “Defining virtual reality: Dimensions determining telepresence,” Journal of Communication, vol. 42, no. 4, pp. 73–93, 1994.
- [6] P. M. Fabrizio Lamberti, Andrea Sanna. Las tecnologías del entretenimiento: Pasado, presente y futuro. [Online]. Available: <https://www.computer.org/web/computingnow/archive/february2015-spanish>
- [7] S. Turkle, La vida en la pantalla: La construcción de la identidad en la era de internet. Paidos Iberica, 1997.
- [8] V. M. Navarro, “Libertad dirigida: Análisis formal del videojuego como sistema, su estructura y su avataridad.” Ph.D. dissertation, Universitat Rovira i Virgili, 2013.
- [9] B. L. Barinaga, Juego. Historia, Teoría y Práctica del Diseño Conceptual de Videojuegos. Alesia, 2010.
- [10] B. A. Rafael Menéndez. Metodologías de desarrollo de software. [Online]. Available: <http://www.um.es/docencia/barzana/IAGP/Iagp2.html>
- [11] C. B. Jurados, Diseño Ágil con TDD. España: SafeCreative, 2010.
- [12] J. S. Ken Schwaber. La guía definitiva de scrum: Las reglas del juego. [Online]. Available: <http://www.scrumguides.org/docs/scrumguide/v1/scrum-guide-es.pdf>
- [13] J. P. Alexander Menzinsky, Gertrudis López. Scrum manager. [Online]. Available: http://www.scrummanager.net/files/sm_proyecto.pdf
- [14] M. P. Esteso. Programación extrea: Qué es y principios básicos. [Online]. Available: <https://geekytheory.com/programacion-extrema-que-es-y-principios-basicos>

- [15] C. E. N. L. Gerardo Abraham Morales Urrutia. Proceso de desarrollo para videojuegos. [Online]. Available: erevistas.uacj.mx/ojs/index.php/culcyt/article/download/299/283
- [16] J. Ward. What is a game engine? [Online]. Available: https://www.gamecareerguide.com/features/529/what_is_a_game_.php
- [17] A. H. Núñez, “Además de presupuesto, ¿qué le falta a la cultura en México?” [Online]. Available: <https://cuadrvio.net/ademas-de-presupuesto-que-le-falta-a-la-cultura-en-mexico/>
- [18] E. mundo de Tehuacan. Tipos de cultura y cultura híbrida. [Online]. Available: <http://www.elmundodetehuacan.com/index.php/opinion/opinion-conten-init/28997-Tipos-de-cultura-y-cultura-h%C3%ADbrida>
- [19] Newzoo. 2017 global games market report. [Online]. Available: <https://newzoo.com/insights/trend-reports/newzoo-global-games-market-report-2017-light-version/>
- [20] ——. 2017 global mobile market report. [Online]. Available: <https://newzoo.com/insights/trend-reports/global-mobile-market-report-light-2017/>
- [21] D. Hefner, C. Klimmt, and P. Vorderer, “Identification with the player character as determinant of video game enjoyment,” in *Entertainment Computing – ICEC 2007*, L. Ma, M. Rauterberg, and R. Nakatsu, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 39–48.

Capítulo 6

Anexos

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

6.1. Interfaces



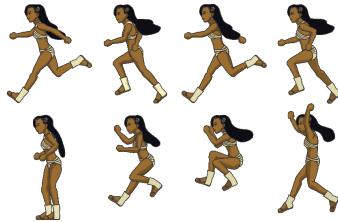
Figura 6.1: a nice plot

6.2. Diseño de Personajes

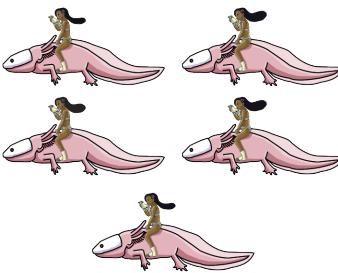
6.3. Modelo de Datos

6.4. Control de adicción en el jugador

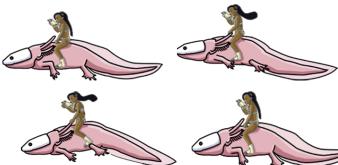
6.5. Maquetas de niveles



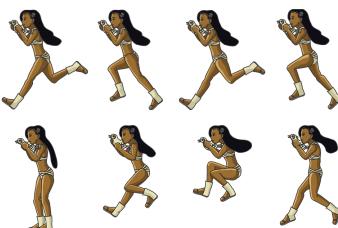
(a) Sprites Malinalli para el primer nivel.



(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 6.2: Sprites del personaje jugable (Autoria propia)

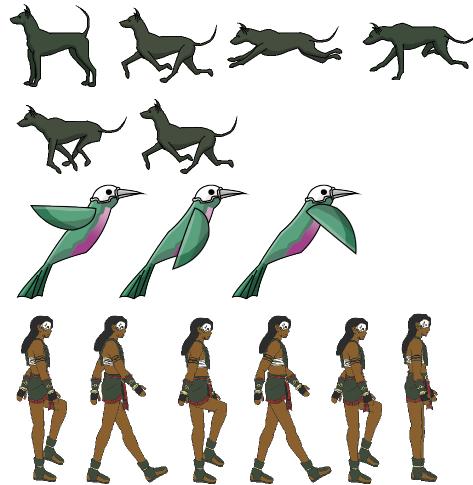


Figura 6.3: Sprites para las diferentes formas que toma Xólotl a lo largo del juego.



Figura 6.4: Sprites para los enemigos normales del juego.

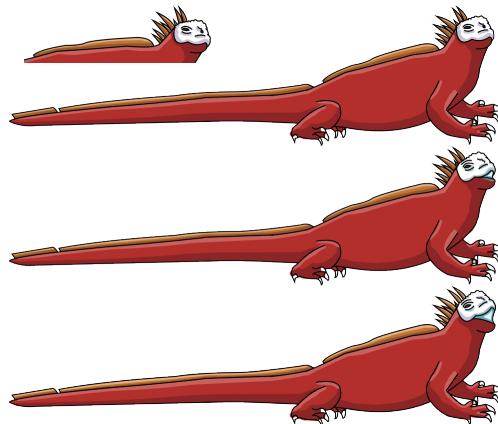


Figura 6.5: Sprites de Xochitonal.



Figura 6.6: Sprites de Itzpapálotl.



Figura 6.7: Sprites de Tlazolteotl.

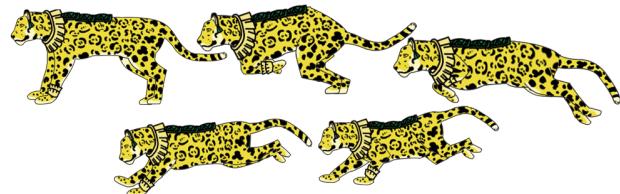


Figura 6.8: Sprites de Tepeyollotl.



Figura 6.9: Sprites de Mictlantecuhtli.

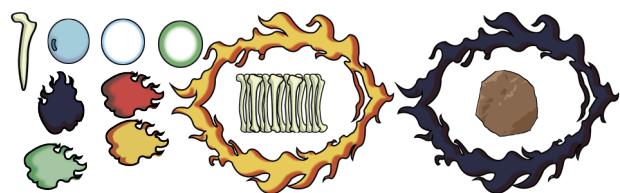


Figura 6.10: Sprites de los ataques de los personajes.