Proyecto de Grado

Presentado ante la ilustre Universidad de Los Andes como requisito final para obtener el Título de Ingeniero de Sistemas

Creación de un método de desarrollo de software orientado a videojuegos. Fase: Implementación.

Por

Br. Miguel Gerardo Martinez Pernia

Tutor: Prof. Alejandro Mujica

Junio 2018

Creación de un método de desarrollo de software orientado a videojuegos. Fase: Implementación.

Br. Miguel Gerardo Martinez Pernia

Proyecto de Grado — Sistemas Computacionales, 55 páginas

Resumen: El proceso que conlleva la creación de un videojuego se divide principalmente en dos enfoques de trabajo: Diseño e Implementación. A través de las herramientas gratuitas que ofrece la web, los desarrolladores independientes de videojuegos han logrado emerger en el mercado, sin embargo se conoce muy poco de los métodos que estas personas usan para desarrollar sus aplicaciones, por lo que se busca diseñar un método de desarrollo que resulte útil para las personas que quieran adentrarse en el campo del desarrollo de videojuegos, enfocándose además, en innovar los métodos de enseñanzas que ofrece la carrera de Ingeniería de Sistemas en la Universidad de los Andes y potencialmente la creación de una nueva carrera que incluya todas las disciplinas necesarias para la creación de un videojuego completo. Este proyecto de grado se enfoca en la creación de un método de desarrollo orientado a videojuegos en su fase de implementación y su aplicación en el desarrollo de un videojuego para demostrar su eficiencia.

Palabras clave: Método, videojuego, diseño, desarrollo, implementación.

Índice

Índice de Tablas

Índice de Figuras

Capítulo 1

Introducción

La mayoría de los estudiantes que comienzan en la carrera de Ingeniería de Sistemas tienen como aspiración trabajar en aplicaciones de las nuevas tecnologías, tales como: desarrollo de videojuegos, realidad aumentada, realidad virtual, inteligencia artificial, machine learning, programación en la nube, big data, etc. Sin embargo, a medida que van cursando las materias del pensum establecido, notan que el conocimiento adquirido no se relaciona íntimamente con las aspiraciones que ellos pudieran tener en un principio.

En el caso especifico de desarrollo de videojuegos, no es sino hasta el ciclo profesional que los estudiantes tienen una idea generalizada de como emprender proyectos enfocados en este tipo de tecnologías. En otro ámbito, los estudiantes aspirantes a crear videojuegos creen que solo se trata de programar cuando es erróneo este pensamiento. El desarrollo de videojuegos es un área que abarca muchas disciplinas que no necesariamente se relacionan entre sí, tales como Publicidad y Mercadeo, Economía, Diseño Gráfico, Ingeniería de Software, Artes, Fotografía, entre otras. Como el desarrollo de videojuegos no es un área que pueda ser trabajada por una persona (aunque siempre existan casos aislados), este trabajo tiene como finalidad presentar un método para el desarrollo de este tipo de proyectos, por lo que, desde el enfoque de Ingeniería de Sistemas, se necesitara trabajar en dos fases principales: el Diseño y la Implementación de videojuegos. En particular, este proyecto se enfoca en la fase

de implementación de un método de desarrollo, siendo la fase de diseño descrita en el proyecto **Creación de un método de desarrollo de software orientado a videojuegos. Fase: Diseño** por Centeno (2018).

1.1 Motivación

La motivación de este proyecto de grado es ofrecer a los estudiantes, que aspiran crear videojuegos, una guía de trabajo y desarrollo, teniendo en cuenta los inconvenientes que surgen al estudiar la carrera de Ingeniería de Sistemas. Adicionalmente, fomentar el estudio y sobretodo el desarrollo de este tipo de tecnologías, tanto en los estudiantes como en los profesores universitarios para promover en el futuro la creación de nuevas materias especializadas, así como la generación de una nueva escuela orientada a formar profesionales capaces de nutrir el campo industrial de los videojuegos.

1.2 Planteamiento del problema

En la industria de los videojuegos existen tres roles principales que deben ocuparse para desarrollar videojuegos: Diseñador, Desarrollador y Artista. La mayoría de los estudiantes que ingresan a la carrera de Ingeniería de Sistemas, opción Sistemas Computacionales, aspiran a emprender sus propios proyectos de videojuegos. Sin embargo, a pesar de que a lo largo de la carrera se les enseña a manejar las herramientas básicas que les puede permitir trabajar en el campo de videojuegos, específicamente en el rol de Desarrollador, no existe un espacio enfocado específicamente en como utilizar dichas herramientas, para poder crear este tipo de productos.

1.3 Objetivos

1.3.1 Objetivos generales

Crear un método de desarrollo de software, orientado a videojuegos, haciendo énfasis en la fase de implementación de un producto de software.

1.3.2 Objetivos específicos

- Buscar y evaluar recursos de terceros que sean útiles en la implementación del videojuego según el diseño establecido en el proyecto Creación de un método de desarrollo de software orientado a videojuegos. Fase: Diseño.
- Adaptar o innovar métodos de desarrollo de software para que sean usados por un estudiante de ingeniería de sistemas, con el propósito que sus aplicaciones consuman la menor cantidad de recursos posibles.
- Implementar el método creado a través del desarrollo de un videojuego.

1.4 Alcance

Desarrollar e implementar un método de desarrollo de videojuegos que permita, a cualquier persona interesada en el campo industrial de los videojuegos, emprender sus propios proyectos.

1.5 Antecedentes

Ludum Dare, es una comunidad *online* que organiza tres veces al año una competencia de desarrollo de videojuegos. En la semana previa a la competencia la comunidad elige la temática que los participantes deben seguir. Seguidamente los participantes tienen 48 horas para diseñar y desarrollar sus ideas, y al final, tanto los participantes como la comunidad, tienen una semana para valorar todos los videojuegos creados. La competencia no tiene un premio en sí, de hecho Mike Kasprzak comenta en el articulo de Porter (2015) *"El premio es tu producto. Ser capaz de decir 'Yo hice esto' es asombroso"*. Varios desarrolladores independientes que participaron en estos eventos han logrado tener éxito en la industria, tal es el caso de Titan Souls, un videojuego de acción-aventura que actualmente se encuentra publicado en la plataforma de Steam.

Riot Games fundada en 2006, es la empresa que desarrolló League of Legends (lanzado en 2009) con más de 67 millones de jugadores activos cada mes. Parte del

éxito de Riot Games se debe a que siempre están abiertos a aceptar nuevos empleados, ya sean estudiantes universitarios o personas sin estudios. Entre los requerimientos que deben cumplir los aspirantes están el ser habilidoso con un lenguaje de programación, hasta ser una persona con muy buenas ideas. Existen varios casos de estudiantes universitarios que han tenido la oportunidad de trabajar con Riot Games entre los que se puede mencionar a Li (s.f.), pasante como Ingeniero de Software, quien relata que su experiencia le permitió aprender mucho sobre su campo en poco tiempo, gracias al trabajo práctico en contraste al trabajo monótono que le ofrecía su universidad.

Unity Technologies es la empresa desarrolladora responsable de crear Unity, que fue lanzado al público en 2005. En sus primeros años, Unity tuvo un alcance que ningún motor de videojuegos potente ofrecía, ya que el bajo costo de la licencia de una versión limitada, permitía a pequeñas empresas y a los desarrolladores independientes ahorrar tiempo. Es decir, ya no tenían que desarrollar sus propios motores gráficos desde cero y a pesar de ser una versión limitada, les permitía incluso publicar sus videojuegos. Fue a partir de la versión 4 de Unity, en la que se concretaron acuerdos con Sony, Microsoft y Nintendo, que hubo un auge de desarrolladores independientes, aquí fue donde el termino *Indie Developer* se reconoció en toda la industria. Actualmente, se dispone de una licencia gratuita de Unity, dando de baja la licencia *Indie* que tanto llamó la atención en sus inicios. En virtud de esto, se puede decir que Unity ya no es sólo un motor de videojuegos, sino que además es una comunidad en sí, en donde todos sus usuarios pueden encontrar material de apoyo, sea en el ámbito de desarrollo, de diseño o artístico. En los foros se puede encontrar testimonios del éxito que han tenido pequeños grupos de personas, como fue el usuario Gigiwoo (2011) que gracias a Unity ha podido desarrollar juegos exitosos en las plataformas de juegos móviles, que le han tomado desde 2 a 16 semanas en completar su desarrollo.

Con referencia a lo anterior, se demuestra la existencia de casos en que pequeños grupos de personas lanzan en las plataformas de distribución digitales, videojuegos que surgieron tanto en comunidades *online* como en pequeñas empresas (primeros años de Riot Games) y gracias a la participación de sus respectivos consumidores, en todas las

etapas de desarrollo, estos videojuegos consiguieron ser reconocidos por la comunidad de jugadores en distintas partes del mundo. Añadiendo además, la importancia que tiene el trabajo práctico en este campo laboral a pesar de contar con poco tiempo disponible para completas estos proyectos. Otro caso es el del Ingeniero de Sistema Henríquez (2016), quien presentó como proyecto de grado Creación de una StartUp con un videojuego aplicando el método Lean StartUp. Este proyecto de grado demuestra que la participación de los usuarios potenciales desde los inicios de la creación del videojuego hasta su finalización, y utilizando métodos emprendedores de desarrollo, pueden resultar en un producto muy atractivo para sus usuarios potenciales.

1.6 Estructura del documento

Capítulo 1. Introducción: Presenta el enfoque del documento, planteamiento del problema y objetivos.

Capítulo 2. Estado del arte: Presenta conceptos que contextualiza el desarrollo del documento.

Capítulo 3. Método de desarrollo DIY: Se describe los procesos enfocados para la implementación de un videojuego.

Capítulo 4. El Producto: Muestra la aplicación del método de desarrollo descrito en el capítulo 3.

Capítulo 5. Conclusiones: Resume los conceptos mas importantes del método de desarrollo DIY, objetivos alcanzados en la aplicación del método y recomendaciones.

Capítulo 2

Estado del arte

2.1 Modelos y métodos de desarrollo de software

La historia le ha permitido a la sociedad cubrir las necesidades básicas sin mucho esfuerzo, cada día han ido emergiendo productos y servicios con el propósito de satisfacer y entretener hasta llegar al mundo contemporáneo. Ya no solo emergen productos y servicios para el entretenimiento de la sociedad, sino también para cubrir nuevas necesidades que aparecen en cualquier ámbito, y con una población tan grande, el tiempo y los recursos son factores que determinan la calidad y el alcance de esos productos y servicios. Tomando en cuenta estos factores, en el ámbito de las tecnologías digitales, han surgido distintos modelos y métodos de desarrollo de software. Las primeras propuestas fueron publicadas a partir de los años 70, tales como el método Programación Estructurada de Jackson, documentado en el libro Principios de Diseño del Programa por Jackson (1975), y el modelo Scrum definido en 1986 por Takeuchi y Nonaka (1986).

Los modelos de desarrollo de software definen a groso modo, como un software debería desarrollarse. No dictan reglas, tan solo ofrecen una estructura en las que el flujo de trabajo puede basarse. Estos modelos son:

- Modelo cascada: Las etapas del desarrollo basados en este modelos siguen un patrón secuencial estricto, quiere decir que no se puede avanzar a la siguiente

etapa de desarrollo hasta que la etapa actual no este completada.

- Modelo iterativo: Propone que el desarrollo del software sea por partes, haciendo que el producto crezca lentamente permitiendo la detección de posibles errores de diseño.

- Modelo ágil: Se basa en el modelo iterativo con la diferencia en que no le toma mucha importancia a la planificación y el diseño. Esto genera productos en poco tiempo pero con gran probabilidad de fallos en el diseño final del producto.

- Modelo espiral: Este modelo combina dos enfoques. Se compone de elementos de los de cascada e iterativos. Su principal característica es que hace mucha énfasis en la planificación y los riesgos.

- Modelo orientado a la reutilización: Básicamente estudia los requerimientos para la implementación y así reutilizar la mayor cantidad de código posible.

En general los procesos que conforman a los métodos de desarrollo de software se estructuran con base a los modelos ya introducidos. En el ámbito académico de un estudiante de Ingeniería de Sistemas, pareciera que los métodos que mejor podrían adaptarse a estas circunstancias son:

Programación Extrema (XP)

Este método de desarrollo de software definido por Beck (1996), se enfoca en la productividad y los valores de los actores que integran el grupo de desarrollo. Trata de minimizar el tiempo invertido en el proyecto y a su vez entregar un producto de calidad. Los actores, en especial los responsables de implementar código, adoptan estrategias de trabajo diferentes a las habituales que respetan la esencia del método, por ejemplo, el código puede ser trabajado por dos personas, una de ellas programa mientras la otra esta evaluando el código y luego siguiendo algún criterio, se cambian los roles; la ventaja de esta estrategia de programar es que el legibilidad del código final es muy clara. El método XP se estructura de la siguiente manera, véase Figura 2.1.

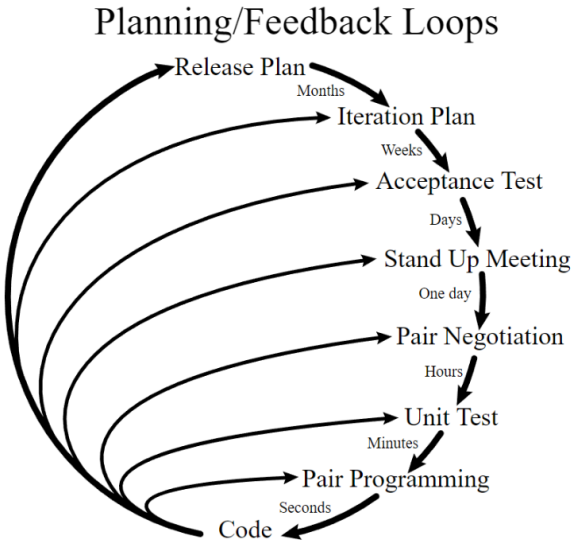

Figura 2.1: Estructura del método de desarrollo XP

Desarrollo rápido de aplicaciones (RAD)

Este método surgió como una alternativa a los métodos de desarrollo basados en el modelo cascada, ya que esos solían tomar mucho tiempo en desarrollarse. Fue entonces que Martin (1993) desarrolló el método RAD; consiste en la creación de prototipos sin tomar mucho en consideración el diseño del software. Este método se estructura tal y como se muestra en la Figura 2.2.

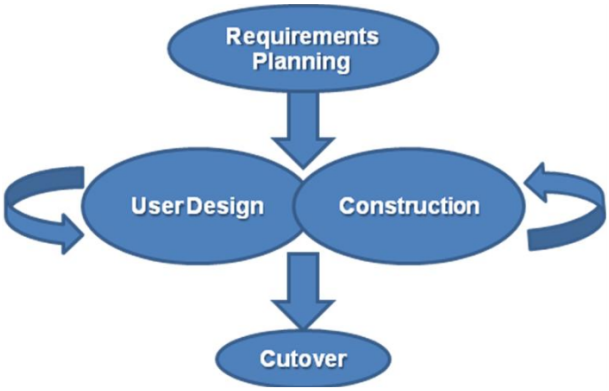

Figura 2.2: Estructura del método de desarrollo RAD

2.2 Motores de videojuegos

Un pilar fundamental en el desarrollo de un videojuego es el Motor de Videojuego o *Game Engine*, este se encarga de integrar todos los componentes que conforman a un videojuego: gráficos, sonido, IA, mecánicas de la jugabilidad, vídeos, niveles, arte, etc. Anteriormente, si un grupo de desarrolladores independientes quería desarrollar un videojuego, tenía que programar desde cero el motor de videojuego, aumentando considerablemente los costos de producción en tiempo y dinero; fue así hasta las apariciones de varios Motores de Videojuegos desarrollados por otras compañías como Unreal, GameMaker: Studio, CryEngine y Unity. El problema radica en que en el auge de varios de estos Motores de Videojuegos, las licencias de uso tenían un costo monetario muy elevado, por lo que los desarrolladores independientes no podían permitirse ese tipo de herramientas. La situación cambió cuando Unity planteó su modelo de negocios como se menciona en la sección 1.5.

2.3 Paradigmas de programación enfocados a videojuegos

Los lenguajes de programación en sí mismos fueron creados para ser usados bajo uno o varios paradigmas de programación. Esto no debe cumplirse estrictamente; si la implementación del lenguaje lo permite y el programador lo requiere, el diseño del código puede basarse en varios paradigmas de programación. A continuación se enlistan algunos de los paradigmas de programación mas comunes:

- Imperativo: El mas común de todos los paradigmas, consiste en dar instrucciones en orden. Los lenguajes que usan este paradigma son C, BASIC y Pascal.
- Orientado a objeto: Se basa en el paradigma imperativo; también agrega las clases, estas representan objetos o conceptos que son modelados mediante atributos y métodos que operan sobre los atributos. Algunos de los lenguajes que usan este paradigma son C++, C#, Java y Python.

- Funcional: Se basa en la declaración de funciones matemáticas, recibe un entrada en forma de parámetros, una función transforma la entrada y finalmente se genera una salida, osea que retorna un valor. La implementación en Python permite el uso de este paradigma.

¿Pero, qué pasa con los paradigmas que no pueden ser usados explícitamente por un lenguaje de programación? Como ya se menciona, si la implementación del lenguaje lo permite entonces puede usarse el paradigma. Este es el caso de Unity, que a pesar de que C# se basa en el paradigma orientado a objeto, el scripting (o la forma en que Unity recomienda programar) se basa en el paradigma Entidad - Componente.

Paradigma Entidad-Componente (EC)

Como lo describe Nystrom (2014), la intención de este paradigma es que una entidad pueda abarcar varios dominios evitando que los dominios se relacionen entre si; refiriéndose a los dominios como los distintos comportamientos o funcionalidad que una entidad puede tener.

Para entender mejor la descripción de este paradigma, se presenta el siguiente diseño basado en programación orientada a objeto o POO: definida una clase *GameObject* que actúa como base para cualquier entidad que puede existir dentro del videojuego, entonces, derivando la clase base se definen las clases Unidad terrestre y Unidad acuática donde cada una implementa el funcionamiento del motor por separado, seguidamente se define una clase Unidad anfibia, la cual hereda la funcionalidad del motor tanto de la clase Unidad terrestre como de la clase Unidad acuática. Véase Figura 2.3.

El problema de este diseño es que se forma un patrón de diamante, causando ambigüedad cuando la clase Unidad anfibia accede a algún método de la clase *GameObject* y adicionalmente C# no permite la herencia múltiple, justamente para evitar ese tipo de ambigüedad semántica. Otra problema de este diseño es que se programa código redundante, funcionalmente hablando, los métodos de motor en las clases de Unidad terrestre y Unidad acuática hacen lo mismo. Ahora cambiando

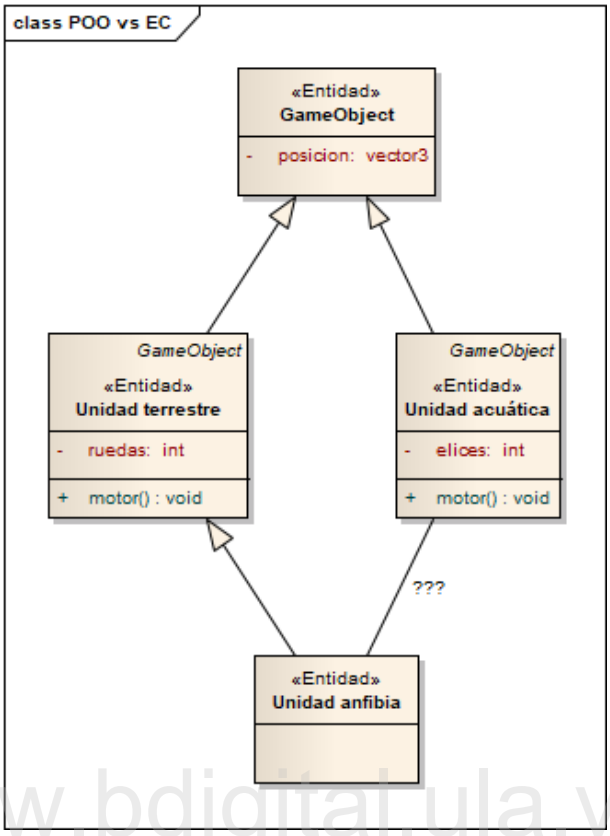

Figura 2.3: Diseño de clases utilizando el paradigma de POO

el paradigma de POO a EC, el diseño propuesto queda como se muestra en la Figura 2.4.

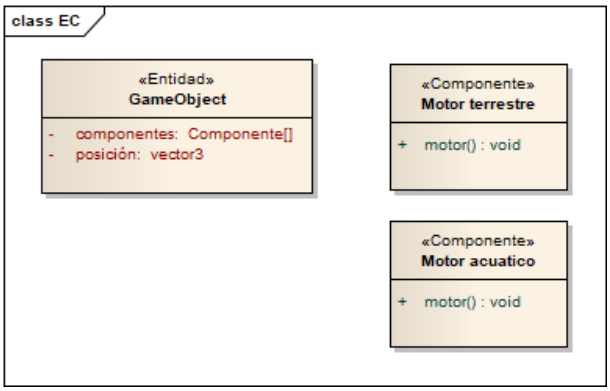

Figura 2.4: Diseño de clases utilizando el paradigma de EC

Como resultado, el diseño EC define 3 clases en vez de 4 como el diseño basado en

POO; la clase *GameObject* es la única que tiene permitido existir (o instanciarse) dentro del videojuego y a su vez esta entidad puede tener cualquier cantidad de componentes. Para tener una Unidad que pueda moverse en tierra, se le acopla a un *GameObject* el componente de Motor terrestre; para una unidad que pueda moverse en agua, se le acopla a un *GameObject* el componente Motor acuático; y para una unidad que pueda moverse en tierra y agua (osea, una unidad anfibia), se le acopla a un *GameObject* los componentes Motor terrestre y Motor acuático. Este paradigma permite el desacoplo entre funcionalidades o dominios distintos para una entidad y para asegurar que esta propiedad permanezca a lo largo de la implementación de un videojuego, es importante que cada componente se encargue unicamente de una funcionalidad y que no dependa estrictamente de otras funcionalidades.

Capítulo 3

Método de desarrollo DIY

El método de desarrollo de software orientado a videojuegos DIY, se divide en dos fases, diseño e implementación. La fase de diseño abarca todo el flujo de trabajo necesario para el desarrollo de un videojuego cumpliendo con los requerimientos fijados por Centeno (2018). El proceso de producción de esta fase indica desde un enfoque general, la forma en que debe implementarse el producto. Sin embargo, es necesario definir un método de trabajo que se aplique específicamente en la implementación, ya que a diferencia del desarrollo de un software común, los requerimientos para un videojuego pueden variar drásticamente a medida que se avanza en la implementación.

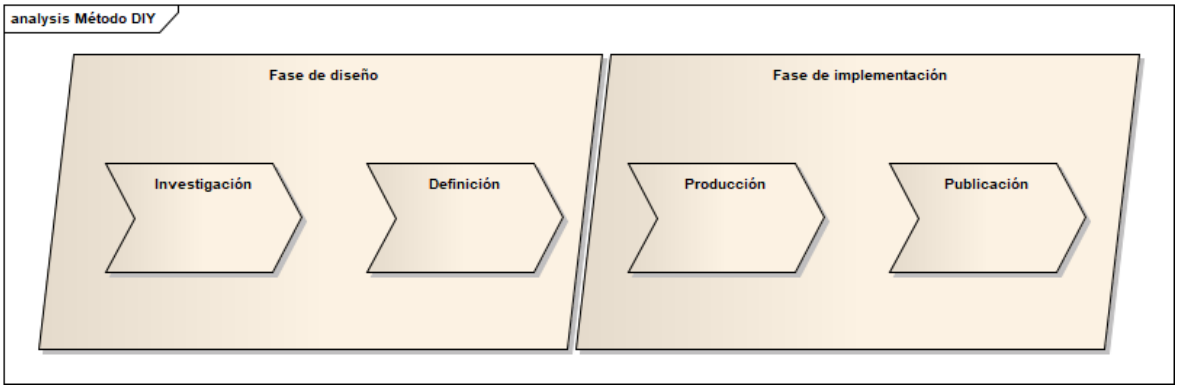

Figura 3.1: Diagrama general del método DIY

3.1 Preámbulo

Como se muestra en la Figura 3.1, el método DIY tiene dos fases de desarrollo, diseño e implementación; ademas se compone de cuatro procesos principales:

- Investigación: Encargado de buscar y estudiar estadísticas referentes a la creación del producto deseado, otorgando un preámbulo de datos actualizados y relevantes para la definición del videojuego.
- Definición: En el cual se define formalmente cada aspecto del producto, generando así la documentación de diseño del videojuego.
- Producción: Describe las herramientas a utilizar, el paradigma de trabajo, la arquitectura del producto, las fases de prueba y las correcciones pertinentes.
- Publicación: Actividad final del desarrollo, ubica el producto en el mercado, estudia su desempeño, genera estadísticas y plantea correcciones.

Los procesos que la fase de diseño agrupa, están pensados para que sean ejecutados una sola vez en el desarrollo del videojuego. La fase de implementación propone la siguiente ejecución, que por el contrario, no sigue una estructura lineal, véase Figura 3.2, por lo tanto se quiere profundizar el diseño en los procesos complejos, el proceso de producción específicamente.

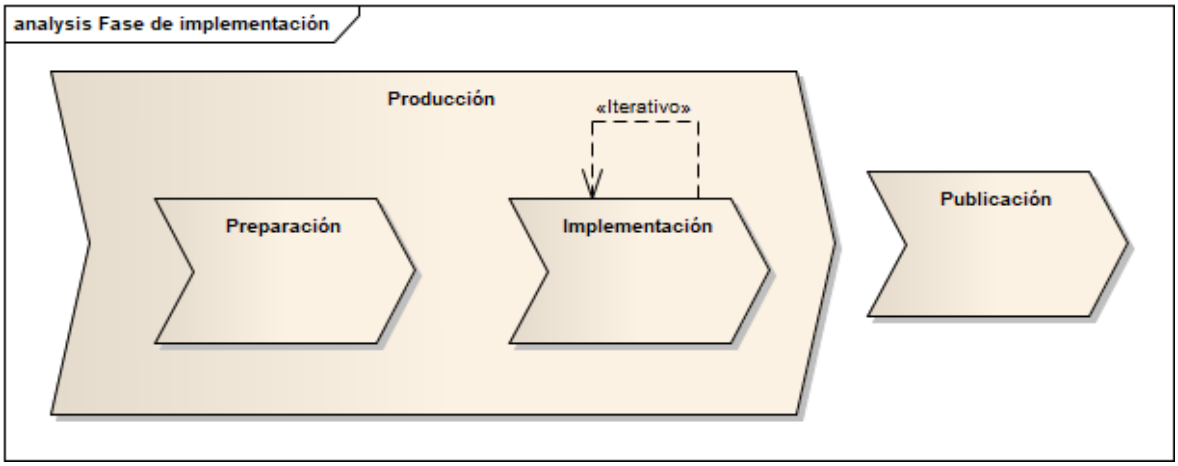

Figura 3.2: Diagrama general de la fase de implementación

Como lo describe Centeno (2018), el flujo de trabajo de este proceso se divide en dos etapas, la primera describe un proceso lineal, en la que se definen las herramientas de trabajo y demás aspectos previos a implementación per se. Posteriormente, se describe un flujo de trabajo cíclico, iterativo, evolutivo y modular. Quedando así el proceso de implementación que conforma el proceso de producción, véase Figura 3.3.

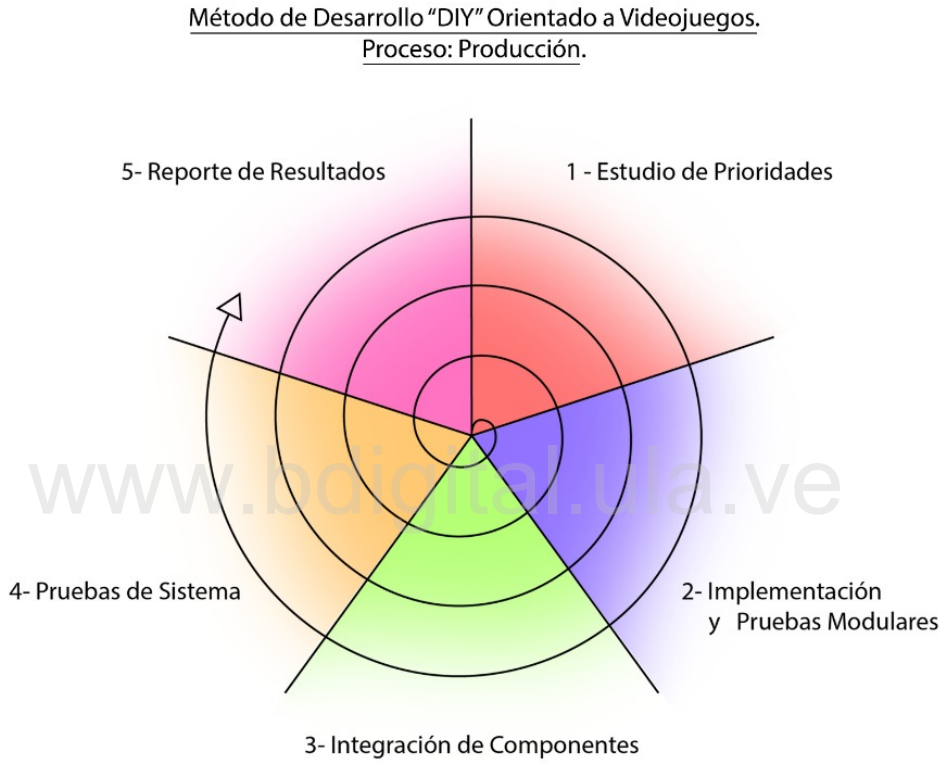

Figura 3.3: Diagrama general del proceso de implementación

3.2 Fase: Implementación

Considerando los diagramas en Figura 3.2 y Figura 3.3 se prosigue a caracterizar los procesos que conforman la fase de implementación del método de desarrollo DIY, enfocándose ademas, en las filosofías de otros métodos de desarrollo de software, principalmente en los métodos XP de Beck (1996) y RAD de Martin (1993).

3.2.1 Proceso: Preparación

Descripción

Define las herramientas de trabajo y demás aspectos previos a la implementación considerando: posibles costos de producción, tiempo a invertir, complejidad del desarrollo y limitaciones.

Objetivos

- Conocer las herramientas a usar a lo largo de la implementación.
- Aprender a usar las herramientas de ser necesario.
- Delimitar el diseño general del videojuego para minimizar el tiempo a invertir.
- Iniciar el proceso de producción.

3.2.2 Proceso: Estudio de prioridades

Descripción

Basado en el documento de diseño del videojuego y en las preparaciones, se procede a estudiar los elementos de mayor prioridad - de acuerdo a su impacto en el producto y a su aporte al funcionamiento de componentes ya realizados en iteraciones anteriores - y definir el siguiente elemento a implementarse.

Objetivos

- Tener clara las limitaciones que se puedan presentar antes de empezar el desarrollo del elemento en cuestión.
- Elegir que recursos se necesitan para el desarrollo del elemento a implementar.
- Integrar y entrenar con nuevas herramientas en caso de ser requerido.

3.2.3 Proceso: Implementación y pruebas

Descripción

El mas importante de los procesos desde un enfoque laboral, ya que según el estudio y la preparación realizada en el proceso anterior, implementa el elemento seleccionado en el menor tiempo posible al seguir los ideales del método XP que consisten en, reducir el costo en los requerimientos de un software (un elemento del videojuego en este caso) al tener ciclos pequeños de desarrollo Beck (1999). Esto asegura que los elementos a desarrollar en este proceso, sean modulares para facilitar las pruebas unitarias y posteriormente su integración en el sistema.

Objetivos

- Implementar el elemento actual en el menor tiempo posible.
- Asegurar que el elemento terminado sea lo suficientemente modular para practicar pruebas aisladas.

3.2.4 Proceso: Integración de componentes

Descripción

Unificación del componente realizado con los componentes previos relacionados.

Objetivos

- Formar sistemas complejos.

3.2.5 Proceso: Pruebas en el sistema

Descripción

Pruebas del componente realizado dentro del sistema y evaluación de su rendimiento al interactuar con el resto de componentes.

Objetivos

- Asegurar que el elemento terminado funcione con el resto del videojuego.

3.2.6 Proceso: Reporte de resultados

Descripción

Basado en las pruebas realizadas, describir el rendimiento del componente y plantear mejoras, ajustes y correcciones de ser necesarias. En este proceso debe considerarse cual es la condición de para del desarrollo, osea, evaluar si todos los requerimientos del videojuego están implementados para dar por terminada la producción.

Objetivos

- Finiquitar el desarrollo del elemento del videojuego para pasar a la siguiente iteración.
- Agregar nuevos elementos al diseño del videojuego.
- Eliminar elementos del diseño del videojuego de ser necesario.
- Evaluar condición del proyecto para publicarlo.

3.2.7 Proceso: Publicación

Descripción

Con el videojuego desarrollado en su totalidad, se prepara el producto para que sea distribuido en la plataforma digital establecida. Una vez publicado, se estudia el impacto que tiene videojuego en los consumidores.

Objetivos

- Publicar el videojuego en una plataforma digital.
- Recolectar información en busca de errores de implementación.
- Mejorar el del videojuego.

3.2.8 Tareas y productos

A continuación, se describen las tablas 3.1, 3.2, 3.3, 3.4, 3.5, 3.6 y 3.7, las tareas y productos resultantes de los procesos que conforman a la fase de implementación del método de desatollo DIY.

Tarea	**Consideraciones**	**Producto**
1- Herramientas a usar.	- Licencias de uso. - Habilidad requerida para utilizar las herramientas. - Comparar con otras herramientas parecidas para asegurar que la selecciona cumpla mejor su objetivo.	- Escoger que herramientas se usaran a los largo de la implementación.
2- Aprender a usar las herramientas.	En caso de que la habilidad requerida para el uso de las herramientas no sea suficiente, se debe aprender a utilizarla hasta un nivel que permita cumplir los objetivos planteados en la selección de herramientas. Se toma esto en consideración para asegurar que el flujo de trabajo sea lo mas rentable posible, en cuanto al tiempo. Recordando siempre la filosofal de trabajo de Beck (1999) con su método de desarrollo de software XP	- Habilidad en las herramientas seleccionadas.
3- Delimitar el diseño general de videojuego a implementar.	Los distintos elementos diseñados para el videojuego resultante en la aplicación de la fase de diseño del método de desarrollo DIY por Centeno (2018), pueden ser a vista previa, muy complejas de implementar. Sin embargo, siempre debe tomarse en cuenta la disposición del tiempo, habilidades de los desarrolladores que se encargan de la implementación y fecha de publicación del producto mínimo viable.	- Posibles cambios en el diseño general del videojuego.

Tabla 3.1: Proceso: Preparación

Tarea	Consideraciones	Producto
1- Revisión de los elementos que no han sido implementados.	- La implementación del elemento debe hacerse en el menor tiempo posible. - Preferir elementos básicos, osea que no sean complicados de implementar. - Considerar la cadena de trabajo definida en el documento generado en la fase de diseño.	- Elemento del videojuego a implementar.
2- Diseñar mediante diagramas la estructura del elemento.	- En esta tarea es donde puede descubrirse la complejidad de los elementos a implementar.	- Diagramas conceptuales que ayuden en la implementación del elemento.
3- Detectar problemas en el diseño del elemento a implementar que comprometa el tiempo a invertir.	Si el diseño planteado resulta muy difícil de implementar, entonces dividir tanto como se pueda el diseño del elemento y con sus partes volver a la tarea anterior para seleccionar que se va a implementar.	Fragmentar el elemento de ser necesario.

Tabla 3.2: Proceso: Estudio de prioridades

Tarea	Consideraciones	Producto
1- Implementar el elemento.	- Las herramientas seleccionadas deben ayudar para que la implementación se complete rápidamente. - Diseño del elemento descrito. - Mantener la mente abierta. quiere decir que en el transcurso de la tarea quizás surge una mejor manera de implementar el elemento, y de ser necesario, regresar a la primera etapa del proceso de producción para ajustar el diseño del elemento. Estas consideraciones se hacen en base al flujo de trabajo que presenta Martin (1993) en su método de desarrollo rápido de aplicaciones o RAD. El cual se enfoca en el desarrollo rápido de una aplicación y adaptar las herramientas o flujos de trabajo con el propósito de entregar un producto de calidad.	- Funcionalidad de un elemento del videojuego.
2- Prueba unitaria del elemento.	- Ninguna.	- Posibles errores en la implementación del elemento.

Tabla 3.3: Proceso: Implementación y pruebas

Tarea	Consideraciones	Producto
1- Acoplar el elemento implementado con el sistema.	- Estar atento de problemas que puedan surgir dependiendo del paradigma de programación que se esta usando.	- Funcionalidad en el videojuego.

Tabla 3.4: Proceso: Integración de componentes

Tarea	Consideraciones	Producto
1- Probar exhaustivamente el elemento integrado al sistema en busca de fallas.	- Ninguna.	- Funcionalidad terminada del videojuego.

Tabla 3.5: Proceso: Pruebas en el sistema

Tarea	Consideraciones	Producto
1- Reportar observaciones de la implementación del elemento al resto del equipo de trabajo.	- Los problemas que surgieron al implementar el elemento. - La eficiencia de las herramientas utilizadas. - El tiempo que se invirtió.	- Nuevas estrategias de trabaja que pueden mejorar el proceso de producción.
2- Discusión con el equipo para pasar a la siguiente iteración.	- El producto formado por la tarea anterior.	- Terminación de la iteración. - Finalización de la implementación del proyecto en la ultima iteración.

Tabla 3.6: Proceso: Reporte de resultados

Tarea	**Consideraciones**	**Producto**
1- Generar aplicación.	- Plataforma de ejecución: Windows, GNU/Linux, Android, etc.	- Producto listo para ejecutarse.
2- El mercado.	- Todo mercado de distribución digital tiene políticas para subir contenido, obtener licencias, generar publicidad, recibir los ingresos que genere el producto y otros factores que deben ser estudiados previos a la publicación del videojuego. - Luego de ubicar el producto en el mercado se debe definir cuales serán las estrategias de distribución, las campañas publicitarias, las redes sociales o cualquier forma de hacer que el publico conozca y se interese por el videojuego.	- Descripción de las políticas del mercado. - Descripción de las estrategias de distribución.
3- Publicación.	- Ninguna.	- Publicación del producto en las plataformas de distribución digital.
4- Análisis.	- La prueba mas grande del producto es su desempeño en el mercado. El análisis de las estadísticas que genere son la clave para determinar los fallos o puntos a mejorar del videojuego.	- Conocimiento de las fallas o mejoras necesarias.
5- Reporte.	- Partiendo del análisis de las estadísticas del mercado se debe generar un reporte de errores y mejoras que luego debe pasar por el proceso de Implementación.	- Reporte de errores y mejoras.
6- Implementar correcciones.	- Reporte producido en la tarea 5.	- Producto refinado.

Tabla 3.7: Proceso: Publicación.

Capítulo 4

El Producto: Aplicación del método de desarrollo DIY

Debe considerarse que lo expuesto a continuación es el desarrollo del videojuego aplicando el método de desarrollo DIY en su fase de implementación. Los siguientes elementos y características fueron desarrolladas en base a un documento desarrollado por Centeno (2018) al aplicar la fase de diseño del mismo método de desarrollo.

4.1 Proceso: Preparación

4.1.1 Unity

Gracias a la popularidad que alcanzó este motor de videojuego, la cantidad de material enfocado al aprendizaje disponible en el Internet es inconmensurable. Además, la curva de aprendizaje del lenguaje de programación C# que se utiliza para el scripting, es mucho mas accesible que la de sus lenguajes padres C/C++. Por ultimo, su versatilidad en cuanto a las plataformas que el videojuego puede ser exportado, estos son:

- **Web:** WebGL.
- **PC:** Windows, GNU/Linux, OS X.
- **Consolas:** PS Vita, PS4, Xbox 360, Xbox One, Wii U, Nintendo 3DS, Nintendo Switch.

- **Dispositivos móviles:** Android, iOS, Windows Phone.
- **Dispositivos de realidad virtual:** Oculus Rift, HTC Vive, Microsoft Hololens, PlayStation VR, Samsung VR.

¿Cómo funciona?

El funcionamiento de Unity se puede dividir en tres capas, véase la Figura 4.1. La primera capa que es la mas intuitiva de todas, la interfaz gráfica de Unity se encarga de editar los distintos componentes del videojuego en desarrollo; la segunda capa desde un enfoque macro con respecto a los recursos (*assets* de ahora en adelante) que utiliza el videojuego creado desde Unity; y la tercera capa que consiste en el paradigma de programación (explicado en la sección 3 del capítulo 2 de este documento).

Para entender la segunda capa del funcionamiento de Unity, hay que diferenciar los *Assets* de los *GameObject* como se muestra en la tabla 4.1 y especificar que es una Escena.

Tabla 4.1: Comparación entre *Assets* y *GameObjects*

Objeto	**Assets**	**GameObjects**
Instancia	Dentro o fuera del videojuego. Es lo que el Jugador no puede ver (BACK-END)	Solo dentro de una escena. Es lo que el Jugador puede ver (FRONT-END)
Quien puede modificarlo	Otros *assets* o *GameObjects*	El objeto mismo u otros *GameObjects*
Función	Actúan principalmente como contenedores pero pueden tener funcionalidad.	Representar entidades

Las escenas son *assets*, como se observa en la estructura general de Unity, véase Figura 4.1, y funcionan como una interfaz entre el programa, osea el videojuego, y el jugador; visto desde otro punto de vista, son pequeños programas que en conjunto conforman el videojuego. Dentro de las escenas es que pueden existir las entidades del videojuego (avatar del jugador, terreno, interfaz gráfica del videojuego, NPC, etc).

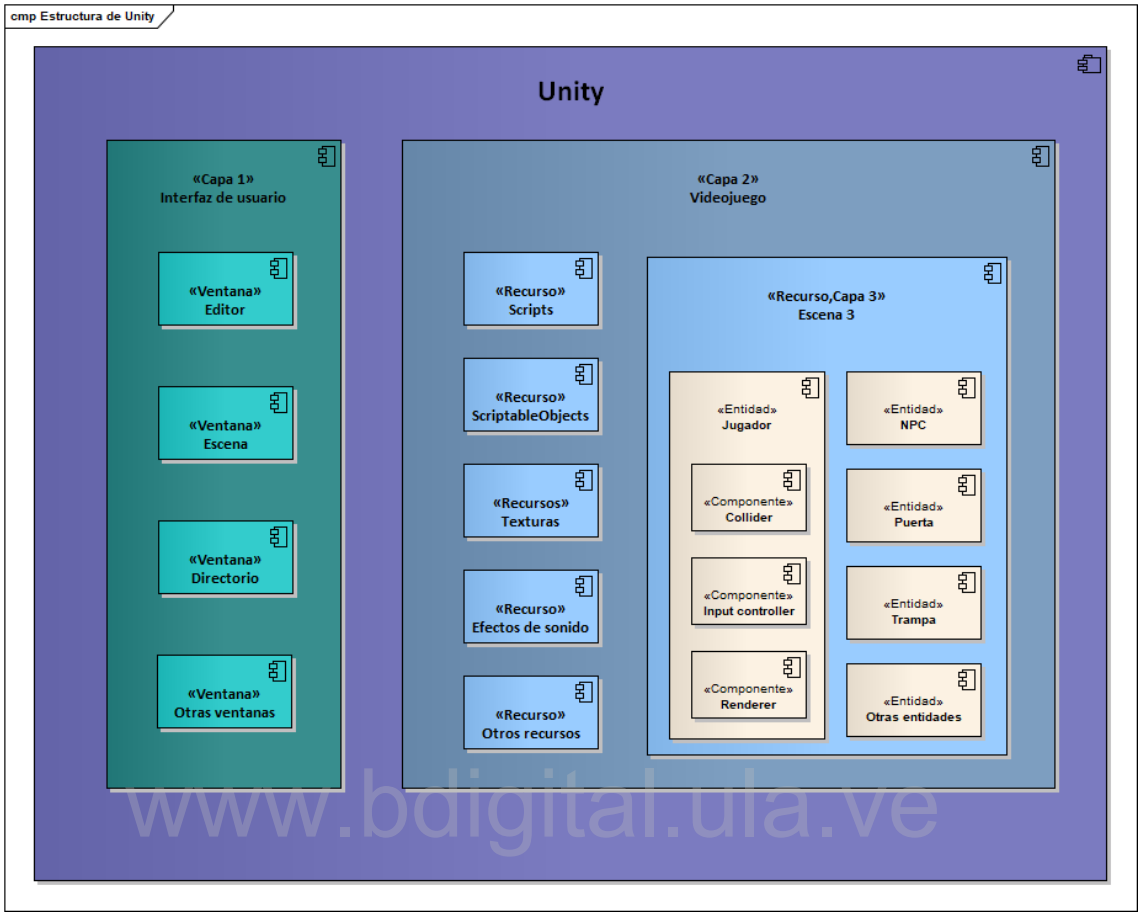

Figura 4.1: Estructura general de Unity

Véase Figura 4.2.

Entonces de forma muy general se pueden mostrar las relaciones posibles entre los *GameObjects* y los *Assets*, véase Figura 4.3. Es importante entender estas relaciones ya que al igual que los componentes de los *GameObjects* (siguiendo el paradigma EC), los *assets* pueden ser programados por los desarrolladores.

4.1.2 Otros recursos

La selección del resto de recursos y herramientas, al igual que Unity, se hicieron por dos razones principales. Las licencias de uso son gratuitas y la habilidad requerida

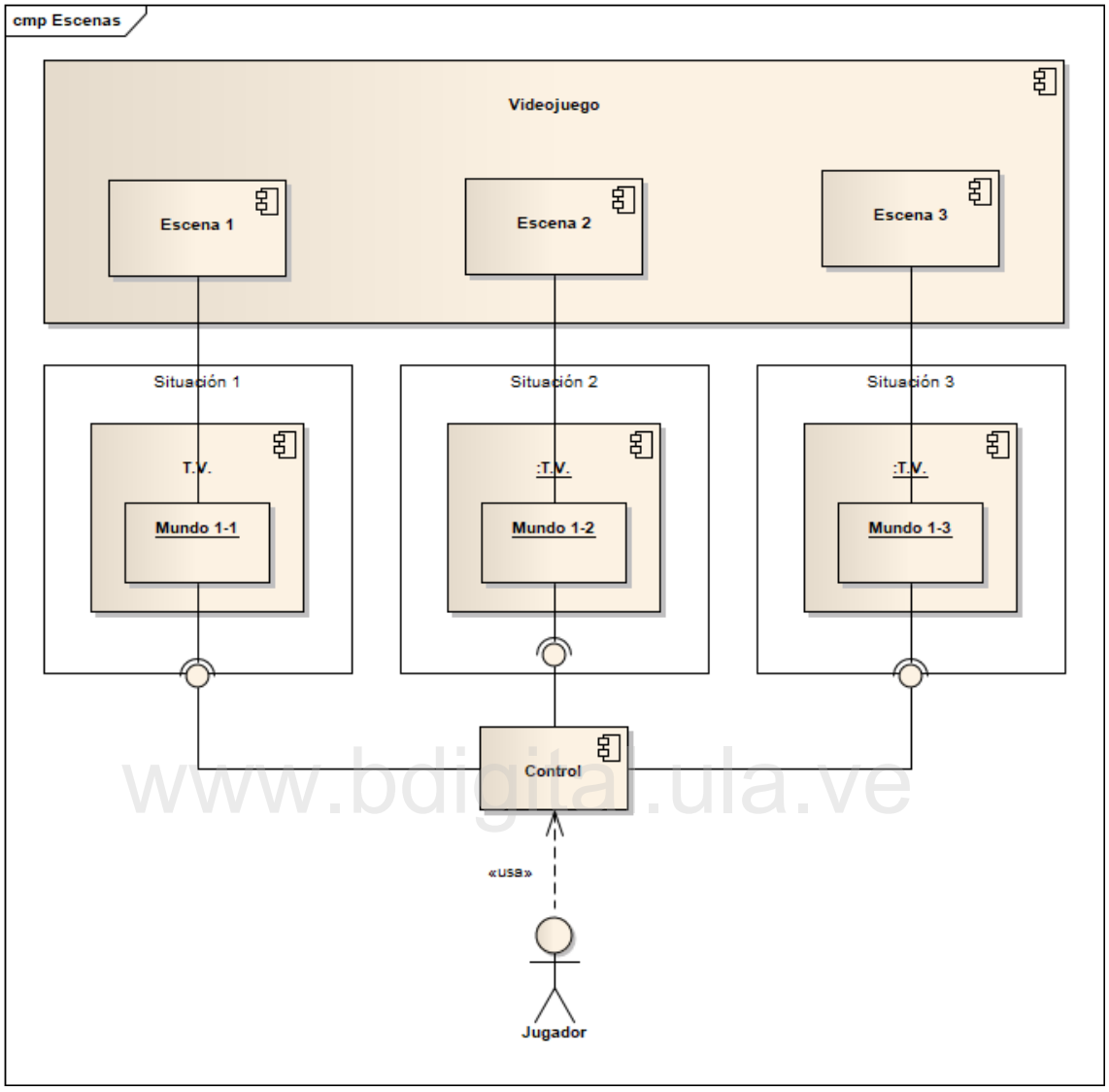

Figura 4.2: Composición y funcionamiento de las escenas en Unity

para el uso o entendimiento de estos no es alta.

- Blender como software para el modelado de figuras 3D y animaciones.
- Assets Store de Unity, sitio para descargar *assets* que pueden ser pagos o gratuitos.
- Patrones de diseño, recurso intelectual imprescindible para la implementación de

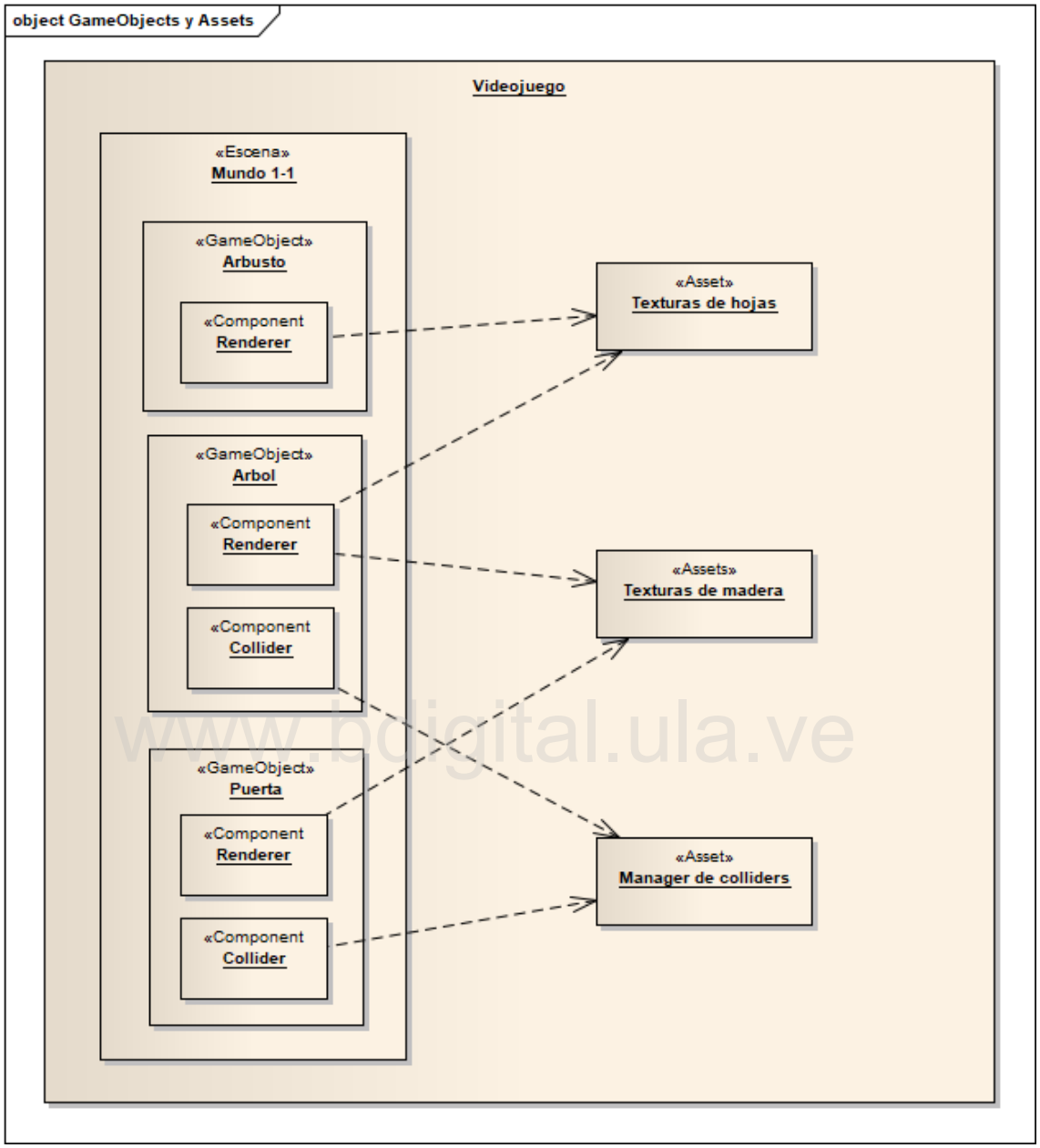

Figura 4.3: Relaciones entre *GameObjects* y *Assets*

conceptos, diseños e ideas. Material tomado de los libros Desing Patterns por Erich Gamma y Vlissides (1997) y Game Programming Patterns por Nystrom (2014).

- La comunidad de Internet. Las habilidades necesarias para usar una herramienta

en cualquier ámbito no se aprenden de la nada. Gracias a la era digital y por la naturaleza del software, los estudiantes pueden aprender prácticamente lo que sea, jugando y viendo como otras personas trabajan en el software. Es un recurso que debe ser apreciado de igual manera que los conocimientos adquiridos mediante artículos o libros. Varias de las implementaciones hechas en esta fase se inspira en los trabajos realizados por Asbjorn (2012) en su canal de YouTube.

Posterior al proceso de preparación

- TouchScrip son unos *assets* para Unity desarrollados por Simonov (2016), que permiten manejar gestos para pantallas táctiles de varios dispositivos, como son los casos de la mayoría de celulares y tablets modernos (Android y Apple), pantallas táctiles de laptops y/o computadoras de escritorio. Para estos últimos, TouchScript simula el funcionamiento de una pantalla táctil mediante un mouse.
- TextMeshPro desarrollado originalmente por Bouchard (2017), ahora propiedad de Unity Technologies, son *assets* que implementan renderizado de texto en alta calidad a las escenas de Unity.

4.2 Proceso: Producción

4.2.1 Sistema de cámara

El diseño del funcionamiento de la cámara se inspiró en los primeros juegos de Resident Evil de Mikami (1996). A medida que el personaje se mueve por el escenario, la cámara se mantiene fija hasta que el personaje sale de los limites de la toma, justo después, la cámara cambia instantáneamente la toma. Para este videojuego, es esencialmente lo mismo con la diferencia de que los cambios de tomas son transitorios y no instantáneos, quiere decir que la cámara se mueve y rota de un punto a otro según la posición del personaje en el escenario, véase Figura 4.4. Este diseño para la cámara junto con la estética del videojuego compaginan muy bien.

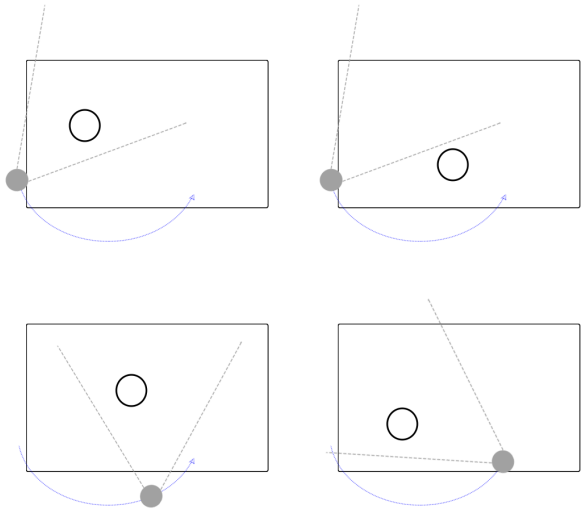

Figura 4.4: Diseño de la cámara.

Para la implementación, el diseño se divide en dos componentes (siguiendo el paradigma EC). El primer componente es un *Script* que se acopla a la *Cámara* y se encarga de mover y rotar de forma transitoria al componente *Transform* - componente que tiene la información de la posición, rotación y escala de un *GameObject* dentro de una escena en Unity. Todos los *GameObjects* contienen el componente *Transform*.

El segundo componente también es un *Script*, llamado *Gatillo* y se acopla a un *GameObject* llamado *Gatillo de cámara* que ademas tiene como componente un *Collider* que delimita una zona de activación; el *Gatillo* se encarga de avisarle a la *Cámara* cuando el *Personaje* entra en la zona del *Gatillo de cámara*, y en caso de que la *Cámara* no este situada en el punto de anclaje de dicha zona, entonces la *Cámara* se traslada. Vease Figura 4.5 y para la Figura 4.6 :

1. Posición actual de la cámara.
2. Zona de activación para trasladar la cámara.
3. Posición de anclaje para la cámara, este elemento tiene una linea verde que indica la rotación que debe tener la cámara.

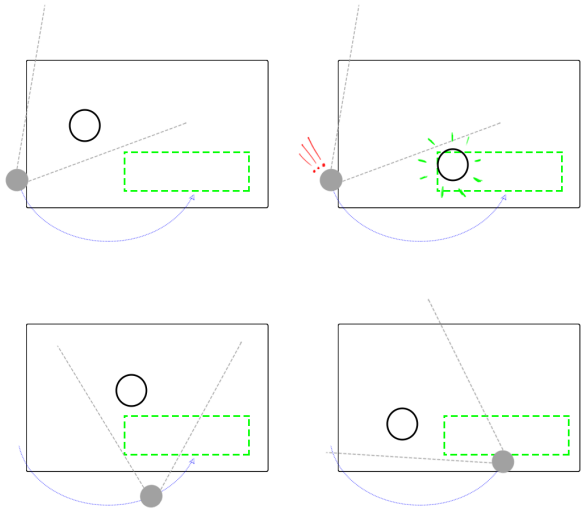

Figura 4.5: Implementación de la cámara.

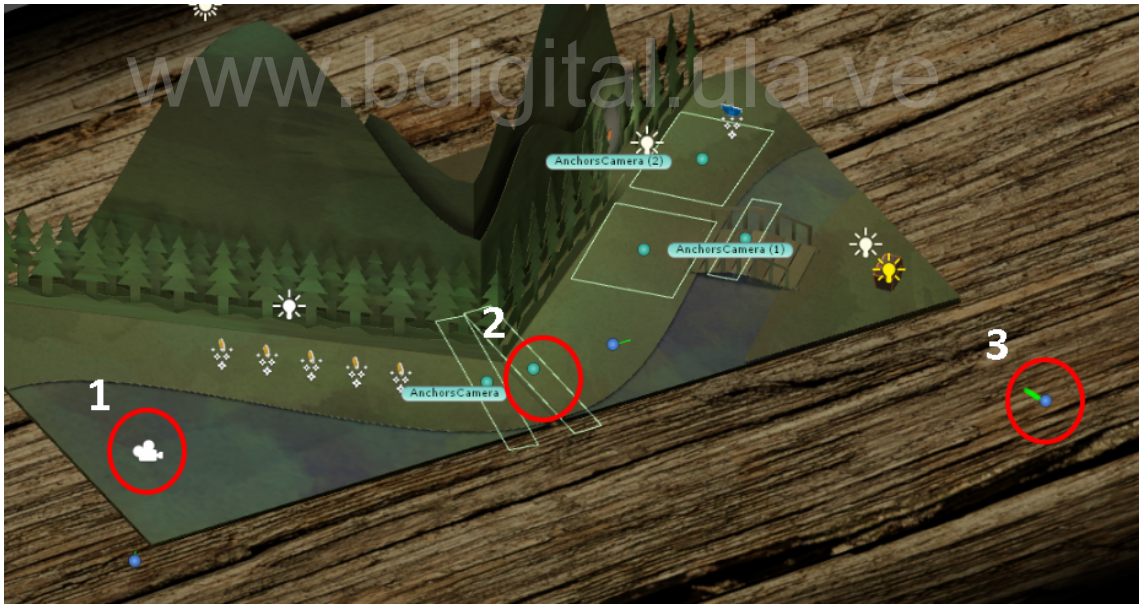

Figura 4.6: Vista de la implementación de la Cámara en Unity.

4.2.2 Movimiento del personaje

Para la implementación del movimiento del personaje se tomaron en cuenta las siguientes restricciones del videojuego:

- Mundo en 3D.
- Posición de la cámara.
- Controles no invasivos. En el contexto de dispositivos móviles (descripción en la siguiente sección).

Respetando las restricciones se diseña el movimiento a partir del vector que se genera al interactuar con la pantalla. Este vector se proyecta al plano base del personaje como se muestra en la Figura 4.7.

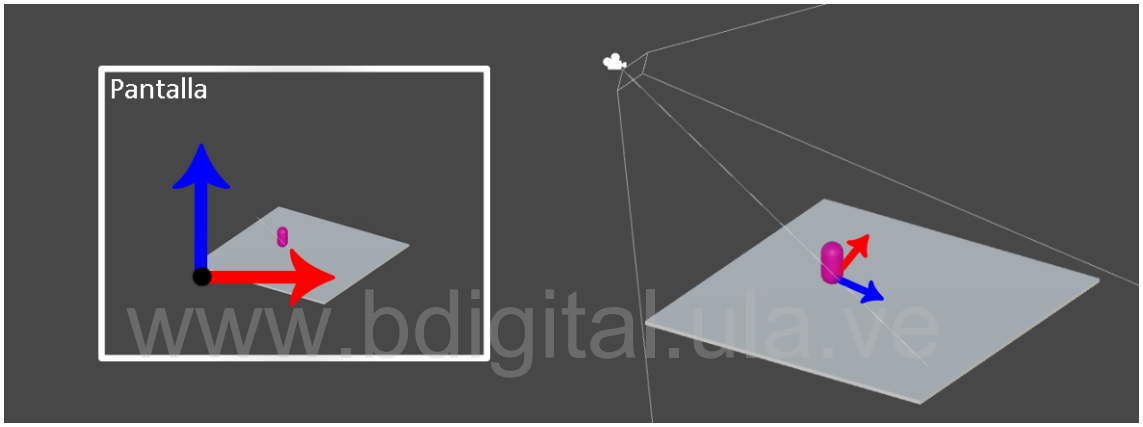

Figura 4.7: Proyección al plano del personaje.

Entonces dado el vector de dirección, se implementa el componente *Motor de movimiento* para acoplarse al *Personaje* y haciendo uso de sus componentes *Transform* y *NavMeshAgent*. Esta funcionalidad se muestra en las Figuras 4.8 y 4.9.

4.2.3 Controles

En un inicio el videojuego estaba enfocado para ser ejecutado en dispositivos móviles, bajo esta premisa el diseño general de los controles no sería muy complicado. Generalmente los controles para dispositivos móviles o basados en pantallas táctiles constan de *Joysticks* virtuales que simulan el funcionamientos de los controles físicos como el DualShock de la consola PlayStation. Sin embargo, dicho planteamiento tenía un alto costo para los requerimientos del videojuego en desarrollo. Específicamente en

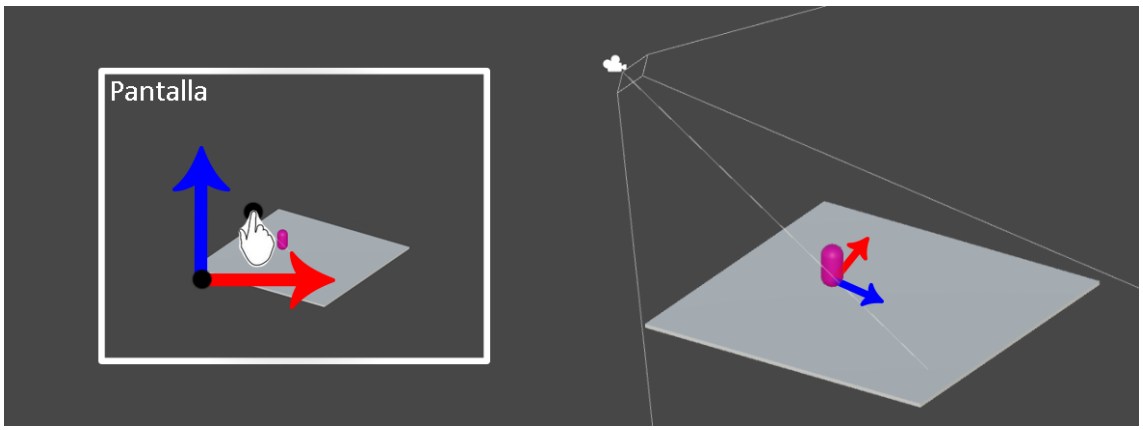

Figura 4.8: Ejemplo gráfico del movimiento del personaje. Parte 1.

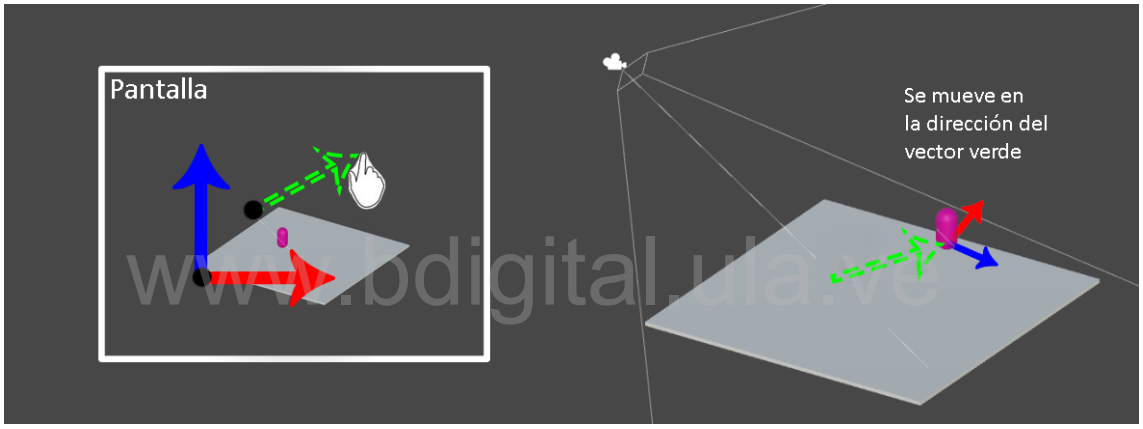

Figura 4.9: Ejemplo gráfico del movimiento del personaje. Parte 2.

la inmersión que se quería que tuviese el videojuego, y un control virtual anclado en la interfaz gráfica rompería con la inmersión. Con la idea de mantener la inmersión del videojuego se planteó que los controles se implementaran mediante gestos, véase Figura 4.10, estos serian:

- Drag: Este gesto seria el encargado de controlar el movimiento del personaje.
- Tap: Gesto para interactuar con las entidades del videojuego.
- Flick: Gesto usado para la navegación entre ventanas de la interfaz.

Entonces al basarse en estos gestos, la implementación debía transmitir la información generada por los gestos al videojuego. La forma mas rápida de desarrollar

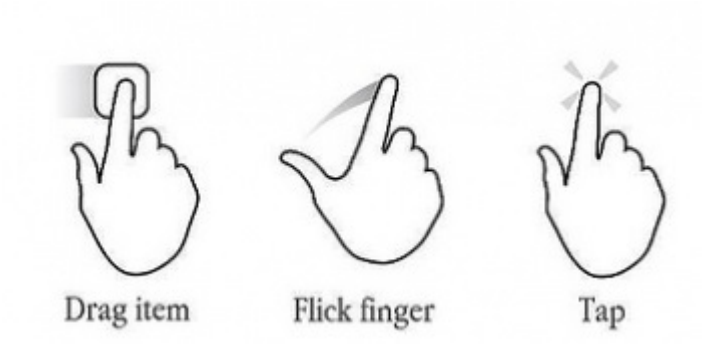

Figura 4.10: Gestos para los controles

esta funcionalidad fue con el uso de TouchScript, el cual se encarga de manejar los punteros entrantes (refiriéndose a cuando se practica un gesto al tocar la pantalla táctil, o por la simulación mediante el mouse) y proyectarlos sobre capas, como lo explica Simonov (2017) en su manual.

Los punteros en TouchScript son objetos que contienen información de su punto de origen en un sistema de referencia cartesiano 2D proyectado en la pantalla, el tiempo de vida del puntero, posición actual en el sistema de referencia y estado (estático, en movimiento, etc). TouchScript con esta información reconoce los gestos que el desarrollador elige (para este caso, Drag, Tap y Flick) los cuales están definidos por defecto en la herramienta.

Finalmente por parte del desarrollador queda manejar los eventos que genera TouchScript al reconocer un gesto. Para los gestos Tap y Flick simplemente se le acoplaba métodos o funciones mediante el manejador de eventos de Unity, por el contrario, el gesto Drag debía ser procesado. Con la información del puntero que forma al gesto Drag, se construye un vector que sirve de dirección para el movimiento del personaje en el videojuego.

Una gran ventaja de implementar los controles usando los recursos que ofrece TouchScript, es que el sistema sirve para cualquier dispositivo móvil e incluso computadoras y laptops.

4.2.4 Interacciones

Las interacciones definen como el Personaje interactua con el resto de los elementos del videojuego: enemigos, NPC, puertas, cofres, botones, etc. Y por esa razón es uno de los elementos mas interesante y complejo a implementar. Con la limitada variedad de controles, la implementación de las interacciones se divide en dos partes.

¿Cómo puede una entidad reaccionar según un tipo de estimulación?. Esta pregunta es lo que define el diseño de la primera parte de las interacciones, por ejemplo: Se tiene una puerta que esta bloqueada y puede reaccionar de dos formas diferentes, si la intentan abrir sin la llave, entonces la puerta reacciona moviéndose un poco o produciendo sonido pero sin abrirse, ahora, si la intentan abrir con la llave entonces la puerta se abre y cambia su estado. Ya no esta bloqueada, sino abierta. En este nuevo estado la puerta puede reaccionar a la acción de cerrar. Este ejemplo se ilustra en la Figura 4.11.

Componente *Interactable*

Es gracias al ejemplo anterior que se implementa una *máquina de estados basada en funciones*; se define igual que un grafo dirigido, pero sus nodos o estados son de tipo *Action*, el cual esta incluido en el *namespace System* de C#. Estos tipos de estados permiten pasar métodos y funciones como argumentos. Volviendo al ejemplo de la puerta, cada vez que el Personaje interactua con ella, el componente *Interactable* que tiene acoplado, maneja la máquina de estados para ejecutar la función del estado actual. Así pues, con un solo gesto de parte del Jugador, el Personaje puede tener mas de una manera - según el objeto que tenga equipado, como un hacha, una lámpara, una llave o nada - de interactuar con una entidad o *GameObject* dentro del videojuego.

Como se observa en el diagrama de la Figura 4.12, el componente que se acopla al *GameObject* de la *Puerta*, define varios métodos: *Close(), Lock()* y *Open()*. Estos métodos son utilizados para la construcción de la *maquina de estados basada en funciones (MSF)*. Ademas, el método heredado *Interact(Item)*, el cual es quien invoca el estado actual de la *MSF* y espera a ser ejecutado por el Personaje.

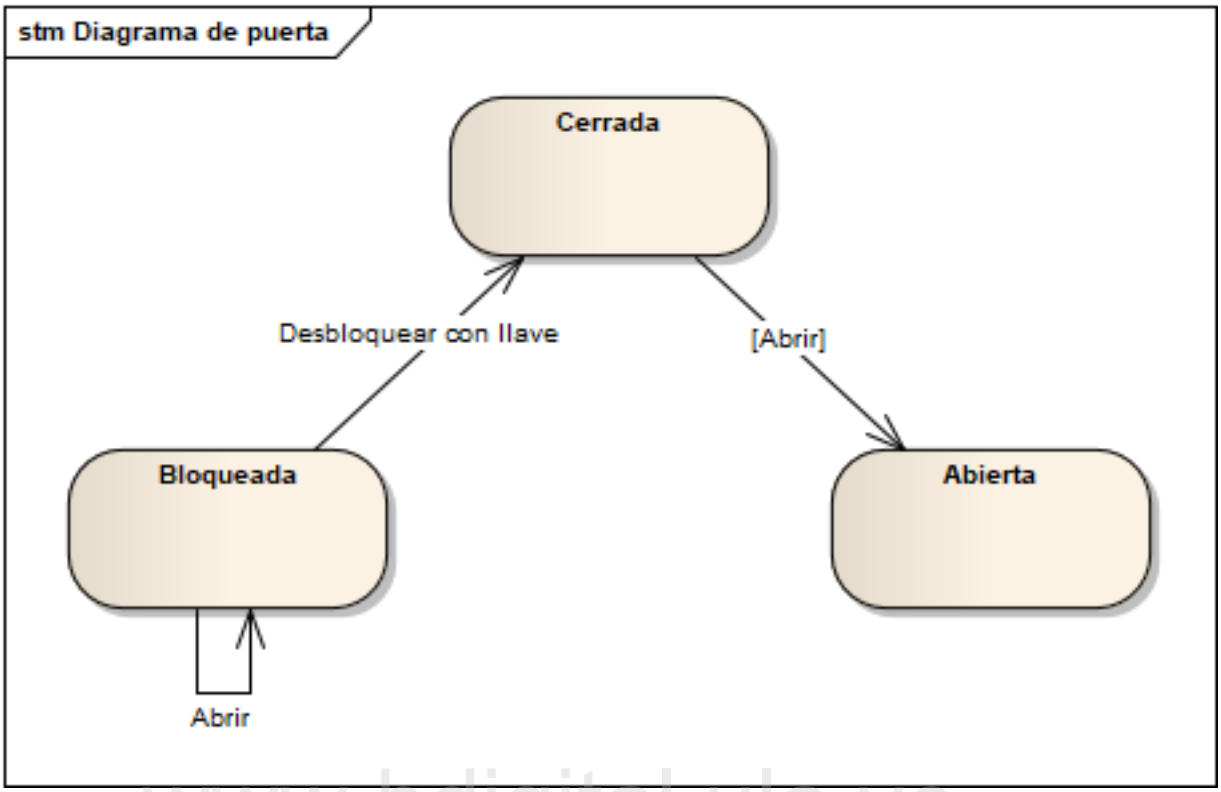

Figura 4.11: Funcionamiento de una puerta.

Componente *Actioner*

El segundo componente forma parte del *Personaje*, simplemente "usa" el objeto que tiene equipado para interactuar con un *GameObject*, solo si dicho *GameObject* tiene un componente *Interactable*. Esto se consigue gracias al método *InRange()* que detecta cuando *Personaje* esta en rango para interactuar. Véase Figura 4.12.

4.2.5 Blender

La mayor parte del aspecto gráfico del videojuego fue importado de Blender. Para los modelos gráficos del Personaje, Enemigos, Interfaz "tangible", escenarios o terrenos, Objetos, etc. Para los elementos estáticos, simplemente se arrastra el archivo (el cual desde el punto de vista de Unity, es un Asset) a la escena de Unity. Este fue el proceso para añadir los niveles y demás modelos estáticos. Véase Figura 4.13.

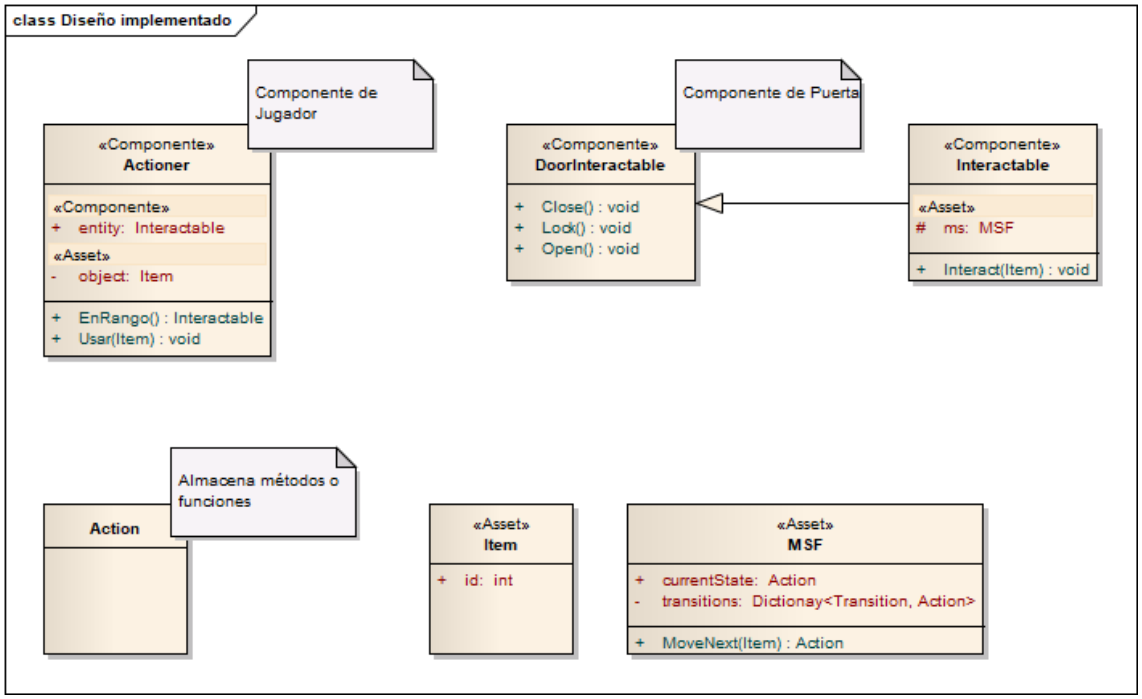

Figura 4.12: Diagrama de clases general del sistema de interacciones.

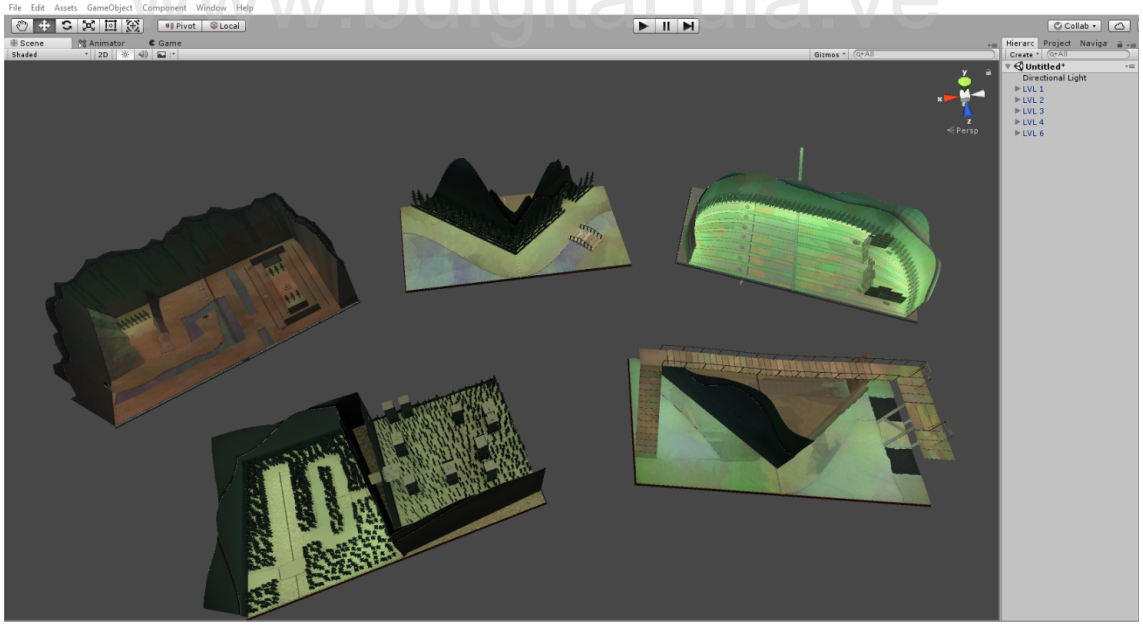

Figura 4.13: Niveles y otros elementos importados de Blender a Unity.

Para los otros elementos importados de Blender, los que contienen sus propias

animaciones, basta con saber utilizar los *AnimatorController*. Estos elementos que ofrece Unity, permite controlar mediante maquinas de estado, las animaciones de los modelos importados. Algunos ejemplos: Figura 4.14, Figura 4.15.

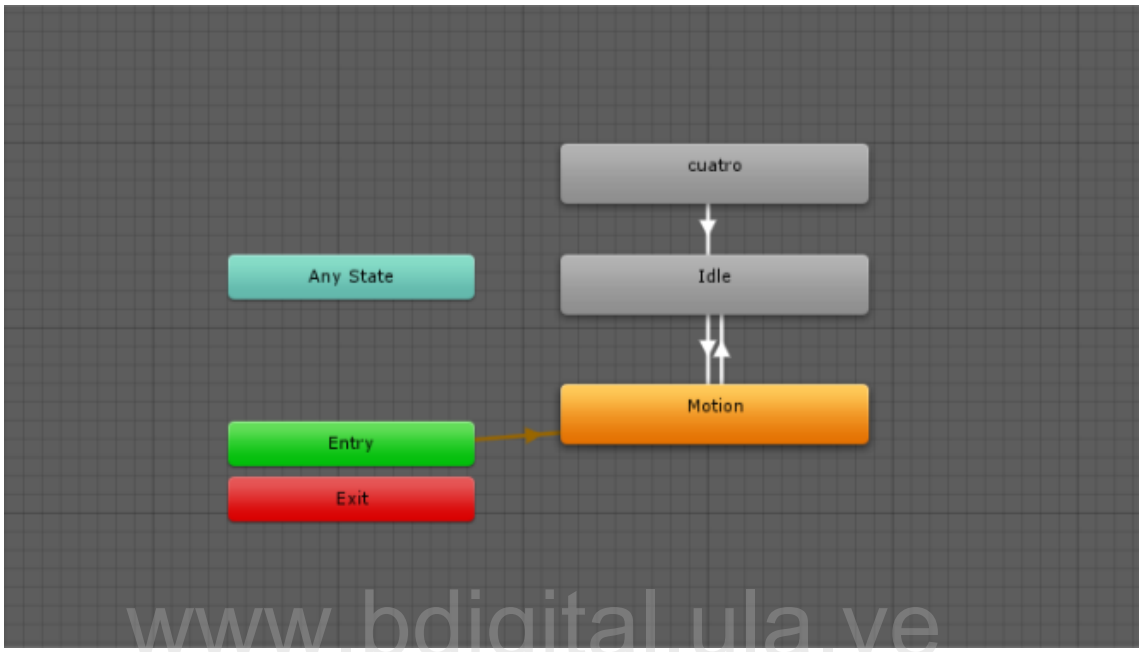

Figura 4.14: Controlador de animaciones para el Personaje.

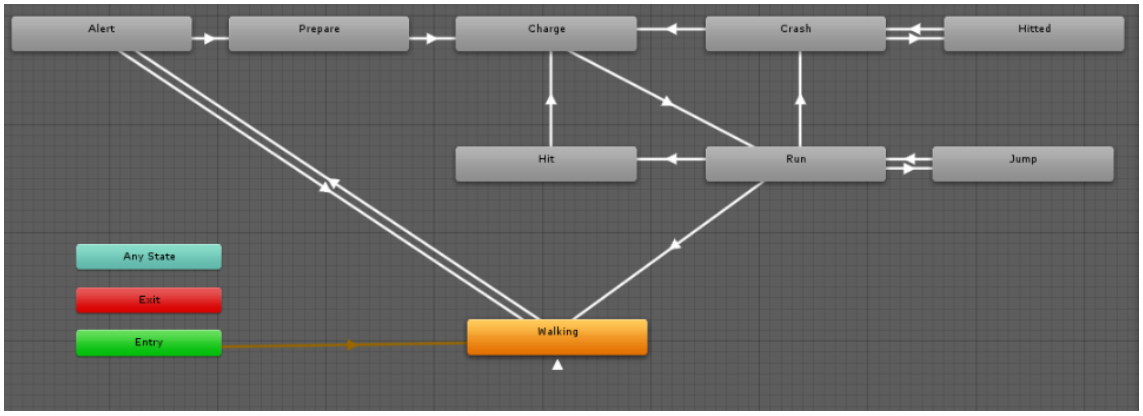

Figura 4.15: Controlador de animaciones para un enemigo.

4.2.6 Enemigos

Movimiento: Componente Patrol

Al igual que el Personaje, el movimiento de los enemigos se implementar haciendo uso del componente *NavMeshAgent*. Este componente necesita una superficie por la cual moverse, y de esto se encarga Unity al decirle que objetos dentro de la escena conformaran la superficie en las que los agentes puedes navegar. Pero para controlar estos agentes se necesita un componente extra; en cuanto al Personaje, de esto se encarga lo descrito en el movimiento del Personaje y los controles de movimiento. Para los enemigos basta con implementar un componente que recibe una lista de puntos, dichos puntos son GameObjects, los cuales fijan los lugares dentro de la superficie navegable por los que los enemigos pueden moverse. Vease Figura 4.16:

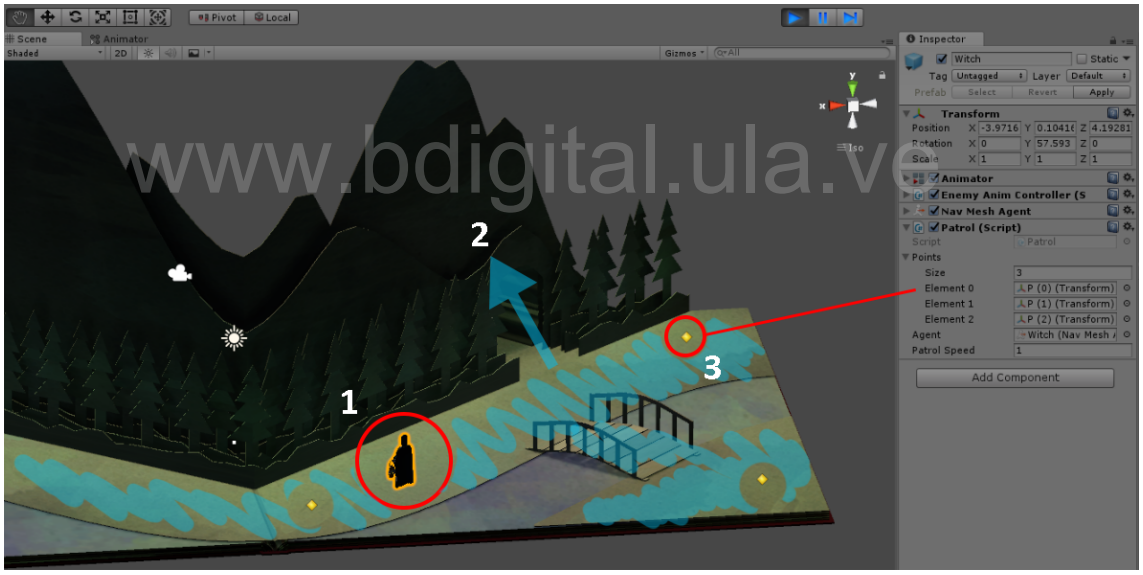

Figura 4.16: Vista de la implementación del movimiento de los enemigos en Unity.

1. Enemigo.
2. Superficie navegable.
3. Punto que define la ruta del enemigo mediante el componente Patrol.

Comportamiento: Componente Sight

La visión es una funcionalidad muy usada en los videojuegos, es a través de ella que las entidades interaccionen con el mundo que los rodea. La implementación para este producto, se inspira principalmente en como los enemigos se comportan en los videojuegos de Commandos, desarrollados por Pyro Studios (1998). El videojuego permite que el jugador seleccione a un enemigo para conocer su linea de visión, véase Figura 4.17.

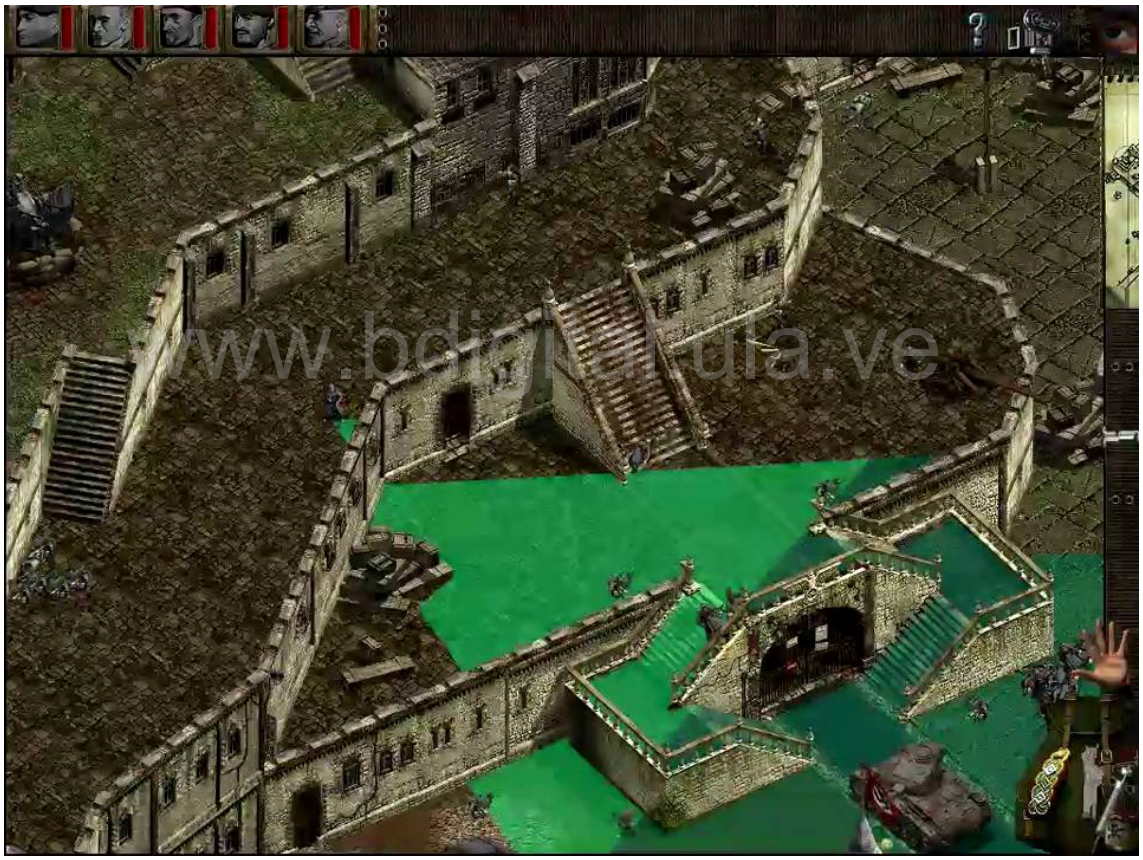

Figura 4.17: Visión de un enemigo en el videojuego Commandos.

Entonces, basado en la Figura 4.17 se implementa el siguiente diseño: Figura 4.18.

El componente implementado a partir del diseño anterior, utiliza unos "rayos" invisibles. Unity permite detectar si dichos rayos colisionan con otra entidad dentro

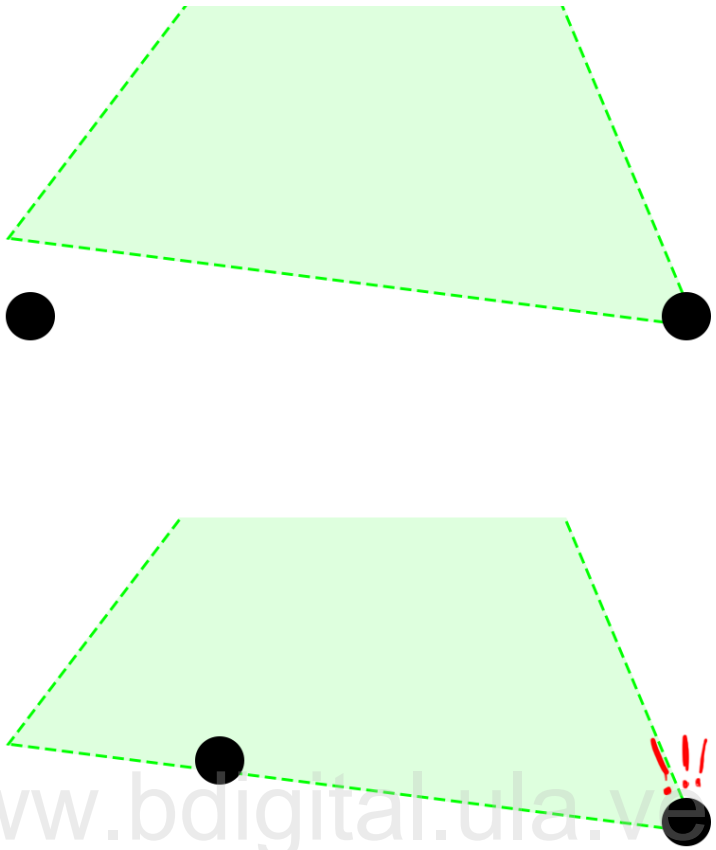

Figura 4.18: Diseño para el componente Sight.

del videojuego. Además, se puede modificar el rango de estos rayos, la cantidad de rayos y el ángulo que el conjunto de rayos puede forma. También cuando el objetivo de la visión es detectado, se dispara un evento de Unity, permitiendo que otros componentes ejecuten comportamientos a parte de este. Estos atributos definen el rango de visión. Véase Figura 4.19.

Desarrollo en base a animaciones

Partiendo de lo descrito sobre Blender en la subsection 4.2.5 se desarrolla el comportamiento del enemigo Toro en base a sus animaciones. Lo típico es que el

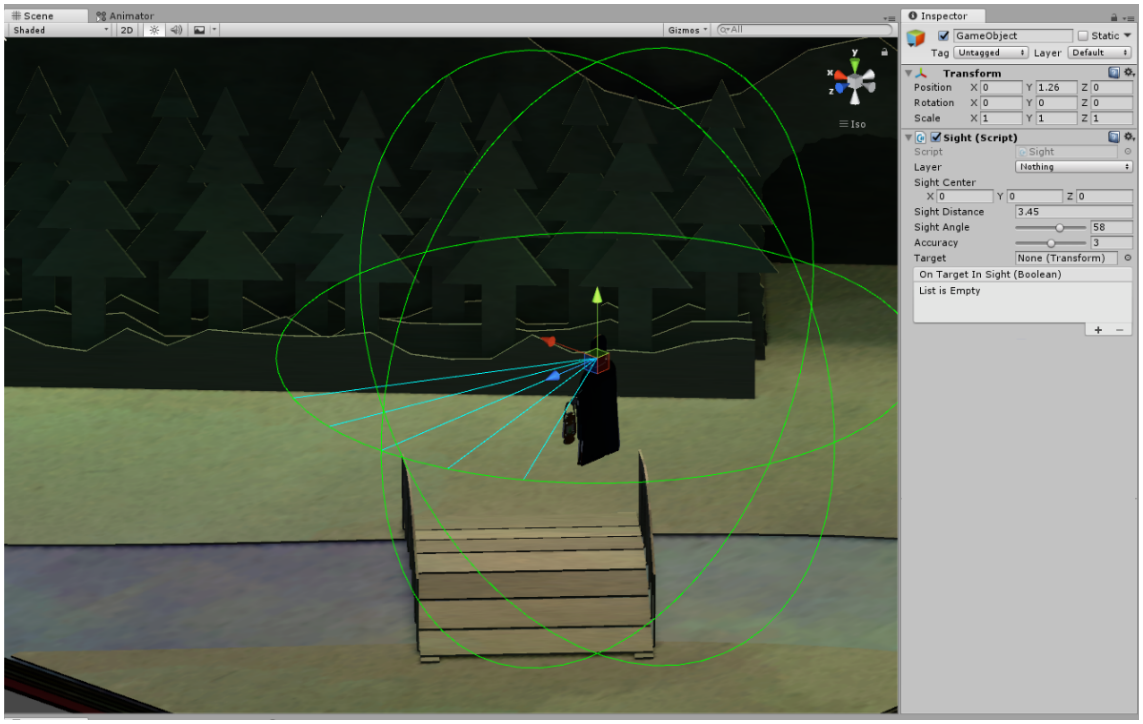

Figura 4.19: Componente Sight implementado en Unity.

AnimatorController sea controlado por otros componentes, y en principio esto se respeta pero solo para los estados de *walking* y *alert* del Toro, véase Fig 4.15. El resto de animaciones que están contenidas en la máquina de estados, disparan eventos. Son estos eventos los que ejecutan el comportamiento del Toro. Puede verse como otra forma de implementar la MSF descrito en la sección 4.2.4. En la Figura 4.20 las notas indican el comportamiento que ejecuta la maquina de estados y los nodos azules son los que reciben estímulos de otros componentes que son del propio *GameObject* del Toro u otras entidades.

4.2.7 Botones y puertas

Básicamente, estos elementos están conformados por componentes *Interactable*. Se definió que el Personaje interactua con los objetos que tiene este componente. Ahora estos mismos elementos pueden invocar eventos de Unity, así entonces, el Personaje interactua con el Botón, el Botón acciona su estado de *presionado* (a través de la MSF) y este estado actúa sobre la puerta. Este sistema se aprecia en la Figura 4.21.

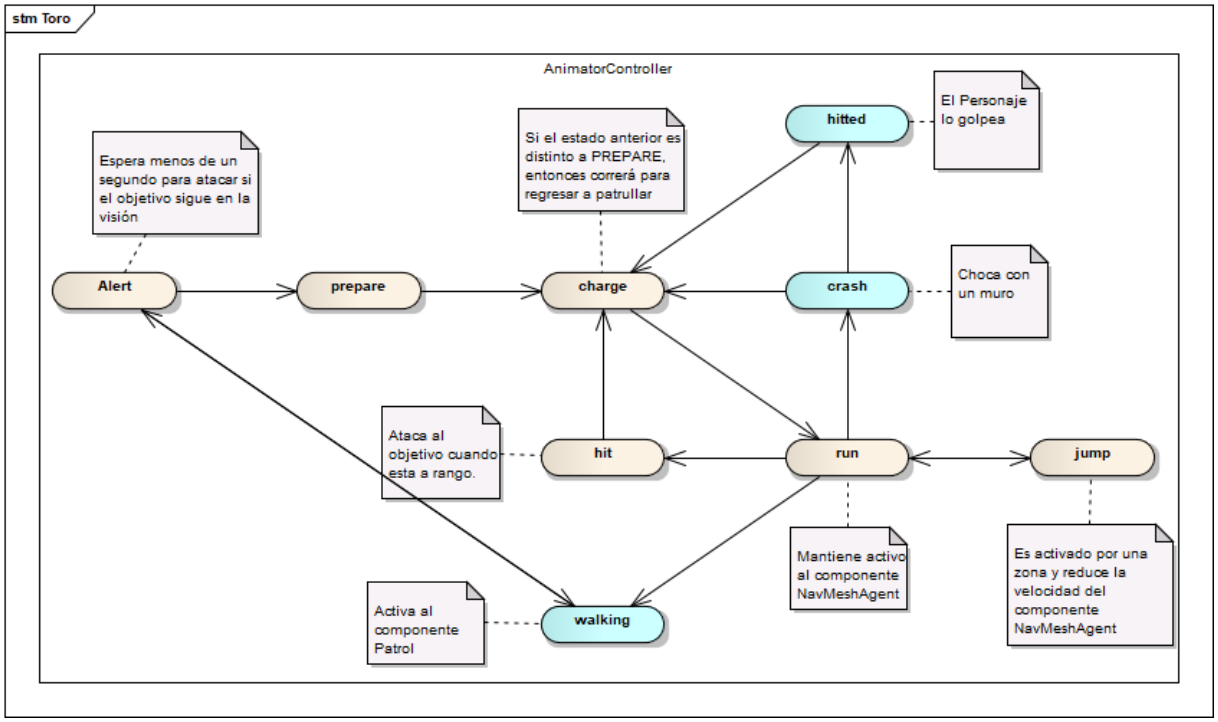

Figura 4.20: Comportamiento del enemigo Toro implementado.

4.2.8 Interfaz gráfica e inventario

Para respetar la estética del producto, la interfaz gráfica debe ser lo mas inmersa posible. El diseño para la implementación se definió previamente en la fase de diseño del método DIY, por eso, la interfaz gráfica se divide en dos parte: una parte "real", lo cual quiere decir que los elementos que la componen existen dentro del universo del videojuego; la otra parte, una interfaz típica que se superpone al universo del videojuego.

La implementación de la interfaz "real" se basa en un papel que tiene escrito información relevante de lo que pasa en el videojuego: objetos en el inventario y progreso del videojuego. Véase Figura 4.22. Para las interacciones, se utiliza los componentes que ofrece TouchScript, el cual permite navegar y utilizar los elementos que componen a esta interfaz. Es aquí donde se le da uso al gesto Flick para cambiar la posición de la cámara y poder ver los otros papeles.

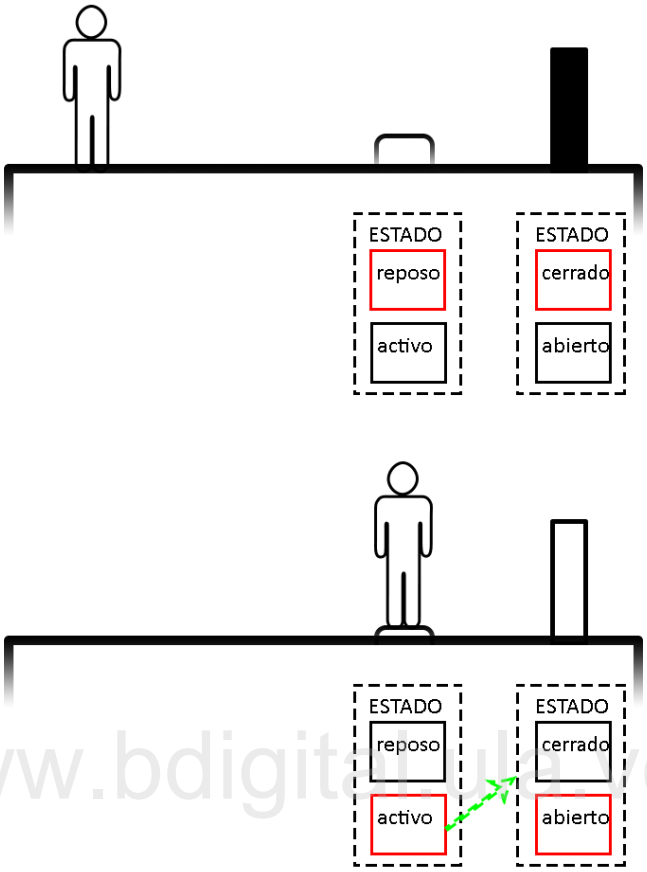

Figura 4.21: Sistema de interacción entre *GameObjects*.

4.2.9 Sistema de diálogos

Este sistema se basa en un tutorial de Asbjorn (2017). El cual consiste en dos elementos principales: Un manejador de diálogos implementado como un *GameObject* y diálogos implementados como *ScriptableObject*. Véase Figura 4.23.

DialogueManager espera a que cualquier otro *GameObject* que usa un Dialogo, lo llame para empezar a mostrar dicho Dialogo en la interfaz. El implementado en el videojuego funciona básicamente igual, pero el *Asset* Dialogo no contiene una sola sentencia si no varias y ademas, indica de quien es la sentencia dándole un nombre y

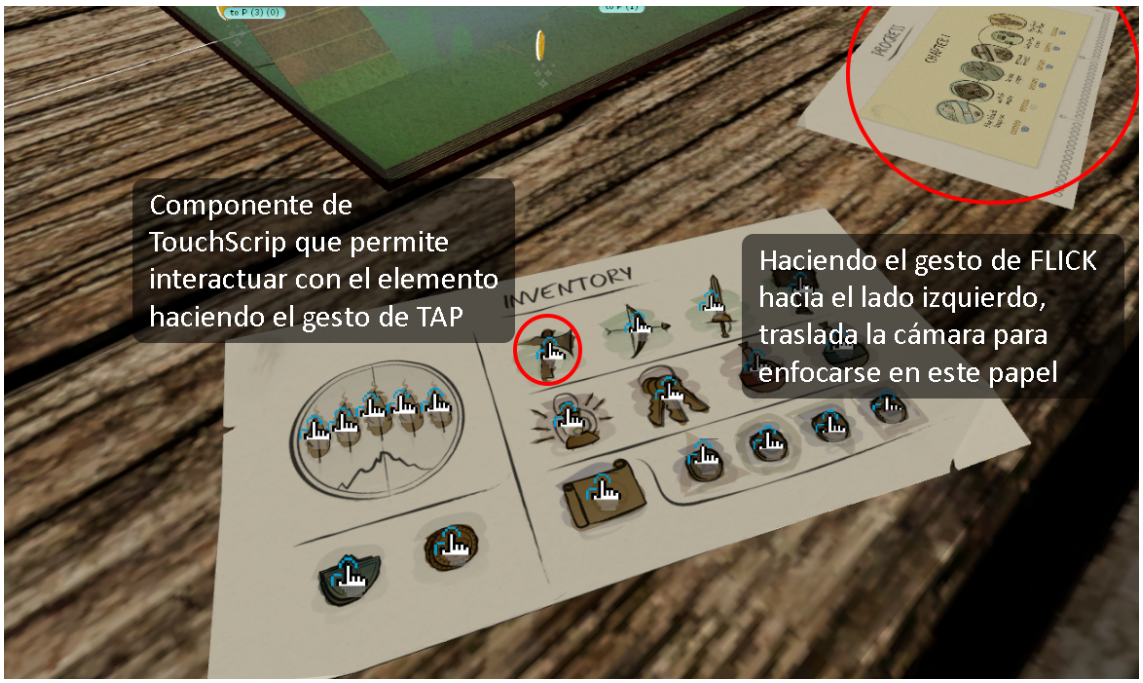

Figura 4.22: Vista de la interfaz "real" en Unity.

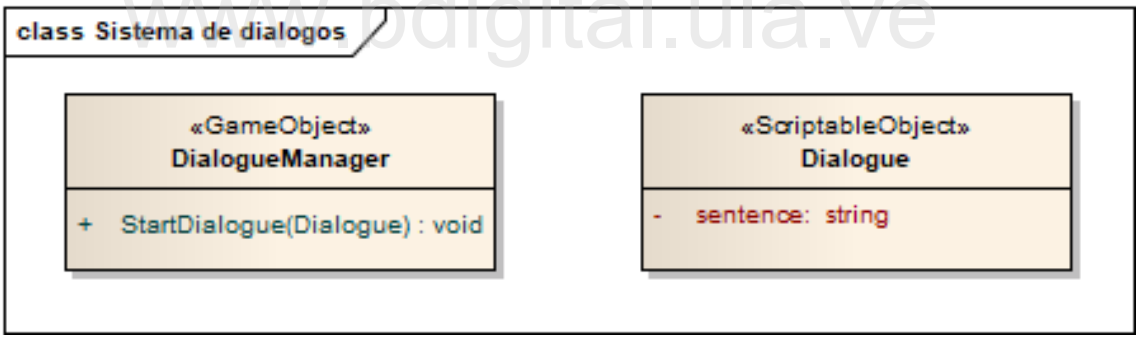

Figura 4.23: Sistema de dialogo.

una imagen para mostrar en la interfaz. Véase Figura 4.24.

4.2.10 Niveles

Una vez implementados las mayoría de los elementos necesarios para que el videojuego funcione, se procede a construir los niveles. Unificando las características de cada nivel. Fue en esta etapa de producción donde mas trabajo se realiza ya que debe

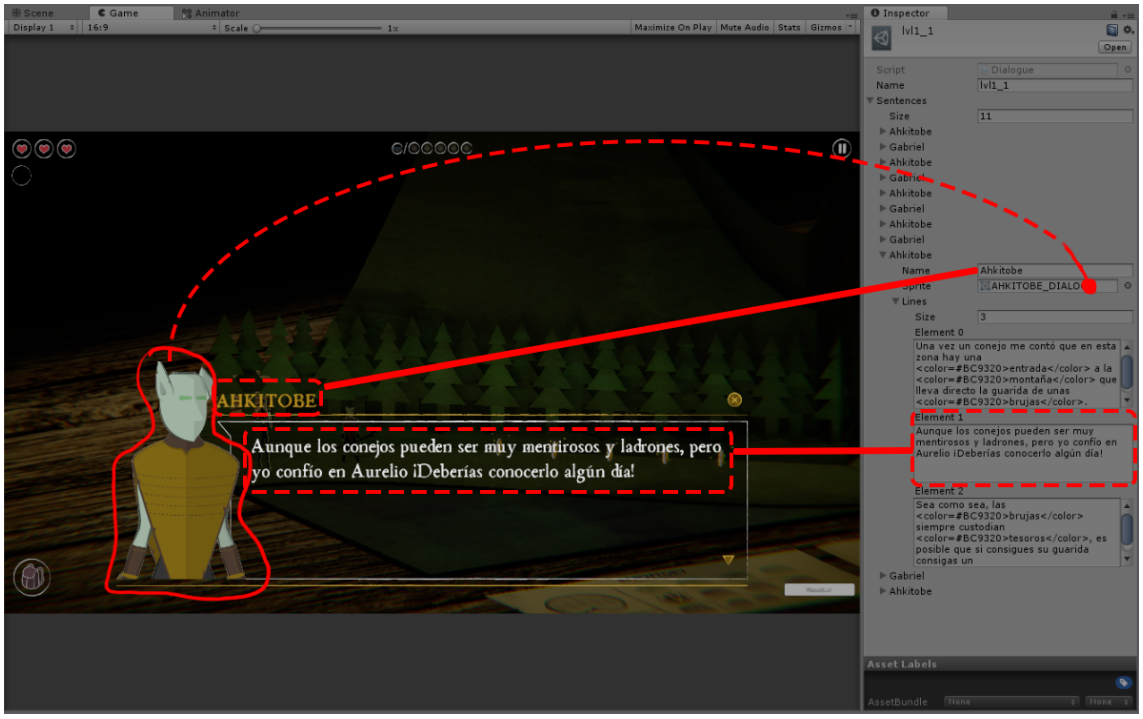

Figura 4.24: Asset de dialogo implementado en Unity.

asegurarse de que el funcionamiento de todos los elementos que componen al nivel, funcionen correctamente en conjunto. A continuación se muestra una vista de cada nivel construido en Unity. Figuras 4.25, 4.25, 4.26, 4.27, 4.28, 4.29 y 4.30.

4.3 Proceso: Publicación

4.3.1 Aplicación

Luego de implementar el nivel 6, se completa el desarrollo de los distintos aspectos del videojuego. Por eso se culmina con empaquetar el proyecto entero para generar una primera versión completa del videojuego, para que sea ejecutada en dispositivos con sistema operativo Windows y GNU/Linux. Cabe destacar que Unity permite insertar un logo que identifique a la empresa desarrolladora del videojuego y por supuesto un ícono que identifique al videojuego.

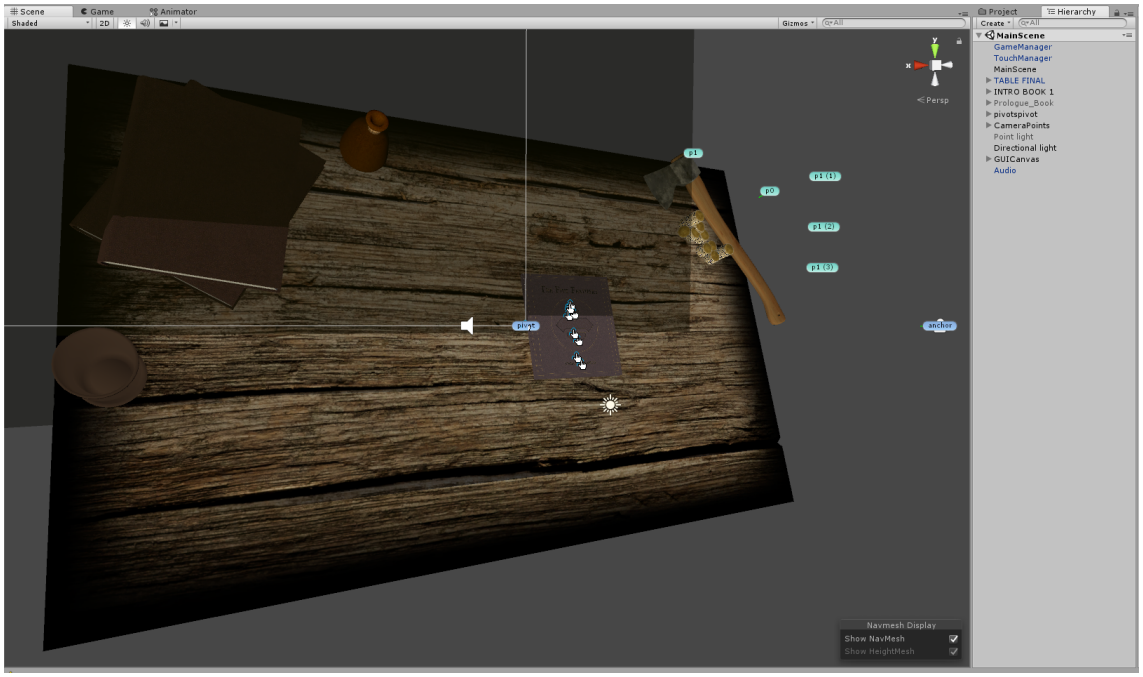

Figura 4.25: Menú principal en Unity.

Figura 4.26: Nivel "La Laguna Negra" en Unity.

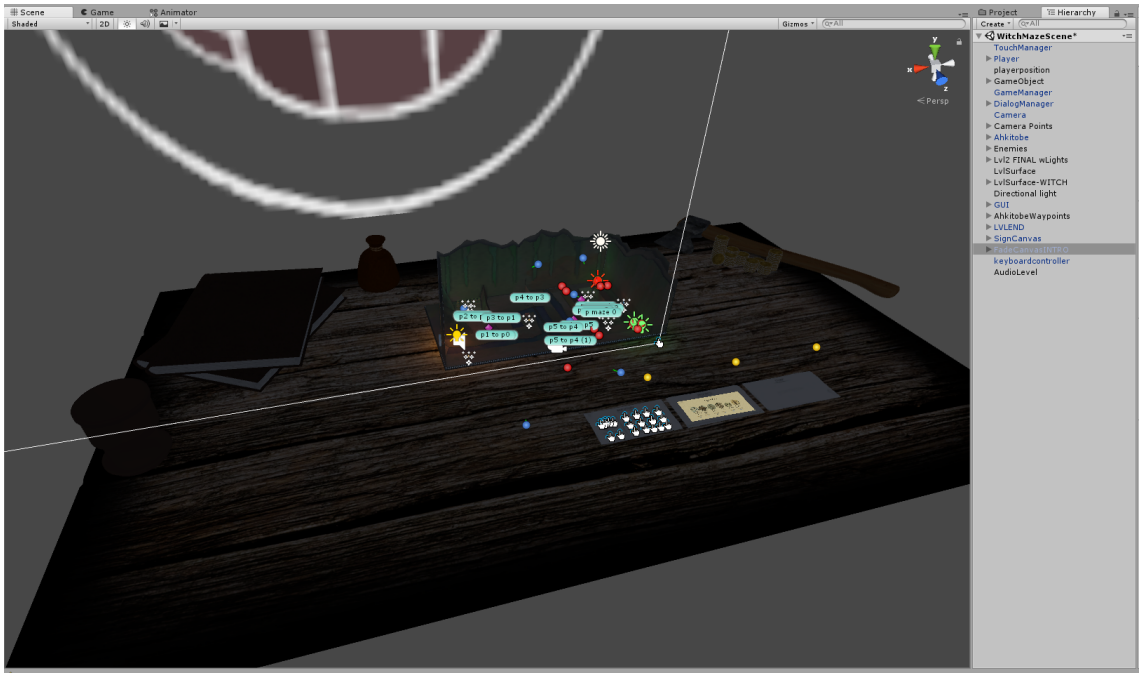

Figura 4.27: Nivel "El Laberinto de las Brujas" en Unity.

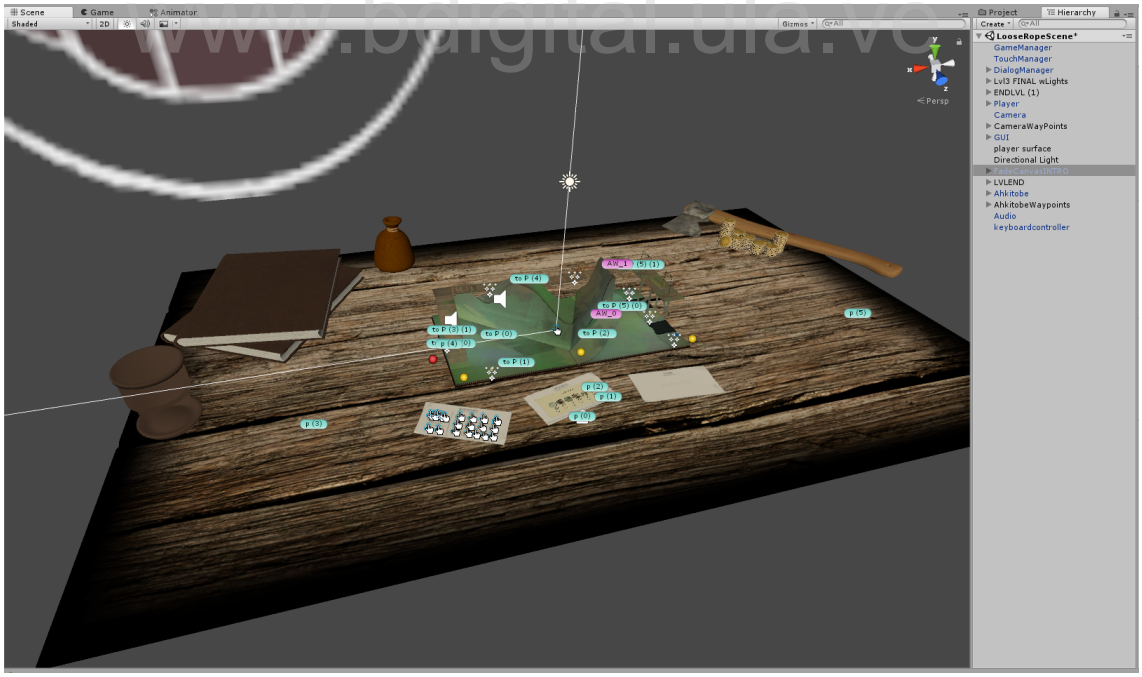

Figura 4.28: Nivel "La Cuerda Floja" en Unity.

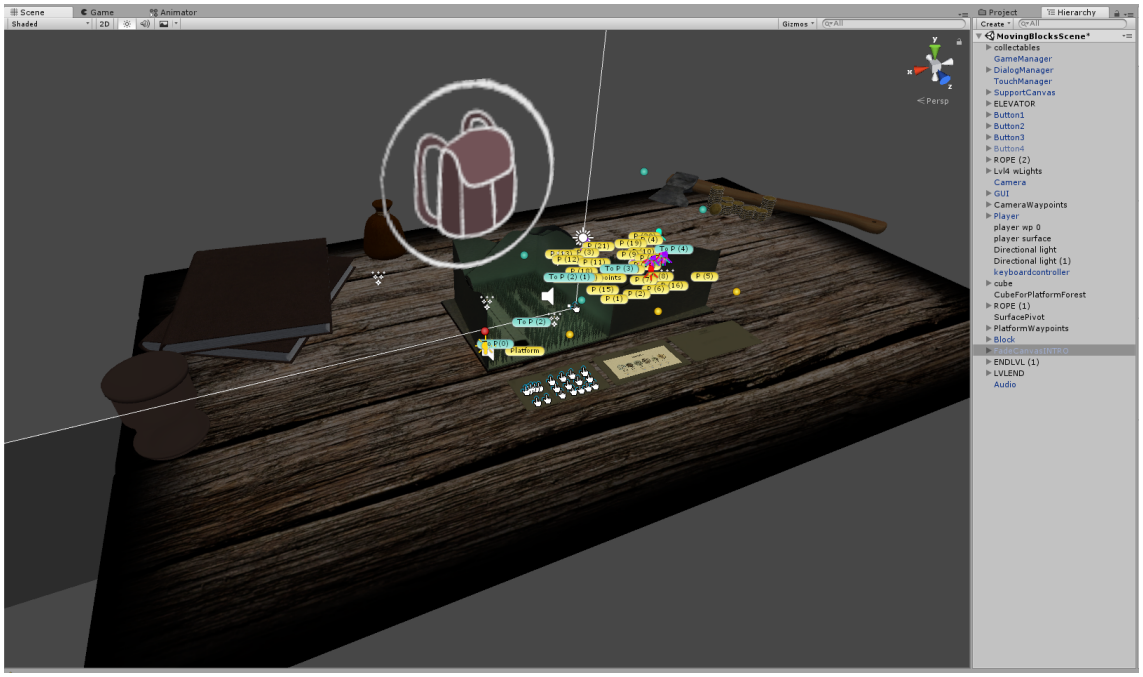

Figura 4.29: Nivel "Bloques en Movimiento" en Unity.

Figura 4.30: Nivel "La Primera Pluma" en Unity.

El nombre de la empresa de desarrollo es "Caballo Blanco", a continuación se presenta su logo, véase Figura 4.31, el cual fusiona la silueta del Caballo Blanco del escudo nacional de Venezuela con una pieza de un Caballo del juego ajedrez; lo primero representa el origen de la compañía - Venezuela - y lo segundo su objetivo - el desarrollo de juegos-.

Figura 4.31: Logo de la empresa Caballo Blanco.

El icono del juego, véase Figura 4.32, es un factor de gran interés para la publicación del videojuego, pues es el primer elemento que verá el usuario y debe ser capaz de captar su atención y decirle, dentro de lo posible, la temática del juego. Objetivo que consideramos se cumplió a cabalidad.

4.3.2 Mercadeo

Plataforma de distribución

El sitio web itch.io es un mercado abierto dirigido principalmente hacia desarrolladores de videojuegos independientes; ofrece los siguientes servicios:

- Catalogo de videojuegos gratuitos y pagos.

Figura 4.32: Icono del videojuego Las Cinco Plumas.

- Permite cargar y vender videojuegos en la plataforma sin pasar por ningún tipo de filtro, osea que no importa el estado del videojuego. Si es un prototipo, versión beta, etc.
- Pagina web personalizable. Dirigido hacia los desarrolladores.
- Distintos tipos de distribución.
- Analytics.

El modelo de negocios de itch.io se basa en ingresos compartidos abiertos y consiste en que el vendedor elije cuanto de sus ingresos son compartidos con itch.io. Puede ser desde 0% hasta 100% de las ganancias.

4.3.3 Publicación

4.3.4 Análisis y mantenimiento

No pasó mucho tiempo desde la publicación del videojuego, por lo tanto no hay suficientes datos para analizar el impacto del producto. Sin embargo el videojuego fue presentado en la IV feria ExpoRais2018 organizada por la Universidad de Los Andes, en esta feria el videojuego fue probado por la comunidad universitaria y la recepción

de la misma fue muy positiva. Los comentarios también arrojaron detalles de ciertos aspectos del videojuego, el mas destacable fue que los diálogos eran muy extensos.

Posteriormente se le hicieron modificaciones al videojuego con base a los comentarios de las personas que asistieron a la feria y probaron el videojuego.

Capítulo 5

Conclusiones

En cumplimiento de los objetivos propuestos en este proyecto de grado, que forma parte de un conjunto con el proyecto de grado realizado por Centeno (2018), se desarrolló un videojuego que posteriormente fue publicado en una plataforma de distribución, demostrando así, la eficiencia del método de desarrollo creado por los mismos proyectos.

La creación del método de desarrollo orientado a videojuegos DIY otorga al lector una guía para realizar un videojuego, desde la concepción de su idea hasta su publicación en el mercado. Este método puede extrapolarse a otras aplicaciones de software, pues a pesar de tratar aspectos propios de los videojuegos, los principios básicos se basan en el desarrollo de software orientado al usuario, al mercado, y a la empresa independiente.

El método DIY resultó ser adaptativo con respecto a las estrategias que los desarrolladores pueden tomar, quiere decir que no hay una sola forma de aplicar dicho método, ya que, los recursos que se utilizan para el desarrollo del videojuego pueden variar considerablemente. Esto exige como ya se menciona, que el método se adapte a las estrategias requeridas para el buen uso de los recursos seleccionados para la implementación de un videojuego.

Un aspecto importante que no se trató en el desarrollo del videojuego fue la

retroalimentación con los posibles usuarios a lo largo del proceso de producción del método DIY. En una industria donde surgieron más de 7000 videojuegos en el año 2017, solo en la plataforma de Steam, según el artículo de Llosa (2018), es importante concebir una expectativa del videojuego que potencie su alcance a mas personas. Existes varias estrategias para cubrir este aspecto del desarrollo, como lo muestran Henríquez (2016) aplicando en método Lean StartUp y Thin Matrix (2013); este ultimo mediante YouTube, publicó periódicamente a lo largo de dos años, vídeos que relatan el desarrollo del videojuego Equilinox, consiguiendo así, una comunidad significativa dispuesta a jugar el videojuego incluso antes de ser publicado en la plataforma de Steam.

5.1 Recomendaciones

- Promover la difusión de este proyecto de grado entre los estudiantes de Ingeniería de Sistemas para que desarrollen sus propios videojuegos.

- Desarrollar nuevos proyectos de grado basados en el método de desarrollo DIY, con el objetivo de refinar el método para que sea mas certero en el flujo de trabajo que este propone y en consecuencia desarrollar videojuegos de mas calidad.

- Trabajar en nuevos proyectos que se especialicen específicamente en áreas que conforman el desarrollo de videojuegos, ejemplos: Motores de videojuegos, inteligencia artificial enfocada a los videojuegos, realidad aumentada, realidad virtual, etc.

Bibliografía

Asbjorn (2012). Brackeys. Recuperado en 2016 de `https://www.youtube.com/user/Brackeys/`.

Asbjorn (2017). How to make a dialogue system in unity. Recuperado en 2018 de `https://www.youtube.com/watch?v=_nRzoTzeyxU`.

Beck, K. (1996). Extreme programming.

Beck, K. (1999). *Extreme Programming Explained: Embrace Change*.

Bouchard, S. (2017). TextMeshPro.

Centeno, J. (2018). Creación de un método de desarrollo de software orientado a videojuegos. fase: diseño. Reporte técnico. Proyecto de Grado, Universidad de Los Andes, escuela de Ingeniería de Sistemas.

Erich Gamma, Richard Helm, R. J. y Vlissides, J. (1997). *Elements of Reusable Object-Oriented Software*.

Gigiwoo (2011). Game financial success stories. Foro. Recuperado en 2017 de `https://forum.unity3d.com/threads/game-financial-success-stories.174152/`.

Henríquez, H. (2016). *Creación de una StartUp con un videojuego aplicando el método Lean StartUp*. Proyecto de Grado EISULA, Universidad de Los Andes, escuela de Ingeniería de Sistemas.

Jackson, M. (1975). *Principles of Program Desing*.

Li, K. (s.f.). s.t. Recuperado en 2017 de https://piazza.com/careers/dashboard?st=1&uid=ippoeeu9f7350a&ab=2&action=rcp#/company_profile/riotgames/.

Llosa, X. (2018). Saturación: Steam lanzó 7.600 juegos en 2017. *MeriStation*. Recuperado en 2018 de http://meristation.as.com/noticias/saturacion-steam-lanzo-7-600-juegos-en-2017.

Martin, J. (1993). Rapid application development.

Mikami, S. (1996). Resident evil.

Nystrom, R. (2014). *Game Programming Patterns*.

Porter, M. (2015). Why is the international games jam ludum dare so important to indie developers? Recuperado en 2017 de https://www.vice.com/en_us/article/xd74x7/discovering-why-international-games-jam-ludum-dare-is-so-important-to-indie-devel

Pyro Studios (1998). Commandos.

Simonov, V. (2016). Touchscript. Recuperado en 2017 de https://github.com/TouchScript/TouchScript.

Simonov, V. (2017). The journey of a touch point. Recuperado en 2017 de https://github.com/TouchScript/TouchScript/wiki/The-Journey-of-a-Touch-Point.

Takeuchi, H. y Nonaka, I. (1986). The new new product development game. Recuperado en 2018 de https://hbr.org/1986/01/the-new-new-product-development-game.

Thin Matrix (2013). Equilinox. Recuperado en 2018 de https://www.youtube.com/user/ThinMatrix.