

UNIVERSIDAD NACIONAL DE TUCUMÁN

Facultad de Ciencias Exactas y Tecnología
Departamento de Electricidad, Electrónica y Computación



TRABAJO DE GRADUACIÓN

Diseño y desarrollo de un videojuego interactivo aplicando Realidad Aumentada

Alumno: Avellaneda, Gabriel Darío

CX: 14-0907-9

Tutor: Albaca Paraván, Carlos

Carrera: Ingeniería en Computación

Junio 2016

Índice

Índice	1
Motivación.....	4
Objetivos	4
Sección 1: Videojuego Para PC.....	5
Capítulo 1: Introducción	6
1.1. Concepto de Videojuego.....	6
1.2. Género de Videojuegos	6
1.2.1. Juegos de Acción	6
1.2.2. Aventuras Gráficas.....	6
1.2.3. Survival Horror	7
1.2.4. RPG (Role Playing Games)	8
1.2.5. Estrategia (RTS).....	8
1.2.6. Estrategia (Turn-Based)	8
1.2.7. Simuladores	9
1.2.8. Deportivos	9
1.2.9. Lucha	10
1.2.10. God Games	10
1.2.11. Otros Tipos de Videojuegos	11
1.3. Partes de un Videojuego.....	12
1.3.1. Motor	12
1.3.2. Assets	14
1.3.3. Scripts	14
Capítulo 2: Diseño de un Videojuego	16
2.1. Introducción	16
2.2. Arquitectura Software de Videojuegos.....	18
2.2.1 Arquitectura Software General	18
2.2.2. Arquitecturas Software para las Entidades de Juego.....	19
2.2.3. Jerarquías de Clases	20
2.2.4. Arquitectura Basada en Componentes.....	21
2.3. Metodologías para el Desarrollo de Videojuegos	22
2.4 Metodología Ágil para el Desarrollo de este Proyecto: SUM	23
2.4.1 Introducción a la Metodología	23
2.4.2 Roles	24
2.4.3 Ciclo de Vida	25
2.4.4 Concepto	26
2.4.5 Planificación	32
2.4.6 Elaboración	34
2.4.7 Beta	35
2.4.8 Cierre	36
2.4.9 Gestión de Riesgos	36
2.4.10 Guías.....	36
Capítulo 3: Adaptación de la Metodología al Proyecto.....	38
3.1 Fase 1: Concepto	38
3.1.1 Herramientas.....	38
3.1.2 Elementos de Diseño	40

3.2 Fase 2: Planificación.....	47
3.2.1 Planificación Administrativa.....	47
3.2.2 Especificación del Videojuego.....	47
3.3 Fase 3: Elaboración	48
3.3.1 Instalación de Unity y Vuforia	48
3.3.2 Interconexión de Escenas o Niveles	50
3.3.3 Personaje	63
3.3.4 Enemigos	70
3.3.5 Personaje vs Enemigos.....	75
3.3.6 Personaje y la UI	79
3.3.7 Items	82
3.3.8 Escenarios	90
3.4 Fase 4: Beta	91
3.5 Fase 5: Cierre	92
Sección 2: Aplicación para Android	93
Capítulo 4: Realidad Aumentada.....	94
4.1 Introducción	94
4.2 Idea de la Aplicación.....	97
Capítulo 5: Especificación de Requisitos de Software	99
5.1 Análisis de Requisitos del Sistema	99
5.2 Identificación de los Usuarios Participantes.....	99
5.3 Catálogo de Requisitos del Sistema	99
5.4 Objetivos y Alcance del Sistema.....	99
5.5 Definiciones, Acrónimos y Abreviaturas.....	100
5.6 Descripción General	100
5.7 Requisitos Funcionales.....	101
5.8 Suposiciones y Dependencias.....	101
5.9 Requisitos de Usuario y Tecnológicos.....	102
5.10 Requisitos de Interfaces Externas	102
5.11 Requisitos de Rendimiento	102
5.12 Requisitos de Desarrollo.....	103
5.13 Restricciones de Diseño	103
Capítulo 6: Selección del Modelo de Ciclo de Vida y Gestión del Proyecto	104
6.1 Introducción	104
6.2 Selección de Metodología de Desarrollo de Software	104
6.3 Selección de un Modelo de Ciclo de Vida.....	106
Capítulo 7: Gestión del Proyecto	108
7.1 Introducción	108
7.2 Procedimiento	108
7.3 Planificación del Proyecto.....	113
7.3.1 Introducción.....	113
7.3.2 Plan del Desarrollo	113
Capítulo 8: Modelización y Diseño	115
8.1 Actores.....	115
8.2 Diagrama de Contexto	115
8.3 Listado de Casos de Uso.....	115
8.4 Diagrama de Casos de Uso.....	115
8.5 Descripción Textual de los Casos de Uso	116

8.6 Diagrama de Actividad.....	117
8.7 Diagrama de Secuencia.....	118
8.8 Identificación de Roles.....	119
8.9 Guiones y Escenarios: Diagrama de Transición de Escenarios	119
8.10 Tabla de Transición de Escenarios.....	120
8.11 Descripción de Escenarios y Objetos de Escenarios.....	120
Capítulo 9: Implementación y Pruebas.....	123
9.1 Herramienta Principal para el Desarrollo de la Aplicación	123
9.1.1 Introducción.....	123
9.1.2 Image Target.....	125
9.1.3 Subir y Manejar los <i>Image Targets</i>	126
9.1.4 Atributos de un Image Target Ideal	127
9.1.5. Características y Puntuaciones de los Image Target	127
9.1.6 Vuforia Target Manager	128
9.1.7 Arquitectura de la Aplicación para Android Usando Vuforia.....	129
9.2 Otras Herramientas.....	133
9.2.1 Android SDK	133
9.2.2 Java JDK.....	133
9.2.3 OpenGL Es	134
9.3 Desarrollo de la Aplicación	134
Observaciones	144
Conclusiones	147
Bibliografía	148

Motivación

Los videojuegos son, hoy en día, una de las principales industrias del arte y entretenimiento, y facturan más que el cine y la música juntos. El crecimiento del desarrollo *indie* o independiente en los últimos años ha crecido de manera exponencial, principalmente debido a los nuevos métodos de distribución en línea y herramientas de desarrollo. Ser desarrollador independiente de videojuegos significa no tener ataduras con lo establecido y luchar por ideas innovadoras que además pueden terminar formando parte de la gran industria. Con esfuerzo y dificultades, los independientes siguen ampliando las fronteras del videojuego.

La realidad aumentada, aunque lleva varios años en el mercado, se mantuvo más bien en un segundo plano, utilizada por algunos juegos o campañas publicitarias. Entre el año 2006 y 2008, gracias al mundo de los videojuegos y a la mejora de las capacidades computacionales de ordenadores y tarjetas gráficas, resultó posible confeccionar experiencias de realidad aumentada de una gran calidad. Aunque todavía los dispositivos que existen en el mercado son un tanto toscos y la experiencia visual es muy mejorable, ya se intuyen numerosas aplicaciones y negocios en nuevos ámbitos como la formación profesional, la educación y el ocio digital.

Objetivos

- Objetivo General: Realizar un videojuego para computadora de escritorio, logrando al mismo tiempo, interacción con el usuario mediante el uso de un dispositivo móvil y realidad aumentada.
- Objetivos Específicos:
 - Entender qué es un videojuego y sus diferentes tipos y características, las partes de las que se compone y cómo es usada cada una de ellas.
 - Conocer el proceso de diseño de un videojuego.
 - Aplicar en el juego conceptos aprendidos en la carrera, tanto los vistos en asignaturas del ciclo básico (como física y álgebra) como los del ciclo superior (lógica, estructuras de datos, programación orientada a objetos).
 - Inteligencia artificial. Creación de objetos capaces de tener comportamiento propio (atacar al jugador, alejarse de él, moverse aleatoriamente o seguir alguna ruta específica).
 - Utilizar las librerías de realidad aumentada del *SDK* (kit de desarrollo de software) Vuforia para trabajar con realidad aumentada en la aplicación del dispositivo móvil.

Sección 1

Videojuego para PC

Capítulo 1

Introducción

1.1. Concepto de Videojuego

Un videojuego (o juego de video) es un juego electrónico en el que, por medio de un controlador, una o más personas interactúan con un dispositivo dotado de imágenes de video y sonido. Este dispositivo electrónico, conocido genéricamente como plataforma, puede ser una computadora, una máquina *arcade*, una videoconsola o un dispositivo portátil (un teléfono móvil, por ejemplo).

1.2. Género de videojuegos

El género de videojuego es una forma de clasificar un videojuego en función fundamentalmente de su mecánica de juego. La mecánica de juego es una regla o conjunto de reglas cuyo objetivo consiste en obtener una serie de resultados coherentes en el seno de un juego.

1.2.1. Juegos de Acción

El objetivo al diseñar un juego de acción es mantener al jugador moviéndose y ocupado todo el tiempo. Muy a menudo, los videojuegos de acción usan la violencia como su principal característica de interacción. Más específicamente, el combate con armas de fuego o cuerpo a cuerpo. Ejemplo: Assassin's Creed, mostrado en la Fig. 1.1.



Fig. 1.1 – Gameplay del juego Assassin's Creed

1.2.2. Aventuras Gráficas

La dinámica de este tipo de juego consiste en ir avanzando por el mismo a través de la resolución de diversos rompecabezas, planteados como

situaciones que se suceden en la historia, interactuando con personajes y objetos a través de un menú de acciones o interfaz similar. Ejemplo: Monkey Island, mostrado en la Fig. 1.2.



Fig. 1.2 – Gameplay del juego Monkey Island

1.2.3. Survival Horror

Se denomina *survival horror* o videojuego de terror al subgénero de videojuegos enfocados principalmente a atemorizar al jugador, con lo que se pretende provocar inquietud, desasosiego o incluso miedo. Estos videojuegos, encuadrados dentro del género acción-aventura, hacen uso de los temas, recursos estéticos y narrativos propios del cine y la novela de terror, potenciándolos a través de la capacidad inmersiva que caracteriza al medio. Otros elementos definitorios del género son la tendencia a minimizar el combate, limitando el acceso a armas o incluso eliminando totalmente la posibilidad de defenderse (forzando al jugador a optar por el sigilo o la huida), y la presencia periódica de determinados enigmas o rompecabezas de dificultad variable que el jugador deberá resolver para poder continuar. Ejemplo: Resident Evil, mostrado en la Fig. 1.3.



Fig. 1.3 – Gameplay del juego Resident Evil

1.2.4. RPG (Role Playing Games)

Usa elementos de los juegos de rol tradicionales (pero estos últimos no son juegos electrónicos, se juega a ellos con dados, lápices y hojas de papel). Se centran sobre todo en la historia del personaje y tienen buena dosis de combate. Ejemplo: Final Fantasy, mostrado en la Fig. 1.4.



Fig. 1.4 – Gameplay del juego Final Fantasy

1.2.5. Estrategia (RTS)

Se basan en el equilibrio de todos los elementos del juego. El jugador controla todo lo que tiene a su alcance. Ejemplo: Age of Empires, mostrado en la Fig. 1.5.



Fig. 1.5 – Gameplay del juego Age of Empires

1.2.6. Estrategia (Turn-based)

Idéntico al anterior, pero las acciones no son en tiempo real sino por turnos. Ejemplo: Heroes of Might and Magic, mostrado en la Fig. 1.6.



Fig. 1.6 – *Gameplay* del juego *Heroes of Might and Magic*

1.2.7. Simuladores

El objetivo es emular lo mejor posible el uso de maquinaria compleja de la vida real. Ejemplo: Silent Hunter 4, mostrado en la Fig. 1.7.



Fig. 1.7 – *Gameplay* del juego *Silent Hunter 4*

1.2.8. Deportivos

Intentan emular lo más realista posible a los deportes. Ejemplo: Pro Evolution Soccer, mostrado en la Fig. 1.8.



Fig. 1.8 – Gameplay del juego Pro Evolution Soccer

1.2.9. Lucha

Los videojuegos de lucha, como indica su nombre, recrean combates entre personajes controlados tanto por un jugador como por la computadora. El jugador normalmente ve a los combatientes desde una perspectiva lateral, como si se tratase de un espectador aunque también hay excepciones que manejan entornos en 3D y primera persona. Este tipo de videojuegos ponen especial énfasis en las artes marciales, reales o ficticias (generalmente imposibles de imitar), u otros tipos de enfrentamientos sin armas como el boxeo o la lucha libre. Otros videojuegos permiten también usar armas blancas (como espadas) o ataques a distancia, normalmente de carácter mágico o etéreo. Ejemplo: Street Fighter, mostrado en la Fig. 1.9.



Fig. 1.9 – Gameplay del juego Street Fighter

1.2.10. God Games

Son juegos abiertos, sin una finalidad clara. El jugador juega sin un objetivo definido. No hay ninguna acción "errónea". Ejemplo: The Sims, mostrado en la Fig. 1.10.



Fig. 1.10 – Gameplay del juego The Sims

1.2.11. Otros tipos de videojuegos

- Juegos casuales: cualquiera donde se espera jugar poco tiempo seguido. Ejemplo: Pacman, mostrado en la Fig. 1.11.

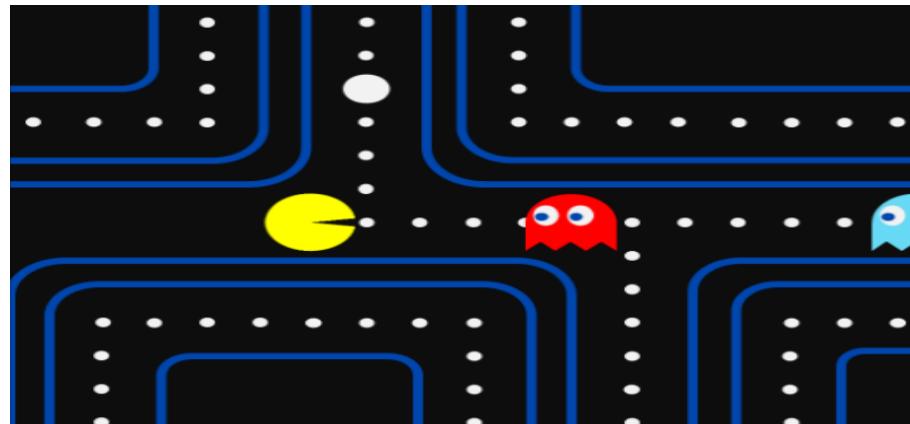


Fig. 1.11 – Gameplay del juego Pacman

- Juegos Educativos: diseñados con el fin de enseñar algo alguien. Ejemplo: Brain Training, mostrado en la Fig. 1.12.

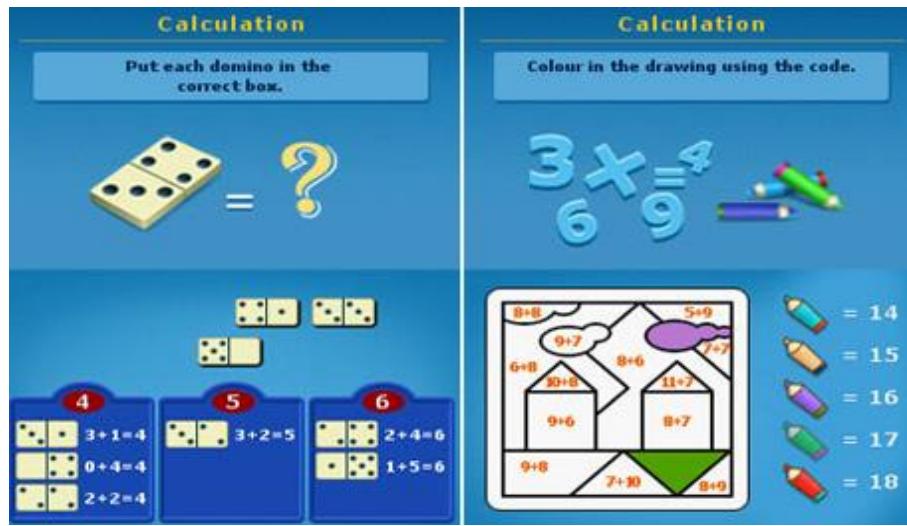


Fig. 1.12 – Gameplay del juego Brain Training

- Juegos Online: cualquier juego de los anteriores, pero su núcleo está en un servidor online. Ejemplo: League of Legends, mostrado en la Fig. 1.13.



Fig. 1.13 – Gameplay del juego League of Legends

1.3. Partes de un videojuego

Básicamente, todo videojuego consta de tres partes:

- Un motor.
- Assets (texturas, mallas, sonidos, animaciones, etc).
- Scripts.

1.3.1. Motor

Un motor muy completo tiene miles de horas de programación y son tan complejos que algunos se venden por cientos de miles de dólares (por ejemplo: CryEngine, UnrealEngine, Gamebryo, Torque, OGRE, Unity).

El motor se puede separar en módulos como se puede apreciar en la Fig. 1.14.



Fig. 1.14 – Un esquema para representar el motor y sus módulos

- Motor de Renderizado (*Render*): La parte más grande de un videojuego es su motor de renderizado (2D y 3D) que utiliza, en general, o bien DirectX o bien OpenGL. Estas son *APIs* que permiten interactuar directa o indirectamente con los componentes del dispositivo sobre el que se programa (*CPU*, *GPUs*).
- Motor de Física: El motor de física es un sistema matemático que implementa las reglas físicas que regulan el mundo en el que se basa el juego. En un videojuego son muy importantes las colisiones y los distintos métodos para detectarlas. Las leyes físicas pueden ser reales, modificadas o inventadas.
- Motor de Inteligencia Artificial: Es la parte que se encarga de las decisiones de comportamiento de los personajes del juego que no son controlados por el usuario. Algunas Técnicas:
 - Máquina de estados.
 - *Behaviour Tree* (árbol de comportamiento).
 - Búsqueda de caminos.
- Motor de animaciones: Las animaciones, al igual que los modelos 3D son creadas por un artista y exportadas dentro del motor. El motor debe ser capaz de pintar la secuencia de una animación cuadro a cuadro, saber cuándo comenzar, terminar o repetir una animación y también hacer una interpolación entre animaciones.
- Sistemas de partículas: Un sistema de partículas se utiliza para simular fuego, viento, humo, explosiones, chispas, rayos y cualquier otro comportamiento caótico de un objeto. Básicamente se trata de una

textura 2D orientada a cámara que se crea con tamaño variable y se anima de forma caótica hasta que toma una forma similar a lo simulado.

- Redes: Si un videojuego está pensado para ser jugado por más de un jugador, necesita una interacción entre ambos sistemas. En internet se necesita un sistema cliente-servidor, y por lo tanto un motor de mensajería implementado tanto en el cliente como en el servidor.
- Cámaras y Control: Cada tipo de juego tiene diferentes cámaras que son las que muestran lo que pinta el motor 3D. Hay cámaras en primera persona, tercera persona, fijas, cenitales, etc. También hay una parte del motor que nos permite controlar los dispositivos de introducción de datos, como teclado, *mouse*, *gamepad* y *joysticks* en general.
- Sonidos y música: Los sonidos, la música y las voces de un videojuego son pistas de audio que se importan dentro del juego y que deben ser lanzadas en algún momento oportuno, su volumen también debe ser ajustado (sube si el personaje se acerca, baja si se aleja, por ejemplo). El sonido puede ser mono o estéreo. También puede ser 2D o 3D (envolvente).
- Interfaz con el usuario (*GUI*): Es una interfaz con la que el usuario interacciona independientemente del mundo 3D. Se ubica por delante del mundo, “en la pantalla”. Son representaciones 2D en un juego 3D. Por ejemplo: un menú o la barra de vida.

1.3.2. Assets

Son los distintos elementos que realizan los artistas del equipo de trabajo:

- Mallas: son objetos 3D expresados como secuencia de puntos 3D.
- Texturas: son imágenes 2D que se pintan sobre los objetos 3D.
- Animaciones: son secuencias de puntos 3D que representan el movimiento de la malla.
- Sonidos: son los efectos especiales que responden a un evento o un lugar específico.

Cada objeto 3D de un videojuego está modelado con un programa especial para ello (Maya, 3D Studio Max, etc). Un modelo 3D es un grupo de puntos en el espacio unidos por líneas, conformando una “malla”. Esta malla está compuesta por triángulos. Mientras más triángulos tenga un modelo, más costará pintarlo en pantalla, y será más carga para el *CPU*. Además de la “malla”, un objeto 3D está compuesto por un material que contiene la textura del modelo, y otra serie de componentes como el mapa de iluminación, normales, etc. Las animaciones son copias de la malla en distintas posiciones emulando un movimiento.

1.3.3. Scripts

Lenguaje de programación de alto nivel, de codificación simple, con pocas funciones (todas ellas relativas al motor del juego). Utilizado para codificar los eventos del juego. Los *scripts* se pueden usar para hacer todo o casi todo en

un videojuego. Se pueden crear objetos dinámicamente, destruirlos, añadirles componentes, quitárselos o crear comportamientos (tanto para el objeto, como para cada uno de sus componentes). Los *scripts* son la mejor herramienta que los programadores poseen para hacer de un juego una experiencia divertida.

Capítulo 2

Diseño de un Videojuego

2.1. Introducción

Desde la aparición de los primeros videojuegos hasta el día de hoy, los juegos y su proceso de desarrollo han recorrido un largo camino. Los primeros videojuegos no mostraban más que unos puntos, líneas o figuras geométricas en la pantalla y se controlaban mediante el uso de simples potenciómetros o pulsadores. Sin embargo, hoy en día, muestran gráficos 3D extremadamente realistas y, los modos de control, son tan abundantes como diversos, yendo desde teclado o mandos hasta reconocimiento de movimiento mediante el uso de cámaras.

El aumento de la popularidad de los videojuegos ha provocado un incremento en la exigencia y, por tanto, el número de personas involucradas en el desarrollo ha ido creciendo con los años. Antiguamente uno o dos programadores se encargaban de todo el proceso de desarrollo mientras que hoy en día, las grandes producciones, involucran a una gran cantidad de gente que cumple diferentes roles dentro del desarrollo. Debido a este fenómeno de especialización de equipo, ahora hay diferentes roles dentro de un desarrollo, donde ya no existen solamente programadores que se encarguen de todas las tareas (al menos en desarrollos medios y grandes), sino que ahora se tiene un equipo multidisciplinar de personas que provienen de áreas muy diferentes. Además de los programadores, que implementan la funcionalidad del juego, y los artistas, que generan los recursos gráficos, se puede destacar el rol de diseñador. Los diseñadores son los encargados del aspecto creativo del desarrollo y se dedican a idear y desarrollar la jugabilidad del juego, sus mecánicas, la historia de trasfondo con sus posibles acontecimientos y diálogos, los tipos de personajes, cómo éstos deben comportarse, etc. Al incremento de la exigencia y la dificultad de gestionar un grupo multidisciplinar se le suma el hecho de que el diseño de mecánicas de juego requiere de un desarrollo rápido de funcionalidad para probar si las ideas diseñadas “funcionan” o debe replantearse el diseño del juego.

Todas estas cuestiones provocan que las metodologías clásicas de desarrollo software no sirvan y se deba optar por metodologías actuales, más flexibles y que asimilen relativamente bien cambios en la especificación, primando la velocidad de desarrollo frente a la sobreingeniería. La sobreingeniería, a grandes rasgos, es el hecho de agregar diversas opciones o funcionalidades a un producto, con la finalidad de que se utilicen en un futuro. En el mundo de la programación, la sobreingeniería implica tener un montón de código, que solo hace más pesado el programa, y que además enreda a los desarrolladores, pero que está ahí porque quizá en algún momento se llegue a utilizar, cosa que rara vez pasa.

El desarrollo de mecánicas o características de un juego suele ser un proceso iterativo donde las ideas que van surgiendo se documentan, se implementan, se evalúan y, en función de esa evaluación, se descartan, se aceptan o se

refinan, volviendo a pasar de nuevo por todas las fases anteriores. Con el paso de los años la tecnología usada para soportar este tipo de metodologías ha ido evolucionando y, hoy en día, prácticamente todos los juegos del mercado usan una arquitectura basada en componentes para implementar las diferentes entidades u objetos del juego. Este tipo de tecnología es muy flexible, permitiendo todo tipo de modificaciones en las especificaciones y premiando la reusabilidad de código. Estas arquitecturas están basadas en composición en lugar de la herencia, la funcionalidad de las entidades del juego se encuentra fraccionada y repartida por varios componentes software. Esto aporta velocidad a la hora de formar entidades nuevas o modificar ya creadas y ahorra grandes tiempos de compilación, pero dificulta también la comprensión a alto nivel de qué es cada entidad, ya que se limita a un conjunto de componentes.

El desarrollo de videojuegos es una tarea llevada a cabo por un equipo compuesto por perfiles de campos muy diferentes. Los roles que se necesitan varían de proyecto a proyecto. De todas maneras, se puede considerar que hay tres roles básicos que aparecen siempre en el proceso de producción: artista, programador y diseñador.

Los artistas son los responsables de la creación de todos los recursos gráficos del juego. Su trabajo es crear y texturizar los modelos de los personajes con sus correspondientes animaciones, elementos del mundo, niveles y todo tipo de contenido 3D así como realizar las imágenes usadas en el juego para los menús y la interfaz de usuario.

La responsabilidad del programador es crear todo el código necesario para hacer que todas las ideas descritas por los diseñadores se hagan realidad y funcionen usando los recursos que han generado los artistas. Los programadores están involucrados en todos los aspectos del juego: gráficos, animaciones, lógica de juego, *gameplay* (jugabilidad), interfaz de usuario, física, sonido, *scripts*, etc. Los programadores suelen ser ingenieros informáticos y, debido a que están implicados en todos los aspectos del juego y son los que saben acerca de las limitaciones técnicas existentes, deben evaluar los riesgos técnicos del diseño del juego y acordar con los diseñadores qué cosas se pueden hacer y qué cosas no son viables y deben ser rediseñadas. De la misma manera, son los encargados de poner restricciones técnicas a los recursos que generan los artistas y que deben ser incluidos en el juego.

Los diseñadores son la cabeza pensante del proyecto, son los que deciden el rumbo que éste debe llevar y cual es el resultado que se desea obtener. Están involucrados desde el principio al fin del proyecto, siendo los encargados de desarrollar la idea inicial del juego y pensar en hasta el último detalle del producto que se quiere obtener, controlando que realmente el rumbo sea el adecuado. Entre sus responsabilidades, aparte de decidir en qué va a consistir el juego, está establecer cuáles van a ser las mecánicas principales de éste, las reglas, cómo va a interaccionar el jugador, qué respuesta visual y sonora va a obtener, cómo van a ser los personajes, cuál va a ser la historia del juego, guión, diálogos o diseñar los diferentes niveles que se tendrán. Una de las tareas más importantes del diseñador es saber

transmitir exactamente qué es lo que quiere de cada miembro del equipo y de plasmar perfectamente cuáles son los requisitos.

2.2. Arquitectura Software de Videojuegos

2.2.1 Arquitectura Software General

Los videojuegos son aplicaciones de tiempo real grandes y complejas que usan una gran cantidad de recursos multimedia, deben proporcionar una experiencia inmersiva y, lo más importante, deben de ser divertidos. Esta búsqueda de la diversión es un proceso iterativo de prueba y error que va desde las primeras etapas de prototipado a las últimas en las que se balancean las mecánicas de juego. Desde el punto de vista de la ingeniería del software, esta situación es muy poco deseable ya que las especificaciones van evolucionando hasta el día en que se entrega el producto final. Aunque hay muchos tipos y complejas descripciones de arquitecturas de videojuego, en su forma más simple se pueden considerar dos grandes bloques:

- *Front-end*: Responsable de retroalimentar la experiencia del jugador procesando los comandos de entrada del usuario (teclado, ratón, etc.) y presentando el juego (motor). Es decir, cargar los recursos externos, procesar la entrada de usuario y presentarle los resultados.
- *Back-end*: Encargado de la lógica de juego que especifica las reglas, determina la dinámica de interacción con los objetos, proporciona comportamientos a los personajes no jugadores (*NPC*), las interacciones entre ellos, etc. Es decir, personalizar el juego.

La Figura 2.1 muestra un esquema simplificado de la arquitectura de videojuegos:

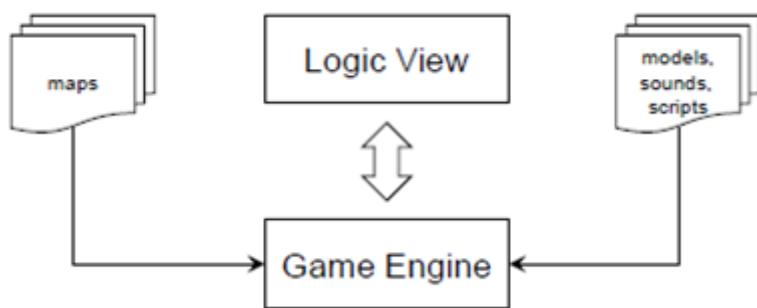


Fig. 2.1 – Vista de una arquitectura de videojuegos en su forma más simple

El funcionamiento de un videojuego puede reducirse a un bucle principal donde en cada iteración del bucle se hacen tres acciones:

1. El motor de juego recoge la entrada del usuario y le comunica a la vista lógica los comandos que ha decidido hacer el jugador.
2. La lógica de juego, en función de los comandos de entrada del usuario, del estado actual del juego y de las acciones realizadas por los *NPCs*,

toma decisiones acerca de qué es lo que se debe presentar al usuario, comunicándose con el motor del juego.

3. El motor de juego debe transmitir el nuevo estado del juego al usuario haciendo uso de recursos gráficos y sonoros.

Pong, que fue el primer videojuego comercial, ya tenía este bucle donde el usuario se comunicaba con un potenciómetro. El sistema iba actualizándose calculando la nueva posición de la pelota y el de las dos paletas y, finalmente, presentaba por el televisor los resultados.

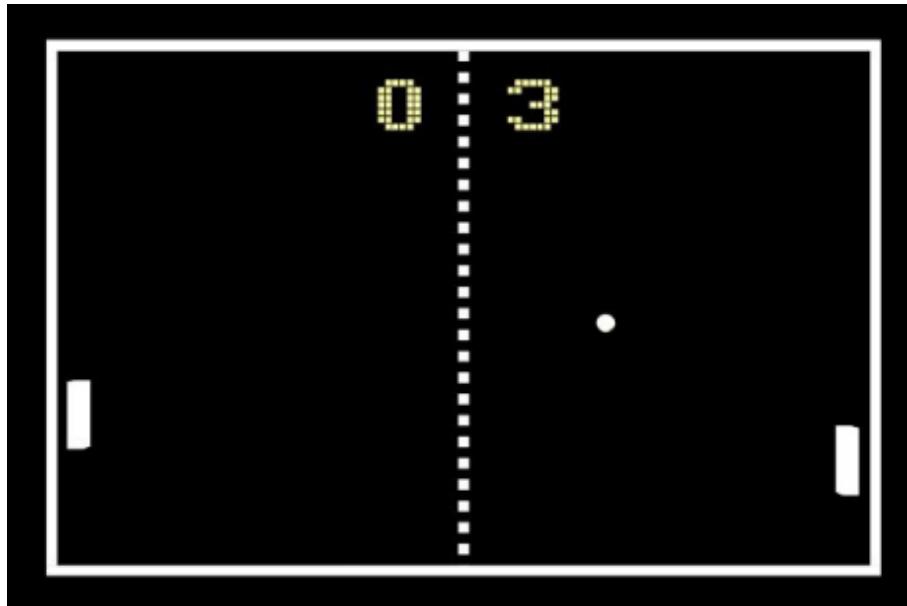


Fig. 2.2 – Juego Pong

Los componentes que conforman el *front-end* suelen estar especificados desde las primeras fases del desarrollo. La parte del *back-end* (lógica de juego), es la que controla el flujo del juego y la que toma las decisiones de qué es lo que se debe hacer en cada instante, comunicándose con el motor de juego. La lógica del juego mantiene siempre un estado del juego y, prácticamente siempre, este estado suele gestionarse mediante el uso de un conjunto de entidades u objetos de juego, que representan todo tipo de elementos del juego.

2.2.2. Arquitecturas Software para las Entidades de Juego

Las entidades de juego son piezas de lógica autocontenido, o piezas de contenido interactivo, que pueden llevar a cabo diferentes tareas como renderizarse a sí mismas, encontrar o seguir caminos, tomar decisiones, etc. Ejemplos obvios son los *NPCs* (personajes no jugables), pero también, son entidades elementos no tan obvios como las secuencias de cámara, puntos de ruta o disparadores (*triggers*) que controlan la historia del juego por ejemplo.

Además de las entidades en si, es necesario un sistema de gestión de éstas para saber cuáles están activas, cuáles se deben crear o destruir, etc. Este

módulo forma parte del núcleo de un juego y, debido a su tamaño y complejidad, requiere de esfuerzo en el desarrollo. Ejemplos:

- El juego Half-Life que data de 1999 tiene más de 65,000 líneas de código en este módulo (sin contabilizar líneas vacías ni comentarios).
- El juego Far Cry de 2004 excede las 95,000 líneas de C/C++ incluso teniendo en cuenta que la mayoría del módulo estaba escrito en LUA.
- En 2006, Gears of War tenía 250,000 líneas de C++ y scripts.

Debido al tamaño, la complejidad y a su naturaleza de especificación cambiante, se hace necesaria una arquitectura altamente flexible y extensible. Al principio de los años 2000, cuando los lenguajes orientados a objetos se fueron imponiendo en el desarrollo de videojuegos, las entidades de juego se organizaban en jerarquías basadas en herencia donde, por ejemplo, jugadores y enemigos son personajes y tanto los personajes como los objetos son entidades. Este tipo de estructura se demostró rápidamente que era rígida y difícil de mantener (se explicará a continuación).

2.2.3. Jerarquías de Clases

Durante el diseño de un juego es habitual categorizar las entidades mediante el uso de una ontología que modela la relación es-un entre todas ellas. Este conocimiento puede ser usado en la fase de desarrollo mediante herencia en el código del juego por lo que no sorprende que la gestión de entidades tradicional se basase en herencia donde las entidades se implementan como subclases que derivan de la misma clase base. En función de la complejidad del juego, las jerarquías pueden ser bastante profundas, con clases abstractas intermedias que agregan funcionalidad común para incrementar la reusabilidad del código. Como ejemplo, la Figura 2.3 muestra una jerarquía parcial hipotética, donde los nodos hoja representan clases concretas mientras que los nodos internos son clases que pueden no haber sido concebidas durante la fase de diseño pero creadas para la reutilización de código.

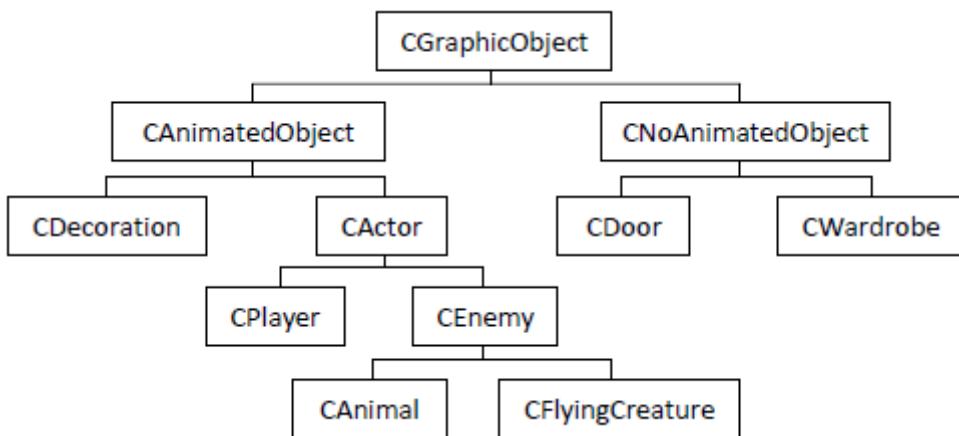


Fig. 2.3 – Árbol de herencia tradicional

Esta jerarquía trae varias decisiones implícitas debido a la manera en que se parten las clases cuando se desciende por la jerarquía. Estas decisiones pueden volverse problemáticas a la larga, sobre todo en un entorno tan cambiante como el desarrollo de videojuegos.

Supongamos que a la jerarquía de la Figura 2.3 se quisiera añadir una clase CHumanEnemy, una entidad con las mismas habilidades que el jugador pero controlada por una inteligencia artificial. La nueva entidad tendría por tanto características de CEnemy y de CPlayer por lo que la solución que parece obvia sería que heredase de ambas. Sin embargo, en primer lugar, la herencia múltiple no es soportada por todos los lenguajes orientados a objetos y, en segundo, aunque fuese soportada provocaría herencia en diamante, donde CHumanEnemy hereda dos veces de CActor, lo cual genera ambigüedad, confusión y una complejidad no deseable. La solución por tanto sería usar herencia simple, de manera que CHumanEntity sería subclase de, por ejemplo, CEnemy. Para evitar la duplicación de características como conducir vehículos en CPlayer y CHumanEntity, éstas pasan a la superclase común (CActor), lo que implica que todas las subclases hereden dichas características (incluyendo CAnimal y CFlyingCreature). El efecto que esto provoca es superclases repletas de funcionalidad y subclases que se encargan de activar y desactivar funcionalidades heredadas, lo que constituye un diseño orientado a objetos bastante pobre.

Por mencionar 2 ejemplos de arquitecturas clásicas reales, la clase base de Half-Life 1 tiene 87 métodos y 20 variables públicas, mientras que la clase base de Sims 1 tiene más de 100 métodos. Algunas de las consecuencias de las superclases gigantes son el incremento del tiempo de compilación, código difícil de entender y el conocido *fragile base class problem*, el cual provoca que cambios aparentemente pequeños en la clase base tienen un impacto negativo en las subclases. Todos estos problemas no solo surgen en el desarrollo de videojuegos, pero en este campo se acentúan debido a los constantes cambios en la especificación, lo que fuerza a los programadores a revisar y extender una arquitectura bastante estática como es la jerarquía de clases.

2.2.4. Arquitectura Basada en Componentes

Aún dentro del paradigma de orientación a objetos, la arquitectura basada en componentes es la tecnología que se usa hoy en día para la implementación del módulo encargado de las entidades de juego. Un componente en esta arquitectura representa una pequeña porción de funcionalidad idealmente autocontenido que puede ser agregada o eliminada de una entidad de juego. Por tanto, un nuevo tipo de entidad es simplemente una combinación de componentes que pueden ser creados en tiempo de ejecución, lo que aporta flexibilidad y potencia la reutilización.

El mayor problema de la jerarquía de clases es su naturaleza estática, ya que los continuos cambios de diseño provocan que la herencia no sea la mejor solución. Por esta razón, los desarrolladores hoy en día tienden a usar sistemas basados en componentes. En lugar de tener entidades de una clase concreta que defina su comportamiento, ahora cada entidad es simplemente un contenedor donde cada funcionalidad o habilidad que tenga la entidad es

implementada por un componente. Mediante el uso de estas técnicas se cambia una jerarquía de clases estática creada por mecanismos de herencia por un diseño versátil que permite crear nuevas entidades de manera sencilla por combinación de componentes. Ahora las entidades son listas de componentes y los componentes concretos que conforman una entidad no tienen por qué estar descritos en el código fuente sino que pueden especificarse en un fichero externo que se procesa en tiempo de ejecución, haciendo que la creación de las entidades esté dirigida por datos. De alguna manera esta aproximación simplifica la creación de nuevos tipos de entidades, ya que no se requiere ninguna tarea de programación sino simplemente seleccionar las diferentes habilidades requeridas para la entidad de un conjunto de componentes.

Motores de juego más sofisticados suelen ser más versátiles aún, permitiendo del uso de lenguajes de *scripts* para la creación de componentes. El motor CryEngine permite el uso de LUA para la creación de *scripts* mientras que Unity permite el uso de UnityScript y C#. El uso de este tipo de motores permite a los desarrolladores crear componentes de una manera más rápida. Finalmente cabe resaltar que prácticamente todos los juegos modernos se crean con una arquitectura basada en componentes, la cual ha pasado a ser la principal solución frente a los sistemas basados en herencia.

2.3. Metodologías para el Desarrollo de Videojuegos

El proceso de producción de un videojuego es todo el camino que recorre un equipo para transformar una idea en un producto software cerrado listo para ser distribuido. El desarrollo de videojuegos cubre un amplio espectro de desarrollos ya que dentro de este campo caben grandes proyectos de juegos donde trabajan mas de 80 personas durante 2 o tres años, y pequeños desarrollos para móviles, donde grupos pequeños de 4 o 5 personas desarrollan un producto en apenas un par de meses. Este proceso obviamente varía en función del tipo de proyecto que se lleve a cabo, del grupo que lo desarrolle, de los recursos que se tengan, de la plataforma objetivo para la que se desarrolla, de la tecnología elegida etc.

Uno de los grandes problemas de la creación de videojuegos es que, dado que el equipo de desarrollo es multidisciplinar y que es necesario generar funcionalidad jugable lo antes posible, es muy difícil planificar correctamente. Debido al comportamiento emergente del desarrollo de videojuegos, cada vez menos estudios siguen un desarrollo clásico en cascada, donde el equipo de desarrollo crea varios componentes de juego independientes que se unen al final del desarrollo confiando en que todo irá según lo esperado.

Hoy en día, el desarrollo software en general está evolucionando, de manera que el proceso de creación de un producto es cada vez más iterativo. En este proceso cíclico se conciben nuevas ideas o mecánicas, se documentan, se implementan, se evalúan a ver si cumplen los propósitos esperados y en función de eso se descartan, aceptan o se refinan dichas ideas volviendo a empezar el ciclo. Es por eso que, desde hace tiempo, cada vez más

desarrolladores de software en general, y de videojuegos en particular, están adoptando metodologías ágiles como “Scrum” o “Extreme Programming” que están enfocadas en realizar un producto software mediante iteraciones.

Las metodologías ágiles pretenden que se genere nuevo software funcional iterativamente cada poco tiempo, donde las iteraciones pueden ir desde una semana a un par de meses, de manera que se trata de maximizar la simplicidad del software realizado, implementando solo lo que se necesita. Esto incentiva a que se mantenga un ritmo de desarrollo constante y que sea más sencillo medir cuál es el progreso del desarrollo. Estos tipos de metodologías son muy flexibles a los cambios en el diseño y permiten un diseño incremental de mejora constante gracias a que se debe usar tecnología flexible como la arquitectura basada en componentes.

Una de las bases de las metodologías ágiles es que todo el mundo puede mejorar cualquier parte del desarrollo cuando lo necesite, y para esto se debe obligar a que haya integración continua de contenidos, donde el trabajo de uno se debe integrar varias veces al día, evitando así grandes conflictos que pueden surgir cuando se trata de juntar dos versiones desarrolladas por separado.

Los principales beneficios que reportan los casos de éxito al utilizar metodologías ágiles son:

- Al ser metodologías iterativas e incrementales se obtienen versiones jugables del producto en intervalos regulares de tiempo. Esto facilita una visión temprana del resultado final del juego, lo cual reduce la probabilidad de cambios de requerimientos en forma tardía y brinda una mayor retroalimentación del cliente.
- Permiten tener una mayor visión y control del avance del proyecto, tanto al cliente como a los desarrolladores. Esto se debe a que se pueden determinar nuevas estrategias, iteración por iteración, para lograr llegar en tiempo y forma a los plazos requeridos.
- Involucran a todo el equipo en las decisiones, lo que logra compromiso y motivación.

2.4 Metodología Ágil para el Desarrollo de este Proyecto: SUM

2.4.1 Introducción a la metodología

La metodología SUM para videojuegos tiene como objetivo desarrollar videojuegos de calidad en tiempo y costo, así como la mejora continua del proceso para incrementar su eficacia y eficiencia. Pretende obtener resultados predecibles, administrar eficientemente los recursos y riesgos del proyecto, y lograr una alta productividad del equipo de desarrollo. SUM fue concebida para que se adapte a equipos multidisciplinarios pequeños (de dos a siete integrantes que trabajan en un mismo lugar físico o estén distribuidos),

y para proyectos cortos (menores a un año de duración) con alto grado de participación del cliente.

SUM adapta para videojuegos la estructura y roles de Scrum. Se utiliza esta metodología ya que brinda flexibilidad para definir el ciclo de vida y puede ser combinado fácilmente con otras metodologías para adaptarse a distintas realidades.

La metodología SUM es una metodología basada en Scrum, que utiliza la tendencia de las metodologías ágiles debido al contexto con el que se trabaja: programación rápida, precisa, optimizada y adaptable son requerimientos comunes de los proyectos en cuestión, con el que se cuenta con poco personal, poco tiempo y un escenario dinámico, con pocas características y funcionalidades bien específicas. La metodología surge básicamente como la necesidad de cumplir con un proyecto con escasa cantidad de recursos, pero con una fuerte cohesión entre ellos utilizando una metodología adecuada para el trabajo, es ahí cuando SUM surge como la solución frente a éstos parámetros de trabajos. Muchas de las industrias de videojuegos de hoy en día utilizan la metodología Scrum para el desarrollo, ya que SUM se basa en Scrum para “poco personal”.

En resumen, el alcance de la metodología SUM es el siguiente:

- Equipos pequeños (de 2 a 7 integrantes).
- Proyectos cortos (menores a un año de duración).
- Equipos multidisciplinarios.
- Equipos que comparten un lugar de trabajo o están distribuidos.
- Alto grado de participación del cliente.

2.4.2 Roles

La metodología define cuatro roles:

- 1) Equipo de desarrollo: El equipo de desarrollo tiene las características del Scrum team, pero a diferencia de Scrum, se definen subroles dentro del equipo (programador, artista, etc). Es necesario esta definición ya que se requiere una alta especialización para satisfacer las distintas disciplinas que involucra el desarrollo de videojuegos, aspecto no contemplado en Scrum.
- 2) Productor interno: En Scrum, corresponde al rol de Scrum Master.
- 3) Cliente: En Scrum, corresponde al rol de Product Owner.
- 4) Verificador beta: El rol de verificador beta no está presente en Scrum pero sí se detecta su existencia en el relevamiento en la industria del videojuego en general. Su responsabilidad es la de realizar la verificación funcional del videojuego y comunicar su resultado. Sin embargo, puede no poseer experiencia ni ser jugador frecuente y participar igualmente de la verificación (*tester* del juego).

2.4.3 Ciclo de vida

El ciclo de vida se divide en fases iterativas e incrementales que se ejecutan en forma secuencial con excepción de la fase de gestión de riesgos que se realiza durante todo el proyecto. Las cinco fases secuenciales son: concepto, planificación, elaboración, beta y cierre, como se aprecia en la Fig. 2.4.

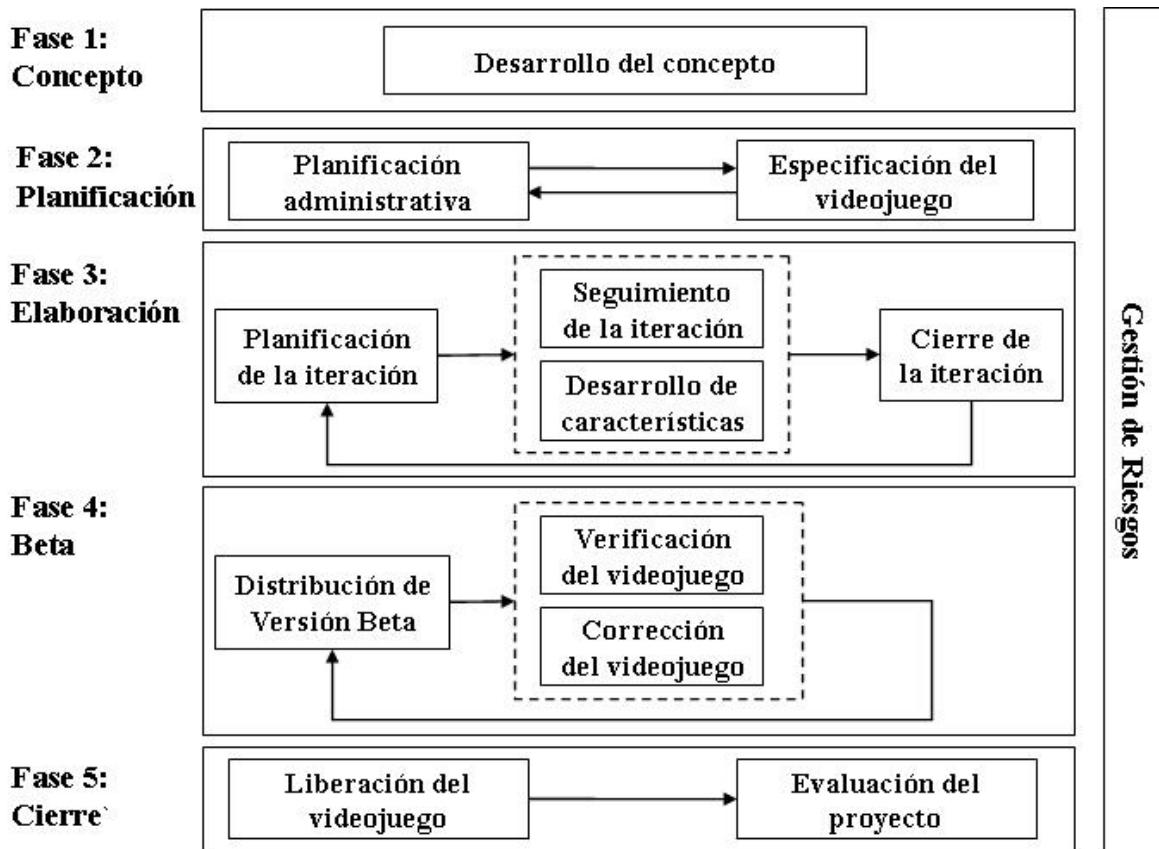


Fig. 2.4 – Ciclo de vida SUM

A tener en cuenta:

- Las fases de concepto, planificación y cierre se realizan en una única iteración, mientras que elaboración y beta constan de múltiples iteraciones.
- Las fases surgen como adaptación al desarrollo de videojuegos de las fases pre-game, game y post-game que presenta Scrum, donde las dos primeras coinciden con las fases de planificación y elaboración, mientras que la tercera se corresponde con las fases de beta y cierre. Esta división se realiza ya que la fase beta tiene características especiales en la industria de videojuegos.
- La fase de concepto no se corresponde con ninguna etapa de Scrum y se agrega ya que cubre necesidades específicas para el desarrollo de videojuegos y se identifica su uso en la industria mundial.

2.4.4 Concepto

Tiene como objetivo principal definir el concepto del videojuego, lo que implica definir aspectos de negocio (público objetivo, modelo de negocio), de elementos de juego (principales características, *gameplay*, personajes e historia entre otros) y técnicos (lenguajes y herramientas para el desarrollo).

El concepto del videojuego se construye a partir de ideas y propuestas de cada rol involucrado sobre los aspectos a definir. Las propuestas se refinan a través de reuniones y se analiza su factibilidad con pruebas de concepto. Esta fase finaliza cuando se tiene el concepto validado entre todas las partes involucradas. No es necesario que el concepto esté definido en forma completa para pasar de fase, ya que hay aspectos que se pueden determinar posteriormente.

A la hora de diseñar un juego, primero hay que tener en cuenta algunos criterios, como ser el género (explicado en el capítulo 1), la vista, la ambientación, la narrativa, las mecánicas y otros más. Se verán estos puntos a continuación.

1) Vista:

Una vista simplemente es cómo se ve el juego en pantalla. Las vistas se diferencian por su aporte gráfico, por su aporte a la navegabilidad y por el tipo de juego en que se usan.

- **Vista en Primera Persona:** Se usa mayormente en juegos de acción o híbridos acción-aventura. Su mayor ventaja es que permite una gran inmersión en el juego. Es la mejor manera de provocar sustos e imprevistos en el jugador. Gráficamente, el detalle se centra en brazos y armas, y en el entorno (para ello se suelen evitar los espacios abiertos). La navegabilidad es sencilla e intuitiva ("adelante" siempre es adelante). La Fig. 2.5 ilustra un ejemplo de una vista en primera persona.



Fig. 2.5 – Vista en primera persona

- **Vista en Tercera Persona:** Es la vista más usada. Su mayor ventaja es permitir que el jugador se sienta representado por el personaje (los

mejores juegos de este estilo, generan franquicias). Gráficamente, el detalle se centra en el personaje.

Tiene ciertas peculiaridades gráficas:

- Se agranda el campo de visión de la cámara.
- Para ganar credibilidad, se quita realismo.

La navegabilidad es más compleja:

- La cámara no se mueve exactamente igual que el personaje.
- A veces, la cámara es independiente del personaje.
- Otras veces, la cámara está fija (cámara predefinida).

La Fig. 2.6 ilustra un ejemplo de una vista en tercera persona.



Fig. 2.6 – Vista en tercera persona

- Vista isométrica: Se la conoce también como "Vista de Dios". Su ventaja es permitir al jugador controlar la mayor cantidad de elementos posibles simultáneamente. Gráficamente, tanto el escenario como los personajes pierden detalle (debido a la distancia y a la cantidad de objetos). En algunos juegos se permite al jugador posicionar la cámara a su antojo (donde él cree que está la acción más relevante). La navegabilidad es la más sencilla de todas (todo está al alcance de un click). La Fig. 2.7 ilustra un ejemplo de una vista isométrica.



Fig. 2.7 – Vista isométrica

- Vista de Cabina: Utilizada en los juegos de simulación. Similar a la vista en primera persona en todos los aspectos, excepto en dos detalles:
 - La cabina tiene un enorme nivel de detalle.
 - Pierde detalle el entorno.

La Fig. 2.8 ilustra un ejemplo de una vista de cabina.



Fig. 2.8 – Vista de Cabina

2) Ambientación

La ambientación indica el cuándo (ambientación temporal) y el dónde (ambientación espacial) se va a ubicar el juego.

- Ambientación temporal Pasado: Puede ser real o alternativo. En el caso de ser real, es muy importante investigar historia, geografía, costumbres, etc primero. Ejemplo: Fig. 2.9.



Fig. 2.9 – Un caballero Templario

- Ambientación temporal Presente: Es donde hay menos diseño posible (se copia más y se inventa menos). Ejemplo: Fig. 2.10.



Fig. 2.10 – Juego GTA V

- Ambientación temporal Futuro: No hay nada que copiar, todo diseño parte desde cero. Hay mayor carga de trabajo en el diseño... y mayor riesgo de cometer errores. A pesar de todo, debe estar anclado a las leyes físicas reales. Ejemplo: Fig. 2.11.



Fig. 2.11 – Escenario futurista

- Ambientación espacial Universo real: No hay carga de diseño. Es algo que existe. Ejemplo: Fig. 2.12.



Fig. 2.12 – Juego de Football Americano

- Ambientación espacial Universo fantástico: Todo es inventado. Ejemplo: Mario Bros. Ejemplo: Fig. 2.13.



Fig. 2.13 – Juego Mario Bros. (Nintendo 64)

- Ambientación espacial Universo anacrónico: ¿Qué hubiera pasado si...?. Ejemplo: STALKER: Shadow of Chernobyl. Ejemplo Fig. 2.14.



Fig. 2.14 – Portada del juego STALKER: Shadow of Chernobyl

3) Otros elementos de diseño

Una vez que ya se tiene definido qué tipo de juego se pretende diseñar, su vista y su ambientación, se pasa al diseño en sí mismo. El diseño está compuesto básicamente por tres partes: el tema, las reglas del juego y la narrativa.

- Tema: La descripción, en una frase, del juego.
- Reglas del juego: Las mecánicas o reglas que hacen al juego. El conjunto de las mecánicas conforman el juego.
- Narrativa: La historia por detrás del juego. Indica los objetivos mediados e inmediatos.

Lo primero que hay que hacer es elegir un tema, en base a ese tema, se hace un *brainstorming* de ideas (podría traducirse al español como un conjunto de ideas). Finalmente, se desarrolla la narrativa. El tema y la narrativa están íntimamente relacionados, es importante que la narrativa de cree basada en las reglas del juego (porque de un tema pueden salir muchas narrativas, pero del conjunto de reglas no, hacerlo a la inversa es un error habitual).

Como observación aparte, un dato a tener en cuenta es que en general, muchos de los temas que eligen las empresas de desarrollo de videojuegos lo hacen por medio de la denominada *Unfulfilled Dream Theory* (Teoría de los sueños no cumplidos). Se busca un tema a lo que mucha gente aspire o quiera ser o tener. Ejemplos: *Guitar Hero* (el sueño de ser un guitarrista de una banda famosa), *NintenDogs* (en Japón, debido al poco espacio en la viviendas, a veces es imposible tener una mascota, de allí la idea de una mascota virtual).

El paso siguiente es definir varios elementos que aseguran la diversión del juego. Estos son:

- *Gameplay* (jugabilidad): La jugabilidad es una secuencia de decisiones interesantes:
 - Secuencia: las decisiones en un juego deben poder tomarse una a una.
 - Decisiones: el jugador nunca debe creer que lo están conduciendo.
 - Interesantes: Una decisión es interesante según tres parámetros: justicia, *feedback* (retorno o respuesta) y variedad.
 - Justicia: el jugador nunca tiene que pensar que el diseñador le hace trampa (el jugador no es un rival, es un compañero).
 - *Feedback*: todo lo que hace el jugador debe tener un retorno.
 - Variedad: las acciones no deben ser ni monótonas ni repetitivas.
- Progresión: La narrativa sigue una línea heredada del cine de Hollywood. La tensión narrativa crece hasta un punto en el cuál decrece hasta otro punto donde está el conflicto. Desde allí comienza un nuevo crecimiento de tensión hasta la resolución con un final feliz. Todo juego tiene dos tipos de complejidad:

- Horizontal: cantidad de acciones que pueden realizarse en simultáneo (ejemplo: simuladores).
- Vertical: acciones diferentes que se puede realizar en distintas etapas del juego (ejemplo: juegos de acción). A mayor complejidad vertical, mayor progresión del juego (y por lo tanto mayor ilusión de libertad). La progresión se establece en ir agregando dificultad al juego. El objetivo de esto es que el jugador nunca se aburra:
 - Si ponemos toda la dificultad de golpe, el jugador no puede superar el obstáculo y deja el juego.
 - Si no hacemos que cada obstáculo sea un poco más difícil que el anterior, el jugador no encuentra desafío y deja el juego.

Puede haber juegos sin progresión (y juegos excesivamente progresivos).

- Riesgo y Recompensas: Una regla de juego es divertida cuando implica riesgo. A mayor riesgo, mayor recompensa. Distintos tipos de juego recompensan de distintas maneras. El jugador no debe sentir que tiene el control de todo en todo momento (siempre hay alternativas extras). No debe confundirse con la progresión.
- 4) Las 3 C: Hay tres elementos muy importantes que definen a un videojuego:
- *Character* (personaje): El personaje es aquél con quien el jugador se sentirá identificado. Se debe diseñar su historia, su apariencia y su comportamiento.
 - *Camera* (cámara):
 - Fija: La cámara está posicionada en algún lugar del mundo. Si el personaje sale de su rango de visión, se cambia de cámara.
 - Móvil siguiendo al personaje: Puede ser en primera persona (en los ojos del personaje) o tercera persona (a cierta distancia del personaje).
 - Usada en juegos deportivos (sigue la pelota y no el personaje).
 - Ovni: La cámara se posiciona lejos del plano del juego. Ofrece mayor amplitud de visión y control de todo el juego.
 - *Control* (control): Define qué hace el personaje y cómo lo hace. Se define qué tipo de periféricos se usarán como interfaz.

Si cualquiera de estos 3 falla (o está mal diseñado), el juego está mal diseñado.

2.4.5 Planificación

La fase tiene como objetivo principal planificar las restantes fases del proyecto. Para ello es necesario definir el cronograma del proyecto junto con sus principales hitos, conformar el equipo para la fase de elaboración de acuerdo a las necesidades técnicas del proyecto, determinar y tercerizar las

tareas que el equipo no pueda cumplir, definir el presupuesto y especificar el videojuego.

El cronograma del proyecto determina la cantidad de iteraciones y su duración en la fase de elaboración junto con las fechas en las que se planea realizar el pasaje a las etapas beta y cierre. Pueden existir hitos intermedios de avance para cumplir con requerimientos del cliente, algo que es común por causa de los contratos que se realizan en la industria de videojuegos. Se conforma el equipo para el resto de las etapas del proyecto de acuerdo a las necesidades técnicas y artísticas que se identifican. Esta definición puede implicar cambios en el equipo de la fase anterior para cumplir con los requerimientos. En caso de que existan necesidades que las personas que integran el equipo no pueden cubrir, éstas deben ser cubiertas por contratistas externos. La selección y la contratación de estos también es parte de esta tarea. Definir el presupuesto consiste en determinar cuáles son y cómo obtener los recursos económicos necesarios para realizar el proyecto.

Dos de los componentes principales del presupuesto son los salarios del equipo y los costos externos, como por ejemplo el *hardware* necesario para desarrollar o el pago a contratistas externos. Estos aspectos componen la planificación administrativa del proyecto, y es el productor interno el responsable de la actividad. Se apoya en el equipo para detectar las necesidades del proyecto y elaborar el cronograma. El cliente también participa, ya que debe dar el aval al cronograma y el presupuesto.

Especificando el videojuego consisten en describir, estimar y priorizar cada una de las características que definen el videojuego. Una característica representa una funcionalidad del videojuego desde el punto de vista del usuario final. La descripción de cada característica es breve pero permite suficiente detalle para poder estimar el tiempo necesario para realizarla. Al ser definidas desde el punto de vista del usuario final, las características son una excelente herramienta que tiene el cliente para comunicar al equipo los requerimientos del videojuego y medir el avance durante todo el proyecto.

El proceso para especificar las características consta de tres pasos:

- 1) En el primero, el equipo junto con el cliente determinan y describen, a partir del concepto del juego, cuáles son las características funcionales y no funcionales del videojuego. La descripción incluye los criterios de aceptación que sirven como herramienta para verificar la característica y para eliminar ambigüedades en la definición de la misma.
- 2) En segunda instancia, el cliente, con el apoyo del equipo, prioriza estas características de acuerdo a su importancia.
- 3) Por último, el equipo estima cuánto tiempo requiere realizar cada una. La especificación que se obtiene en esta fase es flexible ya que a lo largo del proyecto se pueden agregar, modificar y eliminar características, mientras que la prioridad y la estimación de cada característica se actualiza en cada iteración de la fase de elaboración.

2.4.6 Elaboración

El objetivo de esta fase es implementar el videojuego. Para ello se trabaja en forma iterativa e incremental para lograr una versión ejecutable del videojuego al finalizar cada iteración. El proceso sigue la secuencia de actividades que se muestra en la Fig. 2.15 y que se detallan a continuación.

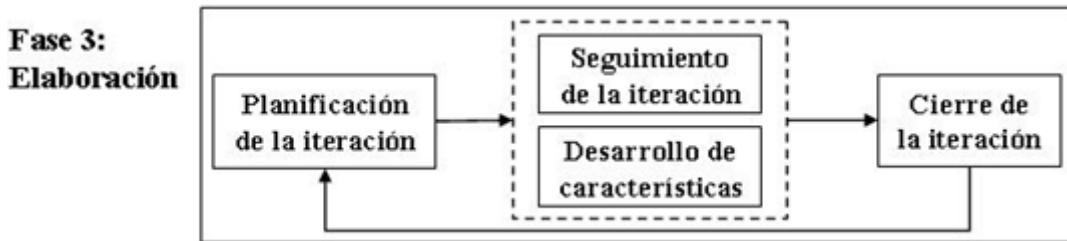


Fig. 2.15 – Fase 3: Elaboración

- 1) **Planificar iteración:** En esta actividad se planifican los objetivos a cumplir, las métricas a utilizar en el seguimiento, las características a implementar y las tareas necesarias para ello. Los objetivos describen qué se pretende lograr al finalizar la iteración y se utilizan para evaluar el éxito de la misma. Sirven también de guía para la toma de decisiones en el transcurso de la iteración. La selección de las características se realiza en base a su prioridad y a los objetivos de la iteración. La suma de los tiempos estimados de las características seleccionadas no debe superar la duración de la iteración. Existen diversas técnicas para llevar a cabo esta tarea, las cuales se brindan como guías del proceso. Cada característica elegida, se divide en tareas de menor duración lo cual hace más sencillo estimarlas, asignarlas a un miembro del equipo, identificar desviaciones, verificarlas y evaluar su completitud. El cliente y el equipo son los responsables de definir los objetivos y las características a implementar. El equipo además determina las tareas necesarias para realizar las características.
- 2) **Seguimiento de la iteración:** su propósito es mantener la visión y el control de la iteración en base a los objetivos planteados. Para ello es necesario definir métricas, registrar medidas y comunicar sus resultados. En caso que ocurran problemas se deben identificar soluciones posibles de acuerdo a su impacto en los objetivos de la iteración y del proyecto. Posibles soluciones pueden ser, ingresar nuevas tareas a realizar en la iteración o cambiar el plan de la iteración en caso de desviaciones críticas. El productor interno realiza el seguimiento y mantiene informado al cliente y al equipo del avance. Las soluciones a los problemas son acordadas entre las personas involucradas.
- 3) **Desarrollar características:** se desarrollan las características planificadas a través de la ejecución de las tareas que la componen. Una vez que se completan todas las tareas pendientes de una característica, esta se verifica de acuerdo a los criterios de aceptación establecidos. En caso de que no cumpla con alguno de los criterios se debe corregir hasta que lo haga. El proceso para llevar a cabo una tarea se ilustra en la Fig. 2.16.

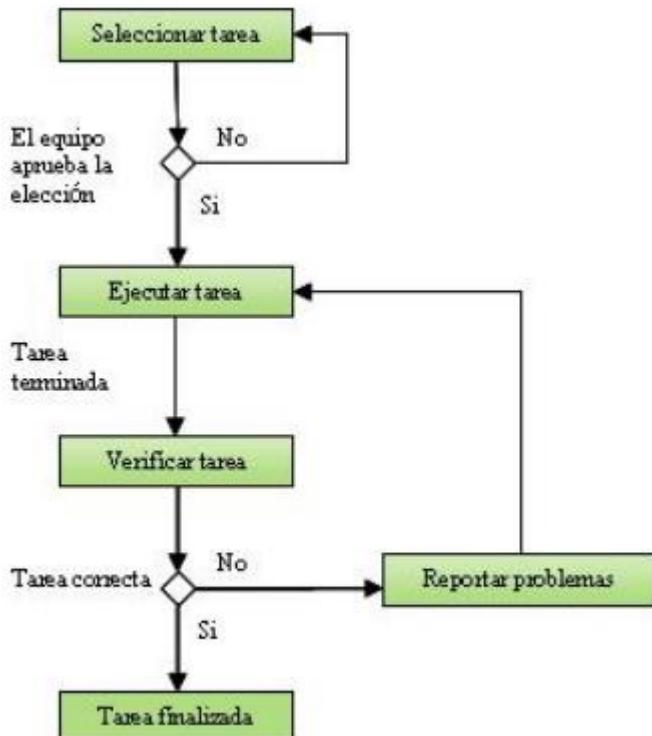


Fig. 2.16 - Proceso para llevar a cabo una tarea

Los miembros del equipo seleccionan las tareas de acuerdo a sus capacidades, y una vez que el equipo aprueba su elección, son responsables por el correcto cumplimiento de estas. Al ejecutar una tarea se pueden identificar nuevas tareas necesarias para completarla, en ese caso se ingresan como nuevas tareas de la iteración. Una vez que se completa la tarea esta es verificada y en caso de encontrar errores se reportan para ser corregidos.

- 4) Cierre de la iteración: Esta actividad implica la evaluación del estado del videojuego y de lo ocurrido en el transcurso de la iteración para actualizar el plan de proyecto respecto a la situación actual. A partir de los criterios de aceptación el cliente puede obtener una medida del estado de cada característica planificada para la iteración. El equipo y el productor interno son los encargados de presentarle la versión actual del videojuego con las características construidas. Con esta evaluación se actualiza el plan de proyecto de acuerdo a la situación actual y se pueden agregar, cambiar o eliminar características del videojuego, así como modificar la prioridad y tiempo estimado de cada una de ellas. Estos cambios los realizan el cliente y el equipo, mientras que el productor interno es responsable de actualizar el plan de proyecto.

2.4.7 Beta

La fase tiene como objetivos evaluar y ajustar distintos aspectos del videojuego como por ejemplo gameplay, diversión, curva de aprendizaje y curva de dificultad, además de eliminar la mayor cantidad de errores detectados. Se trabaja en forma iterativa liberando distintas versiones del videojuego para verificar. Para ello primero se distribuye la versión beta del

videojuego a verificar y se determinan los aspectos a evaluar y la forma de comunicación. Mientras la versión se verifica, se envían reportes con los errores o evaluaciones realizadas. Estos reportes son analizados para ver la necesidad de realizar ajustes al videojuego. Se puede optar por liberar una nueva versión del videojuego para verificar una vez que se realizan los ajustes. El ciclo termina cuando se alcanza el criterio de finalización establecido en el plan del proyecto. El productor interno y cliente seleccionan a los verificadores beta, proporcionan la versión a probar y establecen los mecanismos de comunicación. Los verificadores beta reportan los errores encontrados y sus reacciones sobre los aspectos mencionados, mientras el equipo de desarrollo es quién corrige el videojuego.

Las tres 5s: Esta es una regla de diseño que debe cumplirse para asegurar la diversión:

- 5 segundos vista: a los 5 segundos de juego, el jugador debe sentir que entiende los controles y que le divierte utilizarlos.
- 5 minutos vista: a los 5 minutos de juego, el jugador debe haber experimentado una pequeña misión del juego y tiene que haber encontrado interesante el universo.
- 5 horas vista: a las 5 horas de juego, el jugador debe haber entendido el conflicto narrativo y le divierte solucionarlo.

2.4.8 Cierre

Esta fase tiene como objetivos entregar la versión final del videojuego al cliente según las formas establecidas y evaluar el desarrollo del proyecto. Para la evaluación se estudian los problemas ocurridos, los éxitos conseguidos, las soluciones halladas, el cumplimiento de objetivos y la certeza de las estimaciones. Con las conclusiones extraídas se registran las lecciones aprendidas y se plantean mejoras a la metodología. En la evaluación es recomendable que participen todas las personas que han estado involucradas en el proyecto.

2.4.9 Gestión de riesgos

Esta fase se realiza durante todo el proyecto con el objetivo de minimizar la ocurrencia y el impacto de problemas. Esto se debe a que distintos riesgos pueden ocurrir en cualquiera de las fases, por lo cual siempre debe existir un seguimiento de los mismos. Para cada uno de los riesgos que se identifican se debe establecer la probabilidad y el impacto de ocurrencia, mecanismos de monitoreo, estrategia de mitigación y plan de contingencia.

2.4.10 Guías

Las guías son sugerencias, pautas y herramientas para llevar a cabo en forma efectiva y eficaz las actividades que componen el proceso. A través de ellas, se incorporan a la metodología prácticas aplicadas con éxito para el desarrollo de videojuegos, además de las lecciones aprendidas con el desarrollo de cada proyecto. Actualmente SUM incluye las prácticas y

herramientas de Scrum y XP, y además, artículos publicados sobre la aplicación de metodologías ágiles en el desarrollo de videojuegos. Se pueden encontrar en este enlace:

http://www.gemserk.com/sum/Sum/customcategories/Gu%C3%ADas_BB74B194.html?nodeId=90cb724b

Capítulo 3

Adaptación de la Metodología al Proyecto

3.1 Fase 1: Concepto

Antes de comenzar, es importante aclarar y tener en cuenta la siguiente:

- La metodología es aplicada para equipos de 2 a 7 personas, pues mínimamente un programador y un artista debe haber. En este proyecto, se asume el rol de programador y se utiliza arte descargada gratuitamente perteneciente a terceros.
- El proyecto no tiene fines comerciales, solo académicos.

3.1.1 Herramientas

En el Capítulo 1.4, se vió que un videojuego constaba básicamente de 3 partes: el motor, *scripts* y *assets*. Pues bien, el motor elegido para llevar a cabo este videojuego fue Unity, cuyo logotipo se muestra en la Fig. 3.1.



Fig. 3.1 - Logo oficial Unity

Web oficial: <http://unity3d.com/es/unity>.

Una de las ventajas de trabajar con este motor es que posee versión gratuita, como también la característica de ser multiplataforma, algo que será útil a la hora del desarrollo de la aplicación para Android con Realidad Aumentada, el cuál es otro software independiente a este videojuego y se tratará más adelante. La Fig. 3.2 muestra el listado de plataformas que soporta este motor:



Fig. 3.2 - Plataformas que soporta Unity

Unity soporta dos lenguajes:

- C# (pronunciado C-sharp), un lenguaje de la industria estándar similar a Java o C++.
- UnityScript, un lenguaje diseñado específicamente para uso con Unity y modelado tras JavaScript.

Adicional a estos, otros lenguajes .NET pueden ser utilizados con Unity si estos compilan un DLL compatible.

Luego, para la parte de *scripts*, el entorno de desarrollo elegido es Microsoft Visual Studio, cuyo logotipo se muestra en la Fig.3.3:

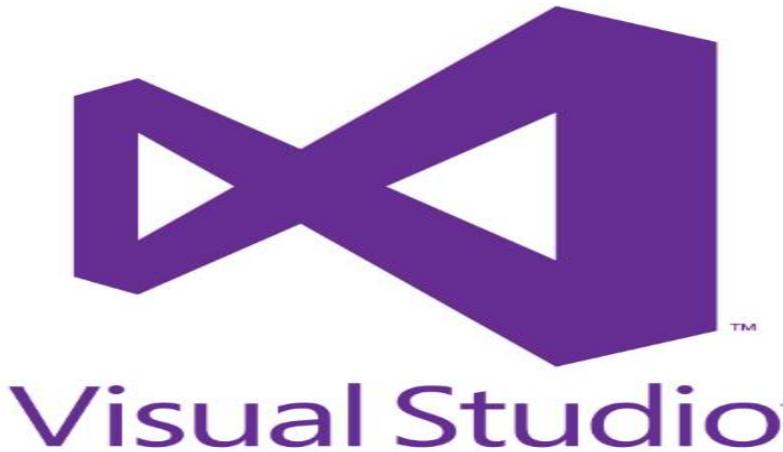


Fig. 3.3 – Logo Visual Studio

Visual Studio aporta Un ambiente de desarrollo para C# más sofisticado, con un autocompletado inteligente, cambios a los archivos fuente con ayuda de la computadora, un resaltado de sintaxis inteligente y más (por defecto, Unity trae el Mono Develop como editor de código). VisualStudio es un producto de Microsoft. Viene en una edición Express o Profesional (la edición Express es la versión gratis). La Fig. 3.4 muestra a Visual Studio como editor de código seleccionado dentro del motor Unity.

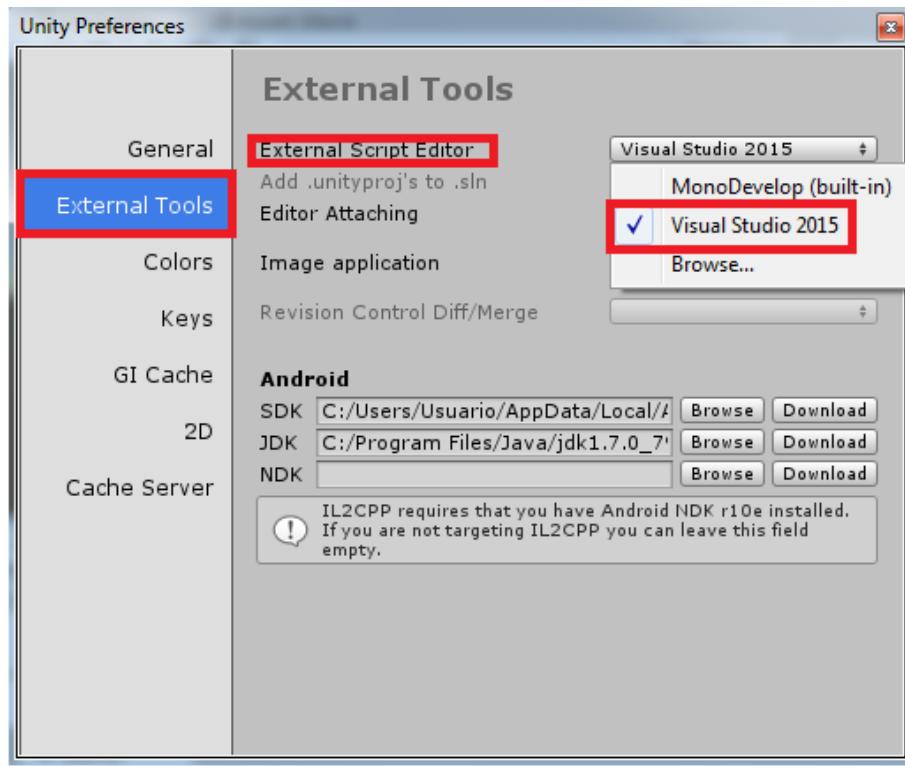


Fig. 3.4 – Visual Studio 2015 como editor de código seleccionado en Unity

Finalmente, los *assets* del juego son provistos por terceros y por Unity.

3.1.2 Elementos de diseño

El género de este videojuego es el de acción/aventura (del estilo Tomb Raider) puesto que el mismo mantiene al jugador moviéndose y ocupado todo el tiempo y usa la violencia como su principal característica de interacción, ya que el personaje cuenta con un arma de fuego (pistola) la cuál utiliza tanto para disparar como para golpear con su culata.

Fig. 3.5: Juego Tomb Raider III. Un antiguo juego de acción/aventura que se asemeja al creado en este proyecto en algunos puntos como la vista, las mecánicas y el personaje principal (una mujer con pistola).

Fig. 3.6: Captura del juego creado “Force of Will”.



Fig. 3.5 – Tomb Raider III lanzado en 1998.



Fig.3.6 – Force of Will. Juego creado

La vista del juego es en tercera persona, donde en algunos niveles, el campo de visión es mayor que en otros (Fig. 3.7 y Fig. 3.8).



Fig. 3.7 – Cámara en tercera persona. El campo de visión incluye el cuerpo entero del personaje en este caso.



Fig. 3.8 – Cámara en tercera persona. El campo de visión es distinto y solo cubre la mitad del cuerpo principal.

La ambientación temporal se sitúa en la época actual. En cuanto a la ambientación espacial, es un universo fantástico (Fig. 3.9).

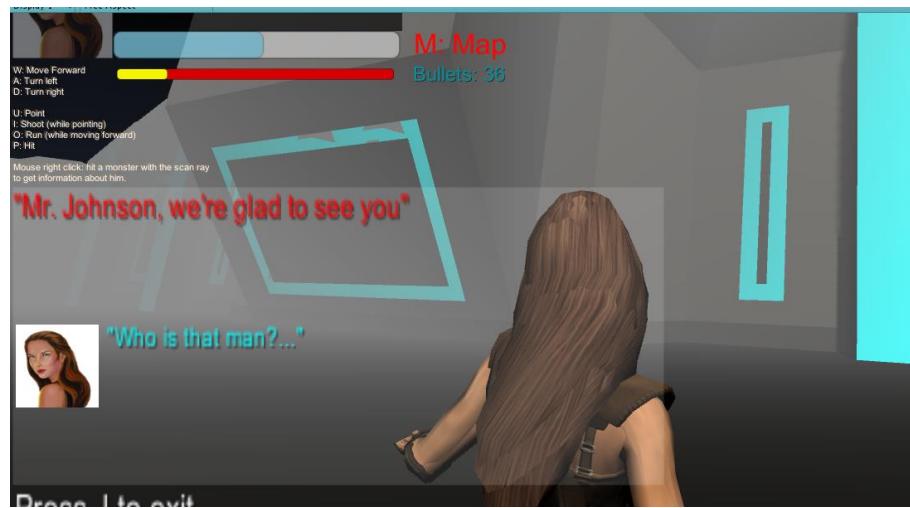


Fig. 3.9 – Un nivel del juego

El tema: Un videojuego para PC el cuál utiliza realidad aumentada mediante un dispositivo móvil externo.

Reglas del juego: un personaje humano que camina, corre, dispara y golpea con la culata de la pistola. El mismo es controlable desde el teclado y utiliza como complemento de ayuda el ratón y un dispositivo móvil en algunas situaciones.

Narrativa: El personaje se llama Galatea, vive en Chicago (Estados Unidos) y se encuentra en terapia intensiva en el hospital debido a un accidente de causas desconocidas. De allí surge el nombre del juego “Force of will” (en español, fuerza de voluntad), donde el personaje libra una lucha entre la vida y la muerte, vagando por su mente buscando respuestas. Al final del juego, este entiende que el único obstáculo a vencer es él mismo. Para ilustrar lo último mencionado, se eligió como “jefe” final al mismo modelo 3D del personaje, pero sin cabeza, para darle un poco de ambientación de terror al juego (Fig. 3.10).

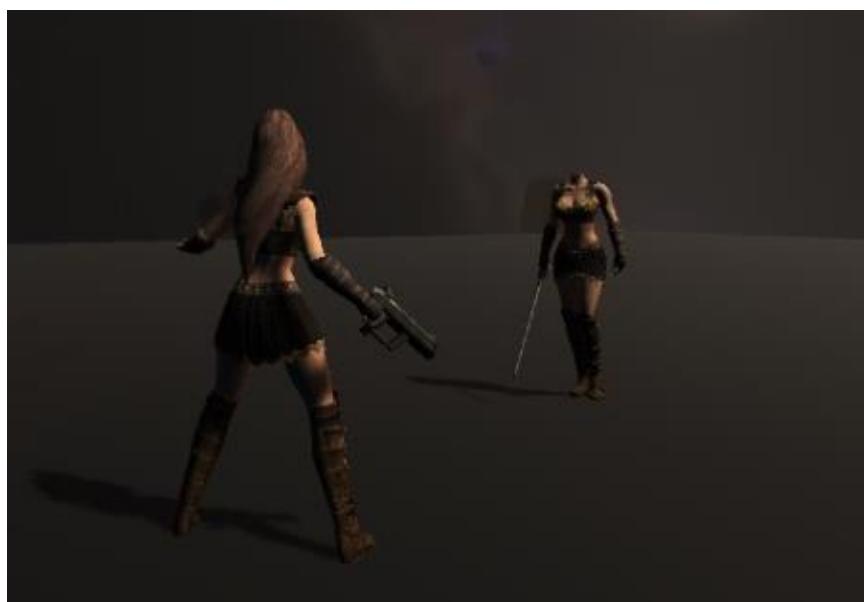


Fig. 3.10 - Nivel final del juego, el último combate

Durante el juego también se observan algunos diálogos confusos, otros sin sentido, y otros provocativos. Muchos de ellos son propios de las personas que se encuentran en la sala del hospital y Galatea los interpreta de alguna forma en el subconsciente. Otros son simples recuerdos/memorias o cosas inexplicables que se experimentan al tener un sueño. Finalmente, están los diálogos más raros, como una voz del más allá que le dice al personaje cosas como “únete a mí” o “ya es hora”. Esto pretende de alguna forma representar a la muerte o al sentimiento de rendición, pero queda a criterio del jugador, ya que hasta el mismo personaje al final del juego se pregunta a sí mismo qué significaba esa voz un poco aterradora.

Se debe recorrer niveles y superar obstáculos, como enemigos y otras restricciones. Las acciones a realizar varían de nivel en nivel, en algunos casos es preferible matar la mayor cantidad de monstruos, en otros, el objetivo es recolectar cristales para cumplir con la misión (progresión vertical). Hay casos en donde a simple vista el nivel es confuso (muy grande y con niebla), por lo que la ayuda del dispositivo móvil y la realidad aumentada serán fundamentales. También se incluye un modo de juego alternativo: *survival* (supervivencia), donde la jugabilidad cambia, ya que el desafío ahora es sobrevivir en un laberinto y encontrar la salida antes de terminar el tiempo.

A la hora de hablar del riesgo y la recompensa, hay niveles donde el objetivo es llegar hasta un destino. En ese caso, el jugador puede optar por no gastar balas y tratar de evadir al enemigo. Esto puede ser útil cuando se juega por primera vez ya que el “mundo” es totalmente desconocido y se puede reservar las mismas para situaciones más complejas.

En otros niveles, donde el objetivo principal es recolectar cristales en un escenario plagado de zombies, la decisión de ir por ellos a veces implica el riesgo de ser golpeado por el enemigo y perder vida; el jugador decidirá en ese caso si es más conveniente esperar por recuperar puntos vitales primero, gastar su balas en dicho enemigo, ignorar el cristal, usar el dispositivo móvil con realidad aumentada para inspeccionar las características del enemigo, etc. La recompensa, claro, obtener el cristal para terminar el nivel lo más rápido posible.

El personaje es una mujer llamada Galatea (el nombre original del Asset es Female Warrior Princess), donde ya se mencionó antes su historia cuando se explicó la narrativa. El personaje fue descargado del *Unity Asset Store*: <https://www.assetstore.unity3d.com/en/#!/content/44041> e incluía el modelo 3D con su set de animaciones. La decisión de seleccionarlo, además de que es gratis, es que tenía una buena cantidad de animaciones diferentes para usar a gusto. Vale aclarar que los sonidos del personaje fueron añadidos de manera independiente y descargados de internet (grito de muerte, de esfuerzo, entre otros).

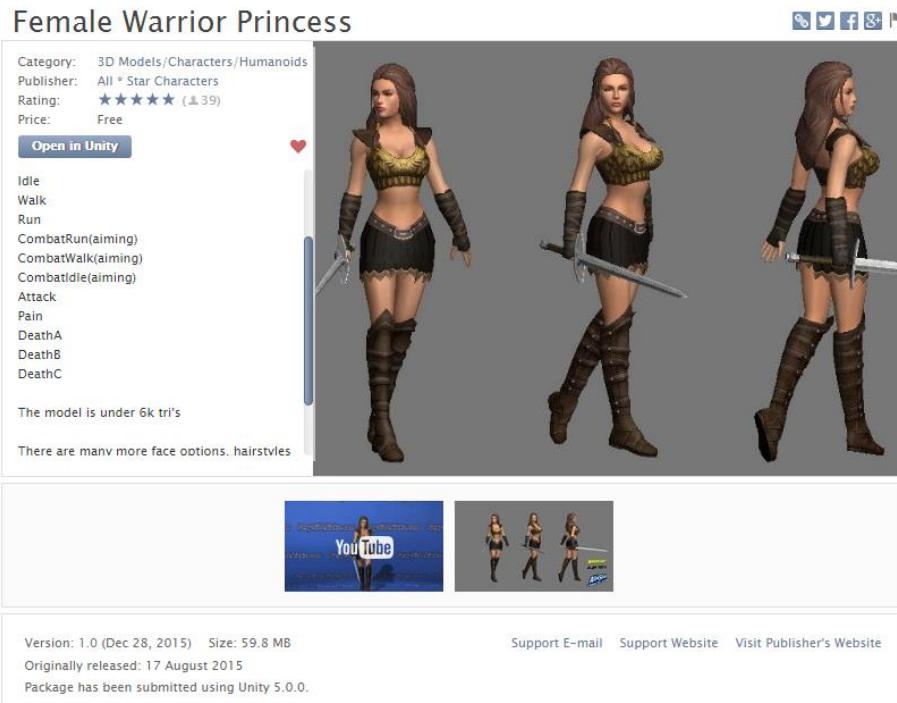


Fig. 3.11 – Asset visto desde el Asset Store de Unity

El personaje puede caminar, girar, correr, disparar y empujar. El mismo tiene una cierta cantidad de vida, estamina y balas. Para el movimiento y las habilidades, el periférico de entrada es el teclado:

- W: Moverse hacia delante (adelante es la dirección positiva del eje Z).
- A: Girar hacia la izquierda.
- D: Girar hacia la derecha.
- U: Apuntar.
- I: Disparar (se debe estar apuntando primero con U).
- O: Correr (mientras se está caminando con W).
- P: Golpear (o empujar).

Fig. 3.12: Captura de pantalla del juego donde se puede apreciar la guía de los controles del personaje. Este texto es visible todo el tiempo mientras se juega.

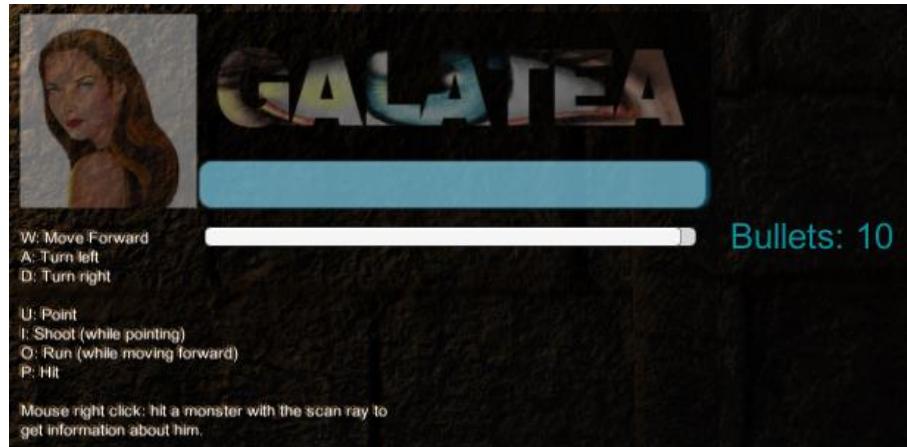


Fig. 3.12 – Dentro del juego, hay una referencia de los controles de forma permanente en la pantalla

Se puede ver en la Fig. 3.12 otro comando, el click derecho, cuya función es lanzar el rayo scanner que, al impactar un monstruo, pausará el juego para mostrar una imagen 2D del mismo y así poder analizarlo con el dispositivo móvil mediante la realidad aumentada (Fig. 3.13 y Fig. 3.14).

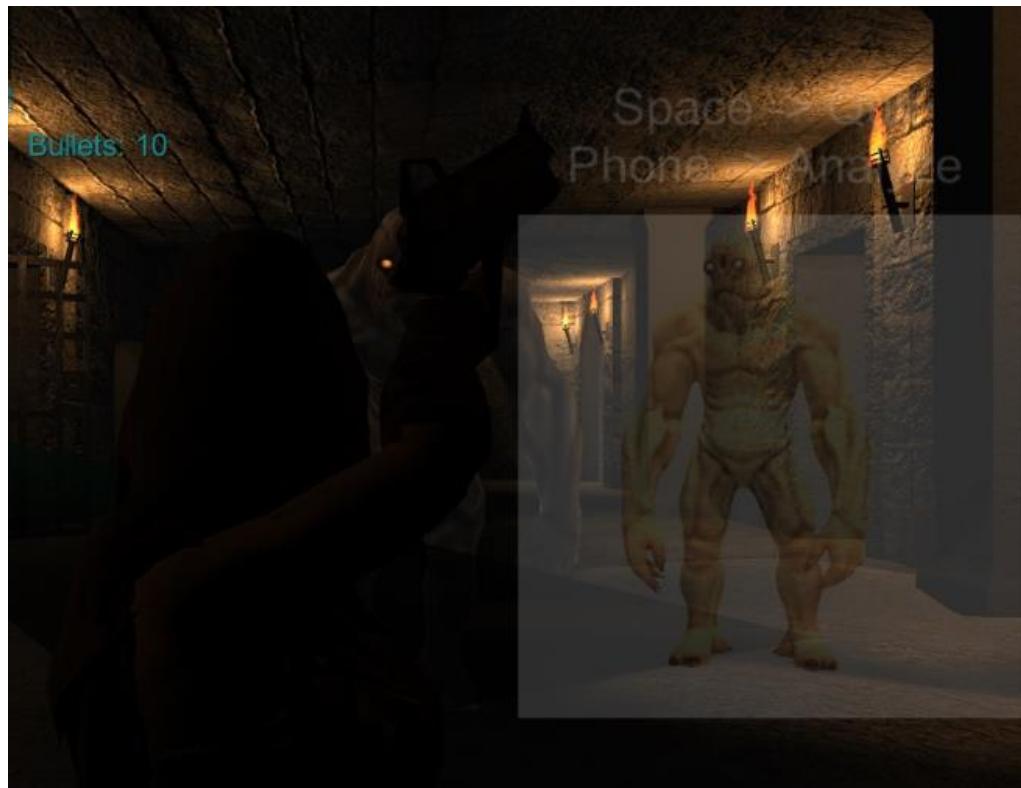


Fig. 3.13 – Instante en el que se utiliza el Scan Ray y este impacta un monstruo. La animación está en curso

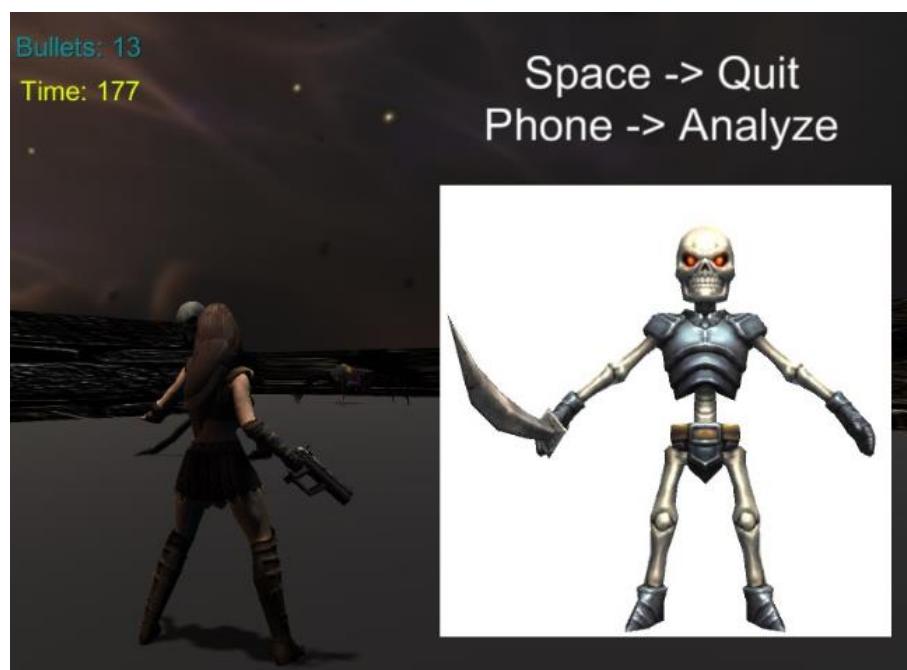


Fig. 3.14 – El Scan Ray lanzado, impactó al enemigo esqueleto. El juego está pausado con la imagen 2D del enemigo disponible en pantalla para ser analizada con el dispositivo móvil (RA)

Imágenes 2D como las de la UI (*User interface* o interfaz de usuario) mostrada en la Fig. 3.12, la del menú de inicio y la del Game Over fueron creadas por un tercero. Las imágenes de los monstruos (Fig. 3.13 y 3.14) son capturas de pantalla recortadas y luego importadas a Unity). La cámara es móvil y está siguiendo siempre al personaje todo el tiempo.

3.2 Fase 2: Planificación

3.2.1 Planificación Administrativa

Anteriormente, se mencionó que este proyecto no tiene fines comerciales sino académicos, y que el equipo de trabajo era de una sola persona, por lo que, se adaptó la metodología SUM de acuerdo a las necesidades del mismo. Temas de presupuesto y salarios son descartados, sin embargo, si hubo colaboración de terceros de forma gratuita para algunos puntos, como ser la música de fondo y algunas imágenes 2D (como el avatar del personaje a la izquierda de la barra de vida). A su vez, ya mencionado en la fase de Concepto, los modelos 3D a usar fueron descargados de forma gratuita desde internet, por lo que todo lo relacionado al arte del videojuego (trabajo perteneciente al rol de artista) fue cubierto por gente ajena al proyecto y no se invirtió para el mismo.

3.2.2 Especificación del videojuego

El proyecto comenzó a realizarse a mediados de Diciembre, se calculó que tardaría aproximadamente entre 4 y 5 meses su desarrollo, teniendo en cuenta que la mayor cantidad de días estarían destinados a la Fase 3 (Elaboración), con un trabajo de 8 horas diarias de lunes a viernes y 4 horas los días sábados. El trabajo del arte (modelo 3D, imágenes 2D y música) se tercerizó como ya se mencionó antes, por lo que el único requisito técnico necesario para llevar a cabo el desarrollo del mismo fue el de una computadora que cumpla con lo siguiente:

- Sistema Operativo: Windows 7 SP1+, 8, 10; Mac OS X 10.8+.
- Windows XP y Vista no son compatibles; y las versiones de servidor de Windows & OS X no se han probado.
- GPU: Capacidades de tarjeta de vídeo con DX9 (modelo de shader 2.0). Todo lo que se haya lanzado desde 2004 funciona.

Los requerimientos técnicos que el usuario final necesita para utilizar este software son una computadora de escritorio o notebook que cumpla con lo siguiente:

- Sistema Operativo: Windows XP SP2+, Mac OS X 10.8+, Ubuntu 12.04+, SteamOS+.
- Tarjeta de video: capacidades DX9 (shader modelo 2.0); por lo general, todo lo que se haya lanzado desde 2004 funciona.

- CPU: Pentium 4 o superior (compatible con el conjunto de instrucciones SSE2).

El videojuego se juega con teclado y mouse, a su vez, también interactúa con un dispositivo móvil. Lo primordial a la hora de desarrollar este proyecto fue obtener un software funcional, es decir, un videojuego con una versión jugable rápido. Vale aclarar que la especificación fue actualizada en varias iteraciones de la fase de elaboración (fase 3), pues, se está trabajando con una metodología flexible a los cambios en la misma.

3.3 Fase 3: Elaboración

3.3.1 Instalación de Unity y Vuforia

Para este proyecto, se utilizó la versión 5.3.1 de Unity, Una versión lanzada el 28 de Diciembre del año 2015. La Fig. 3.15 muestra una ventana informativa en Unity a la hora de abrir la herramienta informando que hay nuevas versiones disponibles para descargar (Fig. 3.16).

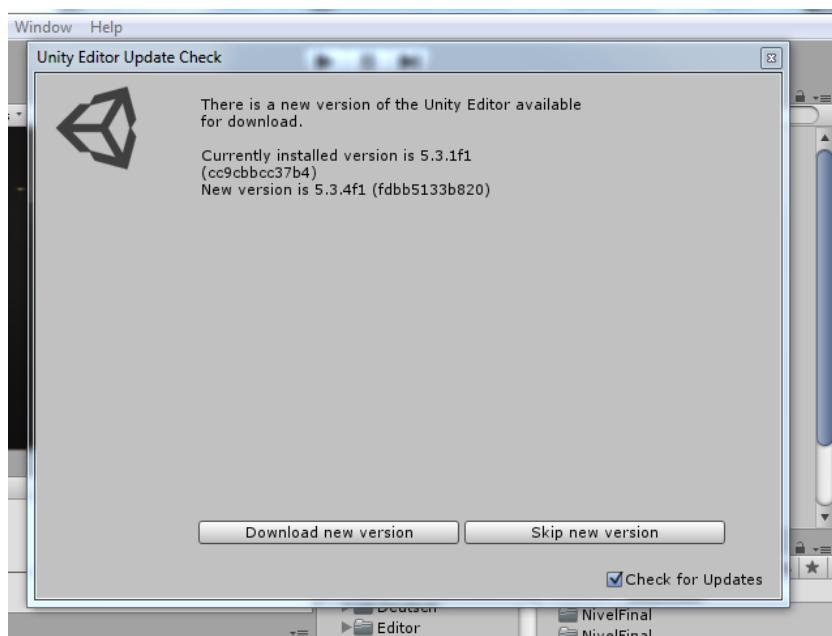


Fig. 3.15 - *Update Check* al iniciar Unity

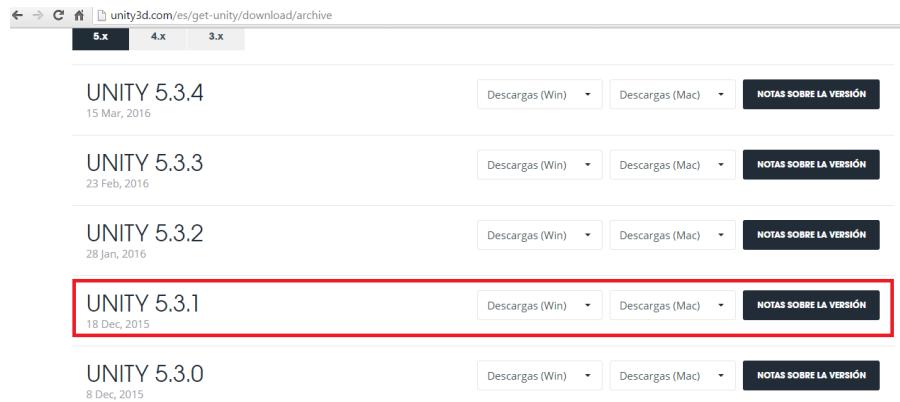


Fig. 3.16 – La versión más reciente de Unity es la 5.3.4

La principal razón por la cual no se actualizó la herramienta a una versión más nueva fue debido a problemas a la hora de utilizar la realidad aumentada. Las primeras pruebas se hicieron con la versión 5.3.1, sin embargo, cuando se actualizó a la versión 5.3.2, las complicaciones aparecieron (Fig. 3.16).

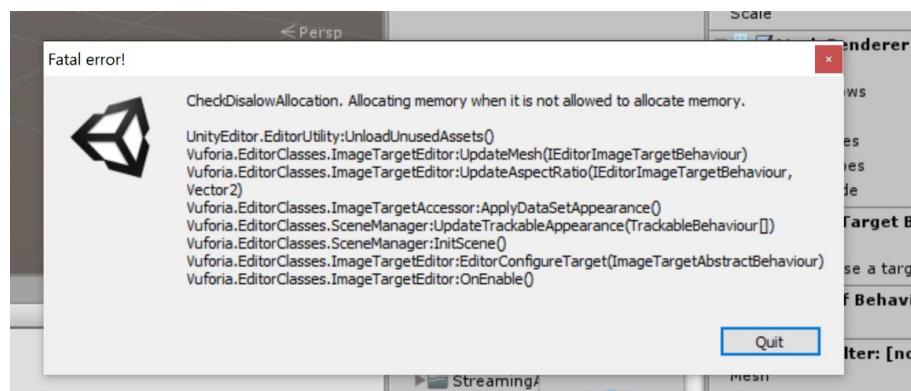


Fig. 3.16 – Ventana de error a la hora de implementar la Realidad Aumentada

Buscando una posible solución en la comunidad de Unity, una persona con el mismo problema creó un hilo previamente. En efecto, la captura de pantalla de la Fig. 4.14 corresponde a dicho usuario, pero es exactamente la misma situación. La Fig. 3.17 muestra el problema planteado en la comunidad de Unity y la Fig. 3.18 la solución del mismo.

Pregunta por FireofLife · 04 Feb a 23: 16 · memory error message error-message fatal error out of memory

fatal error checkdisallowAllocation.
Allocating memort when it is not allowed
to allocate memory

Hello, When I was trying to follow the steps in this link:
<https://developer.vuforia.com/library//articles/Solution/Compiling-a-Simple-Unity-Project>

to use the camera and a target image for a simple AR project. while I was trying to Drag an instance of the ImageTarget prefab into my scene. I received a long message "Fatal error"

Fig.3.17 – Problema planteado en la comunidad de Unity

4 Respuestas · Añadir tu respuesta Ordenar por:

Mejor respuesta
Resposta por sparkle_day · 05 Feb a 13: 09
Hello. I also had the same problem. So, I tried to examine the Vuforia forum.
3 So out of it has is, apparently Unity5.3.2 is the fact that there is no compatibility and Vuforia.
* 5.3.1p4 also the same.

Unity 5.3.1f1 feature release or Unity 5.3.1p3 patch release are recommended to use for now." Once, there was described.
Details: <https://developer.vuforia.com/forum/unity-extension-technical-discussion/notice-unity-532-incompatible-vuforia>

Agregar comentario · Ocultar 1 · Compartir

sparkle_day · 05 Feb a 13: 26 0
Patch is here Unity 5.3.1p3 patch <http://unity3d.com/jp/unity/qa/patch-releases/5.3.1p3>
Unity 5.3.1f1 here <https://unity3d.com/ru/unity/whats-new/unity-5.3.1>
However, when the Unity 5.3.1f1 check, so an error or have a crash often I think that the patch is good.
Details: <http://unity3d.com/jp/search?qq=Unity%205.3.1p3%20patch>

Fig. 3.18 Solución (otro usuario responde)

Aparentemente, la versión 5.3.2 de Unity no tiene compatibilidad con Vuforia, por lo que se volvió a la versión 5.3.1. La versión más actual en ese momento del paquete de Vuforia para utilizar en Unity era 5-0-10, que es la que se utilizó para este proyecto y se tratará más tarde.

3.3.2 Interconexión de escenas o niveles

Antes que nada, es importante mencionar lo que es una escena en Unity (Fig. 3.19). Las escenas contienen los objetos del juego. Pueden ser usadas para crear un menú principal, niveles individuales, y cualquier otra cosa. Se puede pensar en cada archivo de escena, como un nivel único. En cada escena, se coloca el ambiente, obstáculos, decoraciones, el diseño esencial y la construcción del juego en pedazos.

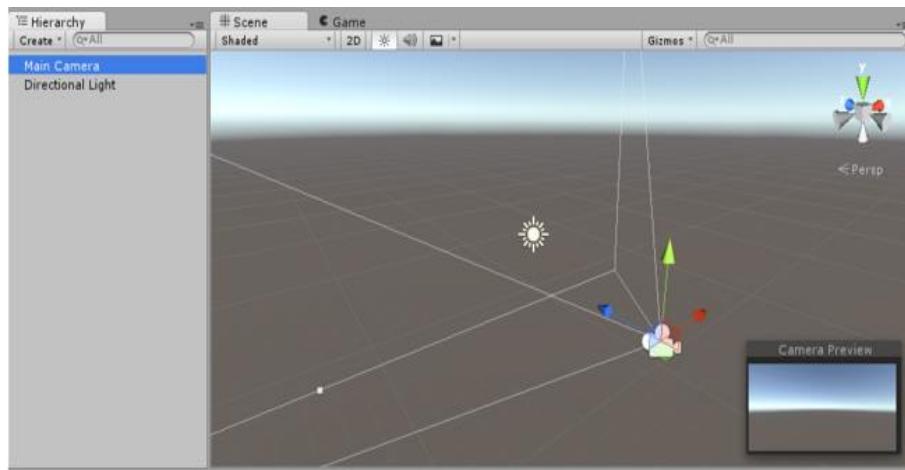


Fig. 3.19 - Una nueva escena vacía, con los objetos 3D predeterminados (una cámara y una *directional light*)

Dentro de cada escena, los objetos están ordenados en forma de árbol, y se trabaja en un entorno de tres dimensiones utilizando coordenadas cartesianas para indicar cualquier punto en el espacio (coordenadas globales), y en muchas ocasiones algunos objetos tienen relación con otros (coordenadas locales). Un ejemplo de este último caso puede ser el objeto arma de un personaje. El personaje es el objeto padre y el arma es un objeto hijo de éste, por lo que su posición y rotación son siempre relativas al padre y no al universo. La Fig. 3.20 muestra el caso de objetos no emparentados y viceversa dentro de Unity.

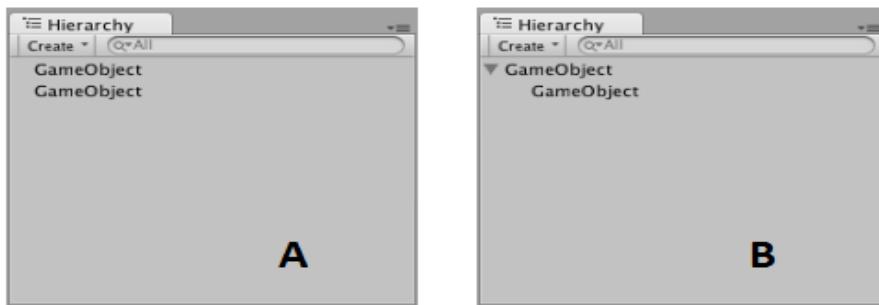


Fig. 3.20 – A: Dos objetos no emparentados – B: Un objeto emparentado a otro

Los objetos no hacen nada por sí mismos. Estos necesitan propiedades especiales antes de que puedan volverse un personaje, un ambiente, o un efecto especial (Fig. 3.21). Si cada uno (objeto) hace diferentes cosas, ¿cómo se hace para diferenciar un personaje interactivo de un cuarto estático?, ¿Qué los hace diferente uno del otro? La respuesta a esta pregunta es que los *GameObjects* (objetos en Unity) son contenedores. Estos pueden guardar las diferentes piezas que son requeridas para hacer un personaje, una luz, un árbol, un sonido, o lo que se desea construir. Entonces, para de verdad entender *GameObjects*, se necesita entender estas piezas que son llamadas componentes.

Dependiendo del objeto que se quiera crear, se agregan diferentes combinaciones de componentes al *GameObject*. Se puede pensar en un

GameObject como una olla vacía de cocina, y los componentes como los diferentes ingredientes que hacen la receta del juego. Unity tiene varios tipos de componentes integrados, y también se pueden hacer componentes propios utilizando *Scripts*.



Fig. 3.21 - Cuatro diferentes *GameObject*s, un personaje animado, una luz, un árbol y una fuente de audio

Los Componentes son las tuercas y tornillos de los objetos y comportamientos de un juego, es decir, son las piezas funcionales de cada *GameObject* (pueden ser agregados, quitados y editados en el mismo). Un *GameObject* siempre tiene el componente *Transform* adjunto (para representar la posición, orientación y el escalado) y no es posible quitar esto. Si un *GameObject* no tuviese un componente *Transform*, sería nada más que alguna información en la memoria de la computadora. Efectivamente no existiría en el mundo. Fig. 3.22: Componente *Transform* de un objeto en Unity.

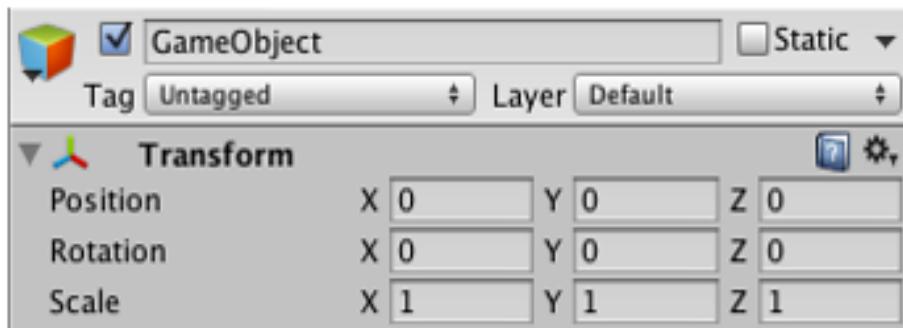


Fig. 3.22 – Componente *Transform*

El juego tiene un total de 23 escenas (Fig. 3.23), de las cuáles:

- 9 escenas 3D (niveles jugables).
- 14 escenas 2D (Textos de la historia, menú, créditos, etc.).

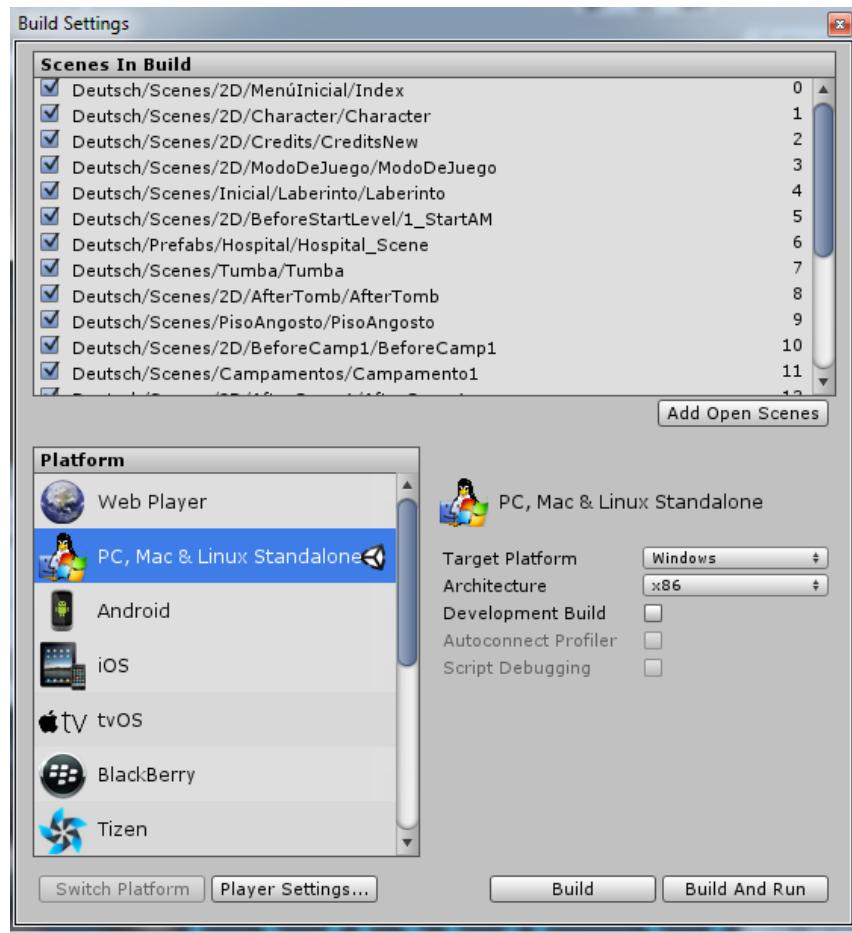


Fig. 3.23 – Esta ventana muestra todas las escenas al momento de crear el archivo ejecutable del juego

En toda escena hay por lo menos un objeto cámara, que será el que visualice lo que ocurre en la misma (Fig. 3.24).

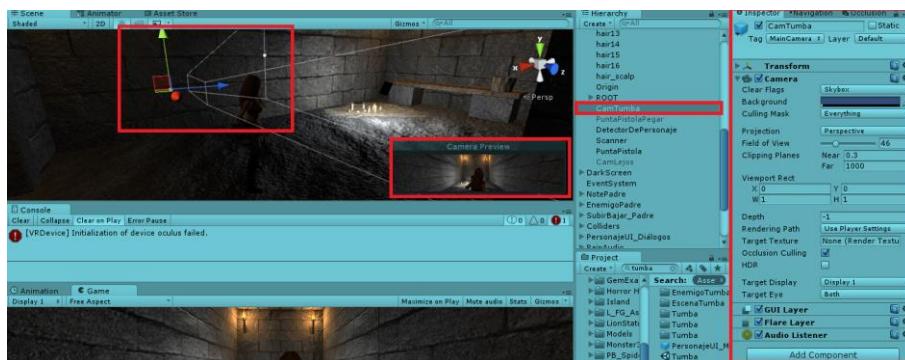


Fig. 3.24 – Objeto cámara en escena. El objeto es hijo del objeto padre que representa el personaje

El juego está conformado por una interconexión de escenas, la condición para pasar de una escena a otra puede variar (presionando una tecla, colisionando con un objeto o haciendo click en un botón). Un objeto perteneciente a la escena "x" es quién determina la condición para pasar a la escena "y". Esa condición está escrita en un *script*, y a su vez, el *script* adjunto al objeto, es decir, el *script* es un componente también de un objeto (Fig. 3.25).

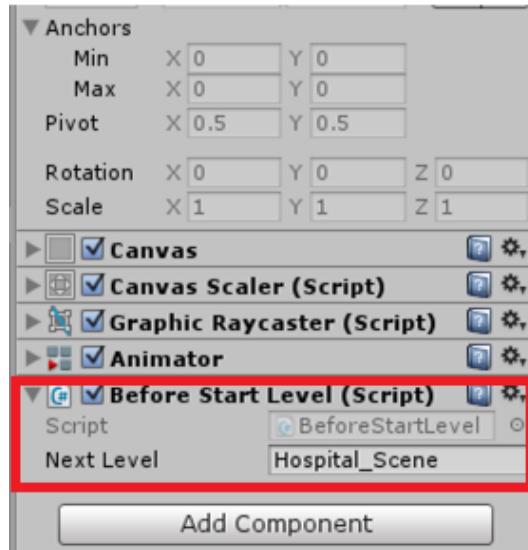


Fig. 3.25 – El recuadro rojo indica el componente *script* en un objeto

Dentro del *script* hay una función perteneciente a las librerías de Unity. Esta es *SceneManager.LoadScene* (“Siguiente Escena”). Para hacer uso de ella, es necesario incluir el espacio de nombres *UnityEngine.Scenemanagement*. Se puede ver un ejemplo a continuación en la Fig. 3.26:

```
using UnityEngine;
using UnityEngine.SceneManagement;
using System.Collections;

public class BeforeStartLevel : MonoBehaviour
{
    public string nextLevel;

    // Update is called once per frame
    void Update ()
    {
        if (Input.GetKeyDown(KeyCode.Space)) SceneManager.LoadScene(nextLevel);
    }
}
```

Fig. 3.26 – Ejemplo de un script del proyecto en el editor de código Visual Studio 2015

El código de la Fig. 3.26 indica que la condición para cargar la siguiente escena es pulsar la tecla Barra espaciadora. La siguiente imagen (Fig. 3.27) muestra un ejemplo del tipo de escena que utiliza esa condición:

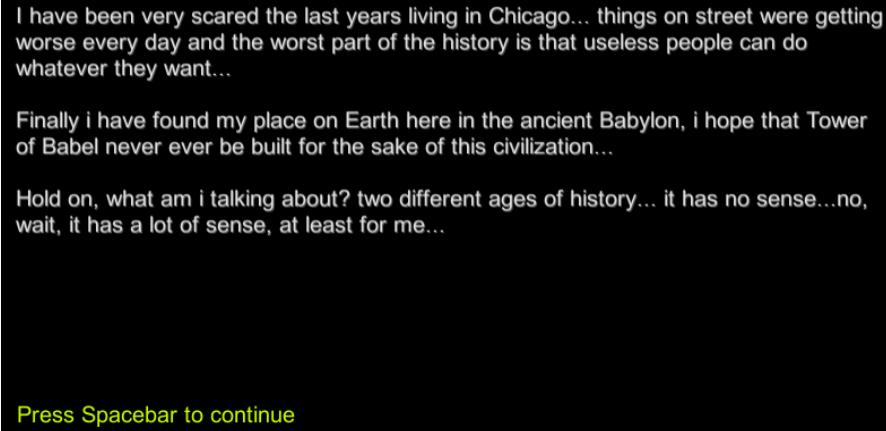


Fig. 3.27 – Escena 2D propia del juego

Todos esos tipos de escena aparecen después o antes de iniciar un nivel jugable, y representan parte de la historia misma del juego (el texto pertenece al personaje) y utilizan una cámara fija (pues, solo se debe mostrar una imagen color negro con textos).

El menú de inicio, mostrado en la Fig. 3.28, es otro tipo de escena similar a las anteriores vistas, solo que el objeto que controla la condición es un botón, y la función es llamada en el evento “click” del mismo (y no controlada cuadro por cuadro esperando la pulsación de una tecla como antes).



Fig. 3.28 – Escena Menú inicio

La Fig. 3.29 muestra un componente *Button* el cuál controla el evento click y llama a la función en el *script* adjunto que carga la siguiente escena. La Fig. 3.30 muestra parte del código de dicho *script* adjunto.

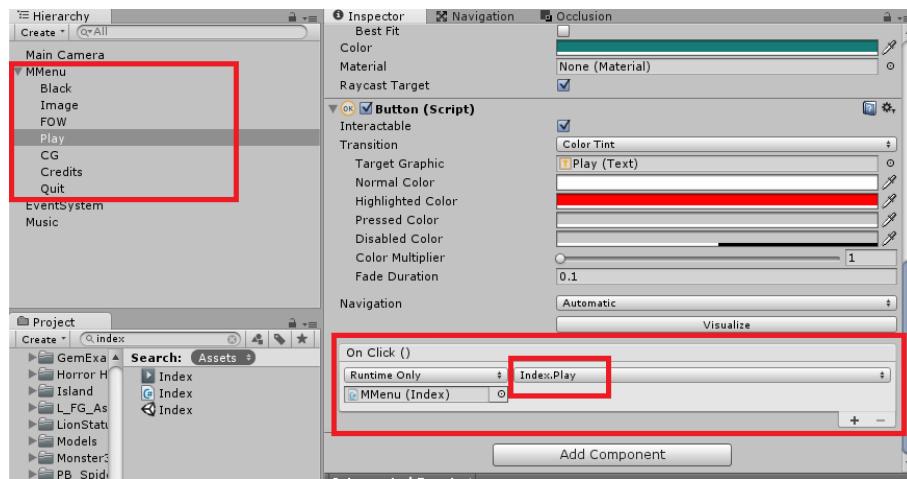


Fig. 3.29 – Objeto que representa el botón Play en el menú de inicio y tiene un componente Button.

```

public void Play()
{
    SceneManager.LoadScene("ModoDeJuego");
}//Play

public void Character()
{
    SceneManager.LoadScene("Character");
    DontDestroyOnLoad(music);

}//Character

public void Credits()
{
    SceneManager.LoadScene("CreditsNew");
    DontDestroyOnLoad(music);

}//Credits

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

```

Fig. 3.30 – Fragmento del script adjunto al objeto llamado Play

La Fig. 3.31 pertenece a la escena que pretende ser una guía del personaje. Ésta muestra sus controles y es la primera escena que aplica realidad aumentada. La imagen de Galatea es una captura de pantalla que luego fue subida al *Target Manager* de Vuforia (se verá con más detalle esto en el capítulo correspondiente a la aplicación Android).

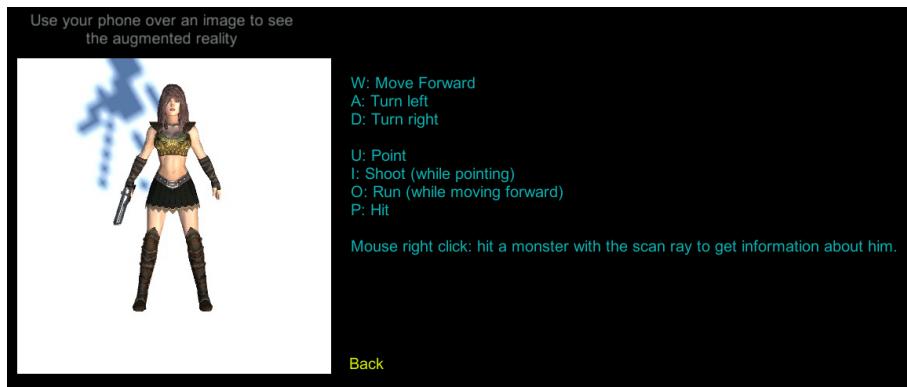


Fig. 3.31 – Escena Character

La Fig. 3.32 muestra la escena de los créditos. Tanto esta escena, como la de la guía del personaje, tienen un botón “Back” que conecta nuevamente con la escena del menú principal.



Fig. 3.32 – Escena Créditos

Hasta el momento, se vió como conectar una escena con otra por medio de la pulsación de una tecla. Para las escenas 3D, que son los niveles “jugables”, conectamos la escena siguiente de forma distinta. Un caso es mediante la detección de un objeto (personaje) “invadiendo” en el espacio de otro (llave de la Fig. 3.33).



Fig. 3.33 – Escena propia del juego

Estos tipos de objetos tienen una propiedad en particular, que son *triggers* (en español esto puede ser traducido a disparadores). Para ello, un componente tipo *Collider* (Fig. 3.34 y Fig. 3.35) con la propiedad *Is trigger* activada es añadido en el objeto que tiene como función conectar la siguiente escena al recibir la instrucción desde un *script* también adjunto al objeto.

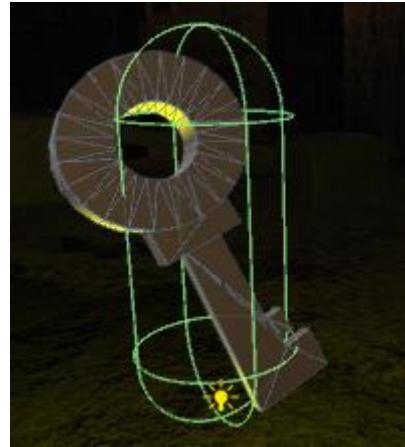


Fig. 3.34 – Modelo 3D del objeto llave. La capsula verde que se ve es su componente *Capsule Collider*

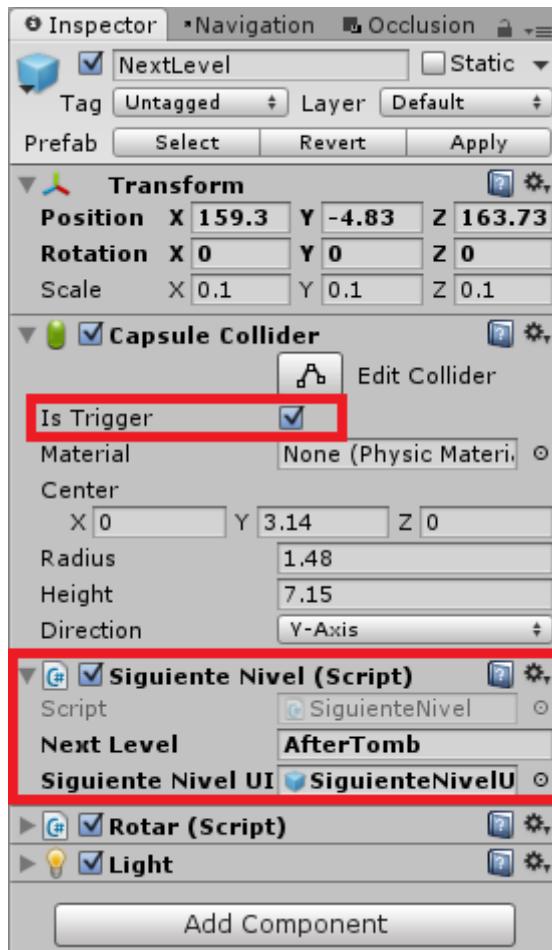


Fig. 3.35 – El objeto y sus componentes

Los componentes *Collider* definen la forma de un objeto para los propósitos de colisiones físicas. Los *colliders* más simples (y menos intensivos al procesador) son los llamados *colliders* primitivos. En 3D, estos son *Box Collider*, *Sphere Collider* y *Capsule Collider* como se puede ver en la Fig. 3.36.

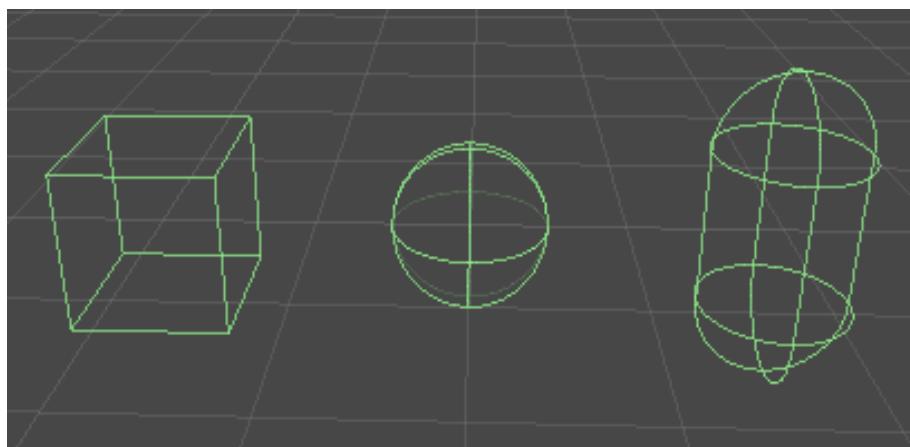


Fig. 3.36 – Colliders primitivos

Se pueden utilizar *Mesh Colliders* (Fig. 3.37) para encajar la figura de la malla del objeto exactamente. Estos colliders son mucho más intensivos al procesador que los tipos primitivos, no obstante, se pueden utilizar de manera escasa para mantener un buen rendimiento.

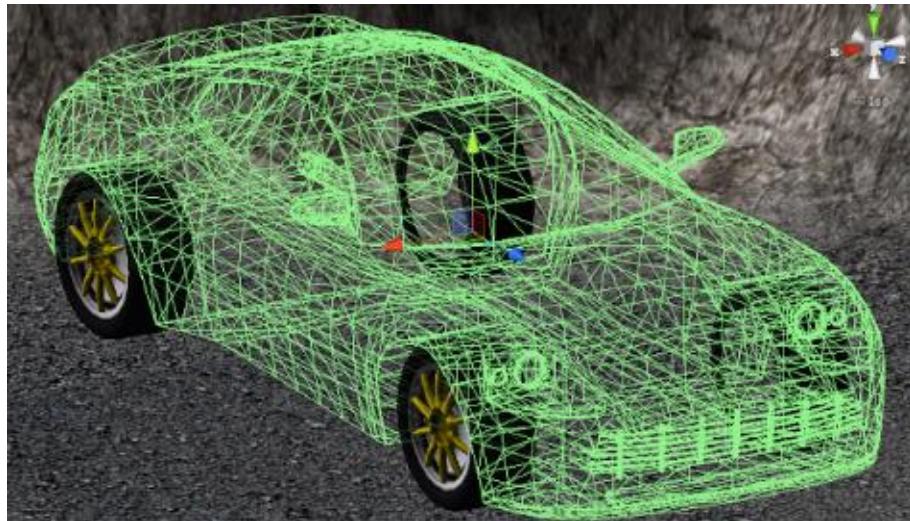


Fig.3.37 – Mesh collider

El sistema de *scripting* puede detectar cuando suceden colisiones e instanciar acciones. Cuando la colisión ocurre, el motor de física llama a las funciones con nombres específicos en cualquier script adjunto a los objetos involucrados. Sin embargo, se puede también utilizar el motor de física simplemente para detectar cuando un collider entra al espacio de otro sin crear una colisión. Un collider configurado como *Trigger* (utilizando la propiedad *Is Trigger*) no se comporta como un objeto sólido y simplemente le permitirá a otros colliders pasar a través de él. Cuando un collider entra en su espacio, un *trigger* va a llamar la función *OnTriggerEnter* en los *scripts* del *trigger* del objeto (Fig. 3.38).

```

void OnTriggerEnter(Collider colliderImpactado)
{
    //Debug.Log("OnTriggerEnter");
    if (colliderImpactado.CompareTag("Personaje"))
    {
        //Debug.Log("Dentro del if");

        animadorDarkScreen = siguienteNivelUI.GetComponent<Animator>();
        animadorDarkScreen.SetTrigger("PantallaNegraTexto");

        timerON = true;
    }
}

//if
}//OntriggerEnter

// Update is called every frame, if the MonoBehaviour is enabled
public void Update()
{
    if (timerON) timer += Time.deltaTime;

    if (timer >= 6f) SceneManager.LoadScene(nextLevel);
}

```

Fig.3.38 – Script adjunto

Resta aclarar que Unity utiliza un sistema de etiquetas el cuál facilita distinguir objetos (Fig. 3.39). La condición a cumplirse en esta escena es que el objeto cuya etiqueta es “Personaje” sea el cuál ha “invadido” el *collider* de la llave.

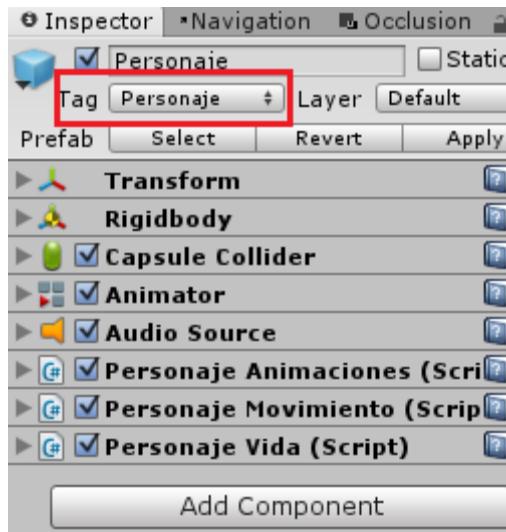


Fig.3.39 – El objeto personaje en una de las vistas de la herramienta Unity. Se puede ver en el recuadro rojo que éste tiene la etiqueta “Personaje”

El otro caso dentro de estos niveles jugables es cuando la misión del mismo consiste en recolectar 100 puntos de cristales. Un *script* adjunto al personaje controlará cuadro a cuadro (función *Update*) si la variable estática que controla el conteo de los cristales es mayor o igual que 100 (Fig. 3.40 y Fig. 3.41).



Fig. 3.40 – Escena del juego. El objetivo es recolectar cristales

```
// Update is called once per frame
void Update()
{
    puntos = ConteoCristales.cantidadCristalesAzul;

    if (puntos >= 100f)
    {
        tiene100Puntos = true;
        animadorDarkScreen.SetTrigger("PantallaNegraTexto"); animator>();
        timerON = true;
    }

    //SceneManager.LoadScene(loadNextScene);
    } //if

    if (puntos >= 1000f) tiene1000Puntos = true;

    if (timerON) timer += Time.deltaTime;
}

if (timer >= 6f) SceneManager.LoadScene(loadNextScene);

//Debug.Log(tiene1000Puntos);
} //Update
```

Fig. 3.41 – Fragmento del script adjunto al personaje

En resumen, se vió como el juego en sí se compone de una secuencia de escenas las cuales se conectan mediante scripts adjuntos que controlan la condición para hacerlo. Algunas condiciones serán con un control cuadro a cuadro del pulso de una tecla (función *Update*), otras al detectar contacto (función *OnTriggerEnter*) y otras al hacer click en un botón.

3.3.3 Personaje

Como se mencionó al principio de este capítulo al hablar del diseño del personaje, se explicó cómo se obtuvo el modelo 3D desde el Asset Store de Unity. Luego de obtenerlo, se creó su historia y comportamiento. Sin embargo, más allá de haber obtenido el modelo 3D hecho por otro artista, el personaje no hace nada por sí solo, y en escena no es más que un modelo 3D inanimado. Un objeto el cual tiene su componente *Transform* para existir en el universo.

Pues bien, también se vió en la Fig. 3.12 un texto que indicaba que el personaje camina, corre, dispara, golpea y tira un rayo que escanea ciertos objetos (monstruos y niveles). Se debe pensar entonces, ¿qué componentes necesita el objeto el cual representa el personaje para que se cumplan todas esas condiciones? (algunos componentes se utilizan para algunas escenas, otros no). La Fig. 3.42 muestra el listado de componentes del objeto Personaje y la Fig. 3.43 a cómo se ve el modelo 3D con sus componentes en el modo editor.

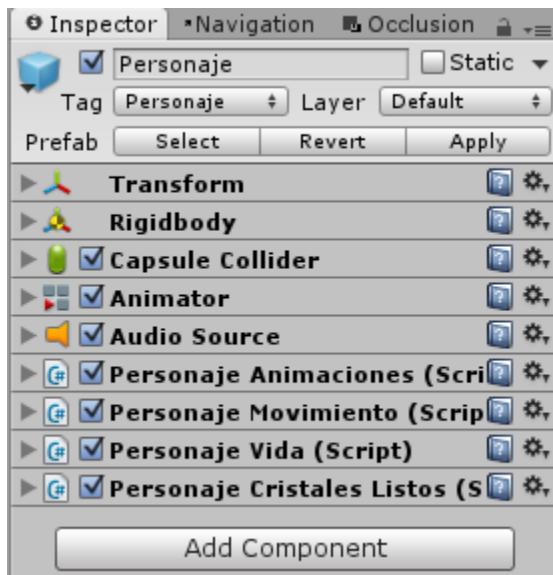


Fig. 3.42 – El objeto personaje y sus componentes

A simple vista se puede notar lo siguiente de la Fig. 3.42:

- Una etiqueta “Personaje”.
- Al igual que todo *GameObject* en escena, tiene un componente *Transform* para indicar las coordenadas (X,Y,Z) del universo donde se encuentra el objeto. También para indicar su rotación y su escala.
- Componentes *Rigidbody* y *Capsule Collider*, por lo que este objeto ya tiene presencia física y responde a las leyes de la gravedad por ejemplo.
- Un componente *Animator* el cual tiene una *state machine* (máquina de estados) para manejar las animaciones (las instrucciones las recibe desde los *scripts* que refieren al componente *Animator*).

- Un componente *Audio Source* para el sonido de pasos al caminar (controlado por los *scripts* también).
- Cuatro *scripts*. La mayoría conectados entre sí, algunos conectados con *scripts* de otros objetos. Un *script* es para las animaciones, otro para el movimiento, otro para la vida del personaje y otro para el conteo de recolección de cristales (estará desactivado en niveles que no se lo utilice).

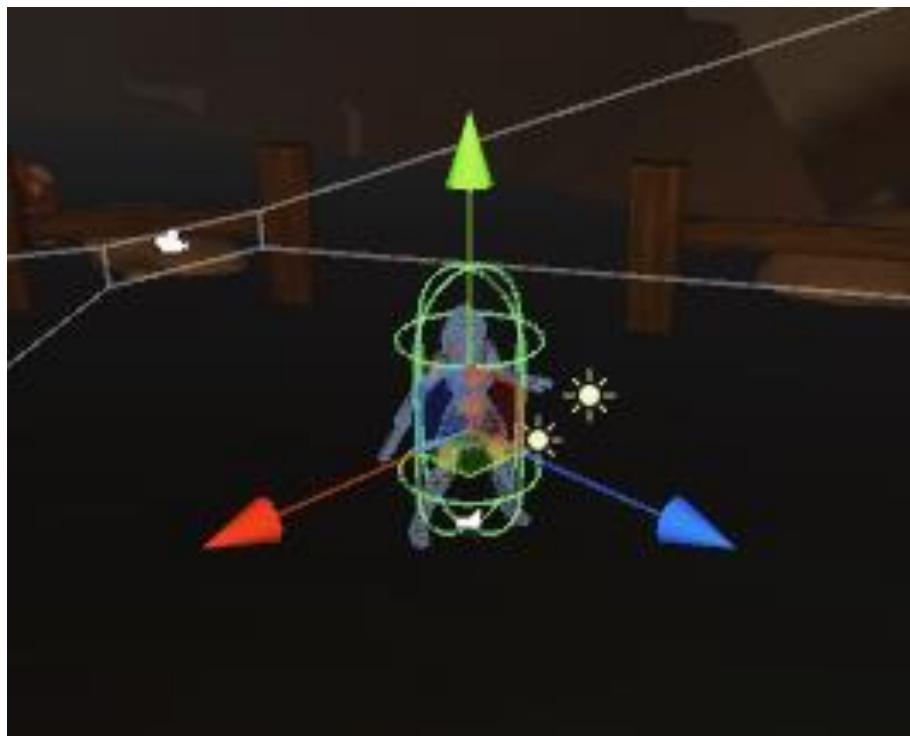


Fig. 3.43 – Personaje

Para lograr que el personaje se mueva, primero que nada se debe arrancar por conceptos de física. Lo que se hace es aplicar una fuerza hacia delante (adelante es la dirección positiva del eje Z, que es el eje azul de la Fig. 3.43), ya que este objeto tiene un componente cuerpo rígido (*Rigidbody*) y uno del tipo *Collider*. Luego, desde el *script* adjunto llamado “PersonajeMovimiento”, en la función *Update* (que es llamada en cada *frame*), controlará si la tecla correspondiente está siendo pulsada y así llamar a la función que aplique la fuerza al cuerpo rígido (Fig. 3.44).

```

} //Update

void Mover()
{
    |
    // Create a vector in the direction the tank is facing with a magnitude based on the input, speed
    Vector3 movimiento = transform.forward * moverEntradaTeclado * moverVelocidad * Time.deltaTime;

    // Apply this movement to the rigidbody's position.
    cuerpoRigidoPersonaje.MovePosition(cuerpoRigidoPersonaje.position + movimiento);
} //Avanzar
}

void Girar()
{
}

```

Fig. 3.44 – Fragmento del script encargado del movimiento del personaje. En el recuadro rojo se puede ver la función Mover

Pero con solo esto no basta, ya que, si bien se está desplazando y rotando el modelo 3D sobre el eje de coordenadas global, este no está animado, por lo que daría la sensación de un modelo 3D “patinando” en el suelo. Para esto se utiliza el componente *Animator*.

La Fig. 3.45 muestra la máquina de estados del componente *Animator* adjunto al *GameObject* Personaje. La pestaña *Base Layer* muestra los distintos estados (cada cuadrado es un estado y contiene un *clip* de animación). El panel de la izquierda contiene los parámetros creados (pueden ser de tipo entero, punto flotante, booleano y *trigger*) para pasar de un estado a otro.

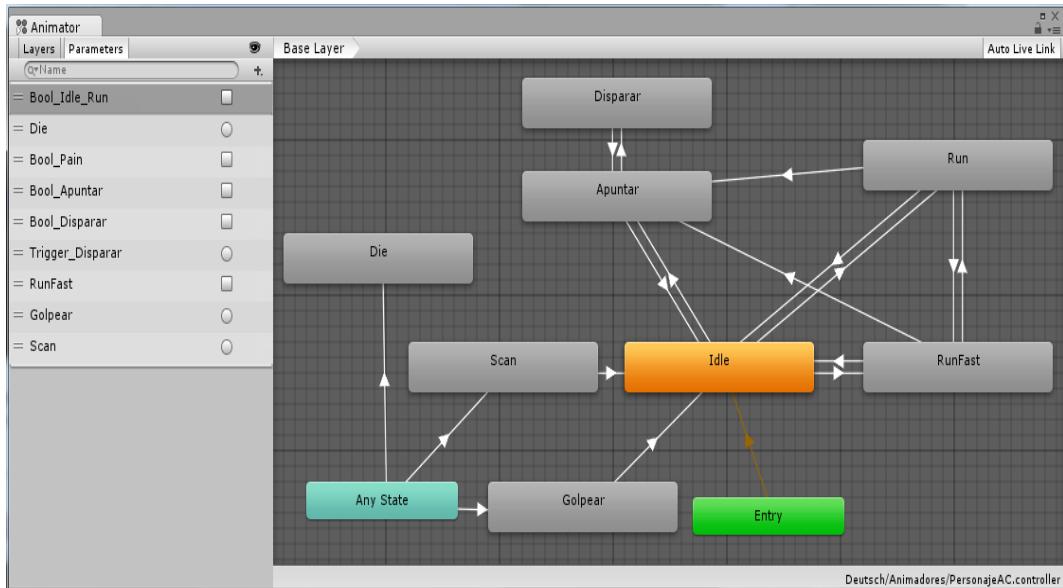


Fig. 3.45 – Máquina de estados de Galatea

Ejemplo relacionado a la Fig. 3.45: la variable velocidad del *script* que controla el movimiento del personaje tiene un cierto valor entero. Mientras el valor de esa variable permanezca así, el estado actual de la máquina de

estados será “Run”, mientras que si se mantiene presionada la tecla “O”, el personaje empezará a correr más rápido, por lo que su variable aumentará el valor (el *script* que controla el valor de la variable correr detectará cuando la tecla “O” esté pulsada). Entonces, de la misma forma, el parámetro que determina cuando pasar de un estado a otro según su valor, indicará que se debe transicionar al estado “RunFast”. Un fragmento del código que controla esa condición puede verse en la Fig. 3.46.

```
void Animar()
{
    //bool corriendo;
    //Debug.Log("AnimarPersonaje");

    if (moverEntradaTeclado > 0)
    {
        if (Input.GetKey("o") && personajeUI_Script.personajeSliderCorrer.value >= 1)
        {
            moverVelocidad = 6;
            personajeAnimaciones.personajeEstaCorriendoRapido = true;
            personajeUI_Script.personajeSliderCorrer.value -= 5;

            if (personajeUI_Script.personajeSliderCorrer.value == 0)
            {
                moverVelocidad = 3;
                personajeAnimaciones.personajeEstaCorriendoRapido = false;
                personajeAnimaciones.personajeEstaCorriendo = true;

                //personajeUI_Script.personajeSliderCorrer.value += 1;
            }//if
        }
    }
}
```

Fig .3.46 – Función Animar dentro del script que controla el movimiento del personaje. En el recuadro rojo podemos ver el control de la condición de la tecla “O” pulsada y la referencia a la máquina de estados de animaciones del personaje

Con lo mencionado recién, se puede ver como los *scripts* están relacionados unos con otros todo el tiempo, ya que, el *script* que controla el movimiento tiene relación con el que controla las animaciones. A su vez, los *scripts* tienen que referenciar a los componentes todo el tiempo, se puede ver en el recuadro rojo de la Fig. 3.47 que el script encargado de las animaciones necesita referenciar al *script* que controla el movimiento como al componente *Animator*.

```

public class PersonajeAnimaciones : MonoBehaviour
{
    public GameObject puntaPistola;
    InstanciarBala instanciarBala;
    Light luzDisparo;

    public GameObject puntaPistolaPegar;

    public Animator personajeAC;
    PersonajeMovimiento personajeMovimientoScript;

    public AudioSource golpear;

    // Hago esta variable pública así la manejo desde el script que
    public bool personajeEstaCorriendo;
    public bool personajeEstaCorriendoRapido;
    public float personajeVelocidadAtaque = 1f;

    // Esta variable la controlaré desde este Update
    public bool personajeApuntar;
    bool presionarTeclaApuntar;
}

```

Fig. 3.47 – Fragmento del script PersonajeAnimaciones

Lo que el personaje hace al disparar es instanciar objetos que representan las balas desde una determinada posición, esto es, un objeto hijo del personaje ubicado a un distancia delante de él (la cuál encaje con la distancia de sus brazos a la hora de realizar la animación correspondiente de apuntar el arma) como punto de referencia el cuál será instanciado el objeto bala y luego impulsado hacia adelante con una fuerza en dirección del eje Z positivo. Para este caso, se desactiva la propiedad “gravedad” en el componente *Rigidbody* de la bala.

En la Fig. 3.48 se puede ver al personaje en el momento que realiza la animación de apuntar al presionar la tecla “U”. A su vez, en la ventana *Hierarchy* (ventana propia de la herramienta Unity) uno de los objetos hijos de Personaje, este es el objeto PuntaPistola, el cual está ubicado a cierta distancia del Z delante del personaje, y, al ser hijo del mismo, todo el tiempo su posición y rotación será relativa a las coordenadas de su padre y no a las globales.



Fig. 3.48 – Galatea apuntando el arma

La posición del objeto PuntaPistola será donde el objeto Bala será instanciado cada vez que se presione la tecla “I”. El objeto que se está instanciando que representa la bala es el que se muestra a continuación en la Fig. 3.49:

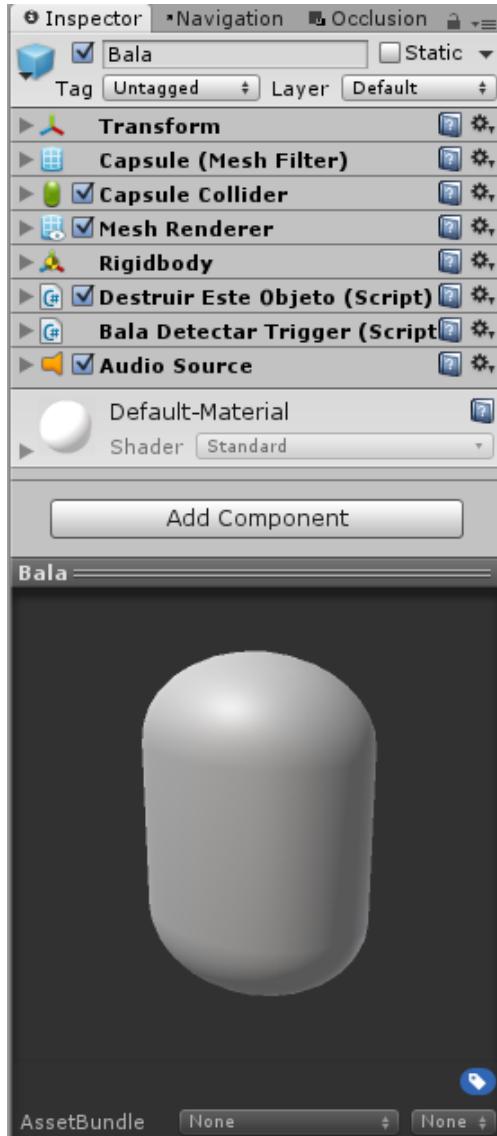


Fig. 3.49 – El objeto Bala

Se trata de uno de los objetos 3D primitivos (una cápsula), al cuál se le adjuntaron *scripts* para que actúe de una determinada manera, como por ejemplo, que se le aplique una fuerza en dirección al eje Z positivo. Otro comportamiento propio es el de autodestruirse en caso de impactar un enemigo. También se autodestruirá si no impacta con nada, pues, no es deseable que el obeto esté “viajando” infinitamente en el tiempo de ejecución, es una carga innecesaria para el *CPU*.

Y dentro del objeto el cuál se comporta como la pistola (PuntaPistola), está adjunto el *script* (*InstanciarBala*) adjunto encargado de que el personaje dispare en el juego (Fig. 3.50).

```

public void Disparar()
{
    Rigidbody rocketInstance;
    rocketInstance = Instantiate(bala, puntaPistola.position , puntaPistola.rotation) as Rigidbody;

    rocketInstance.AddForce(puntaPistola.forward * 5000);
    sonidoDisparo.Play();

    PersonajeBalas.cantidadBalas -= 1;
}//Disparar

```

Fig. 3.50 – La función Disparar dentro del script InstanciarBala

Se puede ver en la Fig. 3.51 componentes agregados (*Light* y *Audio Source*). Estos son solamente para efectos visuales (luz y sonido de disparo) y se controlan también desde el script *InstanciarBala*.



Fig. 3.51 – Objeto PuntaPistola

Al presionar la tecla “P”, el personaje empuja (o golpea). Esta habilidad tiene como objetivo que el personaje pueda tomar distancia rápidamente de un enemigo que actualmente se encuentra atacándolo. Para lograr esto, un objeto hijo adjunto al personaje, que tiene un componente *Sphere Collider*, producirá una fuerza de empuje para disipar a los enemigos atacantes. Este objeto hijo del personaje está desactivado, y se activa al pulsar la tecla “P”. En la Fig. 3.52 se muestra lo ya mencionado.

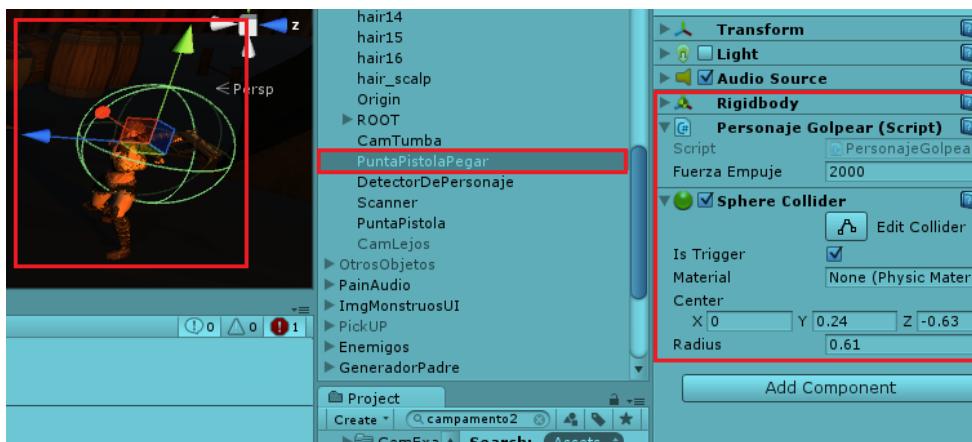


Fig. 3.52 – El personaje usando la habilidad golpear o empujar al teclar la tecla “P”

Vale mencionar que esta habilidad tendrá efecto solo en objetos cuya etiqueta sea “Enemigo” o “Enemigo2” como muestra el código de la Fig. 3.53. También reproducirá una animación perteneciente a uno de los estados de la *state machine* del componente *Animator* del personaje:

```
//Debug.Log(colliderImpactado);
if (colliderImpactado.CompareTag("Enemigo") || colliderImpactado.CompareTag("Enemigo2"))
{
    EnemigoVida enemigoVida = colliderImpactado.GetComponent<EnemigoVida>();

    //Debug.Log(enemigoVida);

    if (enemigoVida != null)
    {
        //enemigoVida.enemigoVidaActual = enemigoVida.enemigoVidaActual - DañoBala /* - Ra
        Rigidbody enemigoCuerpoRigido = colliderImpactado.GetComponent<Rigidbody>();

        Vector3 empujar = transform.forward;
        enemigoCuerpoRigido.AddForce(empujar * fuerzaEmpuje);

        Animator enemigoAC = colliderImpactado.GetComponent<Animator>();
        enemigoAC.SetTrigger("Pain");
    }
}
```

Fig. 3.53 – Fragmento del código adjunto al objeto encargado de realizar la habilidad

3.3.4 Enemigos

Al igual que el personaje, los enemigos también fueron descargados del *Asset Store* de Unity, así como algunos niveles, decoraciones e ítems. Los enemigos de este juego son 4: un *zombie* (Fig. 3.54), un esqueleto (Fig. 3.55), una araña (3.56) y un golem (Fig. 3.57). Todos comparten la mayoría de componentes y las configuraciones de los mismos, pues, más allá que el modelo 3D sea un esqueleto o un zombie, no quita el hecho de que persiga al personaje y lo ataque. Algunos enemigos hacen más daño que otros, otros son más rápidos, y otros tienen más puntos vitales. La aplicación para dispositivo móvil cumple la función de informar que atributos tiene cada uno. También vale recordar que el modelo 3D del jefe final del juego es el mismo que el del personaje, con la particularidad que tiene algunos cambios visuales como por ejemplo el no tener cabeza y usar una espada. Esto quiere decir que, el modelo 3D tiene como objetos hijos sus armas y partes del cuerpo, lo que se hizo fue desactivar los objetos hijos que eran parte la cabeza del modelo, como así también sustituir la pistola por una espada.

Zombie

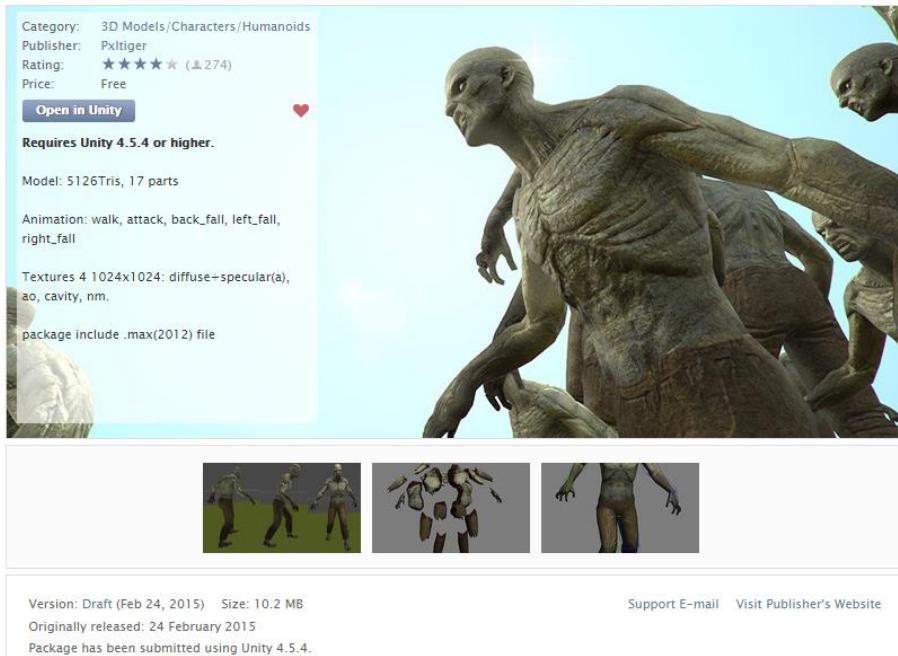


Fig. 3.54 – Zombie: <https://www.assetstore.unity3d.com/en/#!/content/30232>

Fantasy Monster – Skeleton

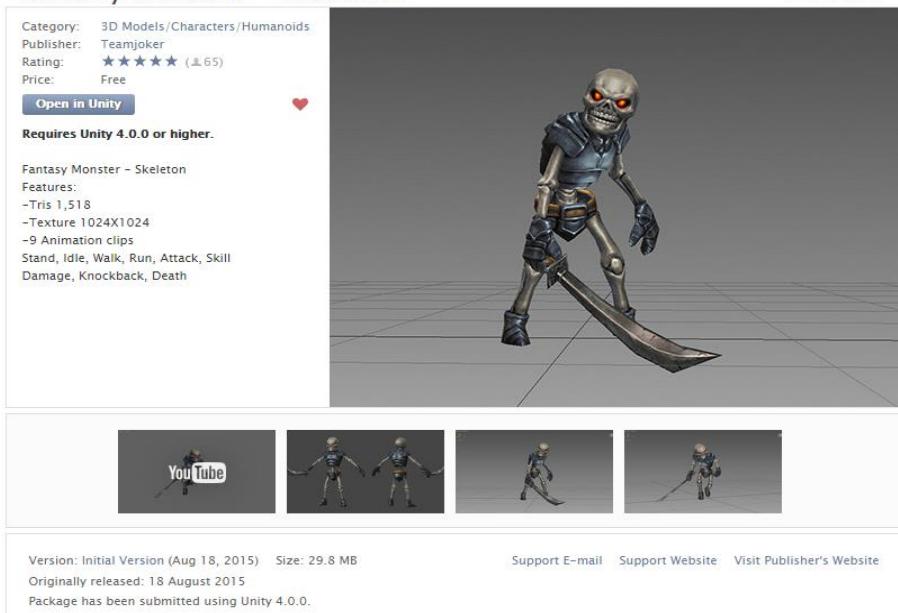


Fig. 3.55 – Esqueleto: <https://www.assetstore.unity3d.com/en/#!/content/35635>

Animated Spider

Category: 3D Models/Characters/Animals/Insects
 Publisher: [prism bucket]
 Rating: ★★★★☆ (127)
 Price: Free

[Open in Unity](#)

Requires Unity 4.5.4 or higher.

Animated spider

-924 tris
 -animations: idle, walk, jump, attack, die

[Preview here.](#)



Version: 1.0 (Feb 14, 2015) Size: 773.2 kB
 Originally released: 15 October 2014
 Package has been submitted using Unity 4.5.4.

[Support E-mail](#) [Support Website](#) [Visit Publisher's Website](#)

Fig. 3.56 – Araña: <https://www.assetstore.unity3d.com/en/#!/content/22986>

Monster 3

Category: 3D Models/Characters/Creatures
 Publisher: BüMSTRUM
 Rating: ★★★★☆ (133)
 Price: Free

[Open in Unity](#)

Requires Unity 5.2.0 or higher.

creature model with 17 Animations
 tris = 5802
 textures = 2048*2048 in .png format



Version: 1.0 (Nov 05, 2015) Size: 79.3 MB
 Originally released: 5 November 2015
 Package has been submitted using Unity 5.2.0.

[Support E-mail](#) [Support Website](#) [Visit Publisher's Website](#)

Fig. 3.57 – Golem: <https://www.assetstore.unity3d.com/en/#!/content/48933>

A diferencia del personaje, los enemigos no se mueven por medio de fuerzas físicas, sino a través de áreas navegables. La Fig. 3.58 muestra los componentes de un enemigo en el juego:



Fig. 3.58 – Componentes de un objeto Enemigo (el esqueleto en este caso)

Se puede observar en la Fig. 3.58 que aparece un nuevo componente, este es, el *Nav Mesh Agent*, y es el que hay que agregar a todo objeto que se mueva dentro de un área navegable en un nivel. El área navegable se configura previamente, creando zonas las cuales son “caminables” por los agentes, es decir, las zonas por donde pueden desplazarse en la escena. En este proyecto, los agentes a menudo están patrullando, la Fig. 3.59 muestra un ejemplo de cómo se ve un área navegable mientras se trabaja en el juego:

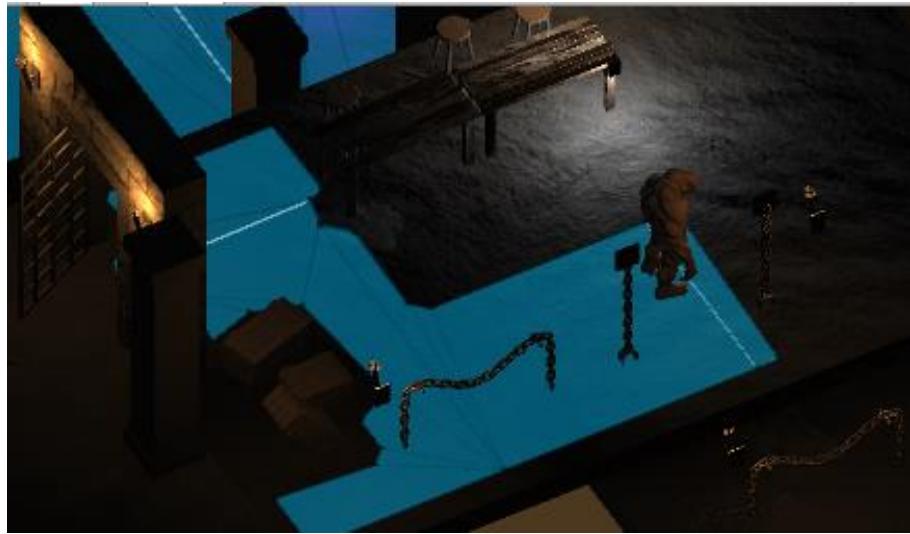


Fig. 3.59 – El área celeste indica que es un área navegable para los agentes. Este ejemplo muestra al enemigo Gólem (agente) situado en ella

Se mencionó que los enemigos no se mueven por aplicación de fuerza física sino gracias al área navegable. Sin embargo, esto no basta, pues, los enemigos necesitan una *IA* (inteligencia artificial) para saber a qué coordenada (x,y,z) de la escena deben dirigirse en determinado instante de tiempo. Este comportamiento también es controlado desde los *scripts*. El comportamiento de los enemigos es el siguiente: patrullar hasta detectar al personaje, y una vez detectado, comenzar a perseguirlo hasta alcanzar una distancia lo suficientemente cercana al mismo para luego detenerse y atacarlo.

Al igual que el personaje, el enemigo también utiliza un componente *Animator* para reproducir los clips de animaciones propios del modelo 3D. La Fig. 3.60 muestra la máquina de estados del enemigo:

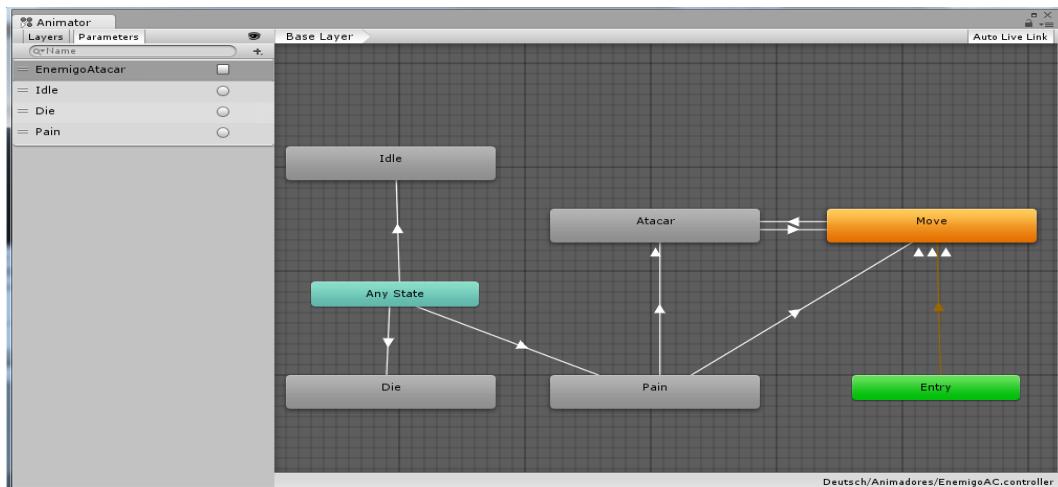


Fig. 3.60 – Máquina de estados de un enemigo

La máquina de estados para las animaciones de los enemigos es exactamente la misma para los 4 pertenientes del proyecto (*zombie*,

esqueleto, gólem y araña) y para el jefe final. Lo único que difiere una de la otra son los *clips* de animaciones que contiene cada estado, ya que la animación “caminar” de un *zombie* no es la misma que la de una araña, cada una viene con su respectivo modelo 3D. Si se quisiera descargar un nuevo modelo 3D desde el *Asset Store* para sumar otro enemigo, la máquina de estados seguiría siendo la misma, simplemente habría que colocar el clip de animación correspondiente (que en teoría debería incluir el paquete importado del modelo) en el estado correspondiente.

La *IA* del enemigo a la hora de patrullar tuvo varios cambios, la primera vez que se programó, consistía en brindar, dentro del rango del área navegable, valores aleatorios (x,y,z) en cada cuadro a la función que indicaba hacia donde tenía que dirigirse el enemigo (función *SetDestination(vector3)*). Esto se vió modificado ya que dentro del juego a veces perdía coherencia el movimiento del enemigo, por lo que se modificó de 2 maneras de acuerdo al nivel:

- Método 1: se toman dos puntos del espacio y cada cierto tiempo el enemigo se desplaza de un punto a otro.
- Método 2: se toma un punto en el espacio y un valor aleatorio. Cada cierto tiempo el enemigo se desplaza de un punto a otro.

Así, se logra un poco más de coherencia a la hora de ver en el juego a un enemigo patrullando de un punto a otro.

3.3.5 Personaje vs Enemigos

Unity tiene un sistema de etiquetas para reconocer cierto objeto en particular (o cierto grupo de objetos similares). Esto es bastante útil para muchos tipos de interacciones entre objetos, por ejemplo: entre el personaje y un enemigo o un objeto recolectable (por ejemplo un item que restaure parte de los puntos vitales del personaje). En particular, el personaje del proyecto utiliza la etiqueta “Personaje” y cualquier enemigo ya sea un *zombie* o una araña utiliza la etiqueta “Enemigo”.

El personaje dispara balas y estas, solo tienen efecto si colisionan en objetos cuya etiqueta es “Enemigo”. De la misma forma, un enemigo deja de patrullar y se desplaza hacia un punto (x,y,z) del espacio donde se encuentre un objeto que tenga la etiqueta “Personaje” (además de atacarlo).

El personaje y un enemigo deben tener algún *script* adjunto que contenga una variable pública que represente sus puntos vitales. Así, una bala instanciada hará efecto en un enemigo accediendo a su *script* y restando el valor de dicha variable, de igual forma lo hará el enemigo que ataca al personaje. Ambos se reconocen por medio de etiquetas al colisionar y acceden a sus respectivas variables para restarles valor. A continuación un fragmento del script adjunto al objeto Bala (3.61):

```

void OnTriggerEnter(Collider colliderImpactado)
{
    //Debug.Log(colliderImpactado);
    if (colliderImpactado.CompareTag("Enemigo") || colliderImpactado.CompareTag("Enemigo2"))
    {

        EnemigoVida enemigoVida = colliderImpactado.GetComponent<EnemigoVida>();

        //Debug.Log(enemigoVida);

        if (enemigoVida != null)
        {

            enemigoVida.enemigoVidaActual = enemigoVida.enemigoVidaActual - DañoBala - Random.Range(1f, 10f);

            Rigidbody enemigoCuerpoRígido = colliderImpactado.GetComponent<Rigidbody>();

            Vector3 empujar = transform.forward;
            enemigoCuerpoRígido.AddForce(empujar * 100);

            Animator enemigoAC = colliderImpactado.GetComponent<Animator>();
            enemigoAC.SetTrigger("Pain");

            if (enemigoVida.enemigoVidaActual <= 0) sonidoMounstruo.Play();

        }
    }
}

```

Fig. 3.61 – Fragmento de código del *script* adjunto al objeto Bala

Este fragmento de código pertenece al *script* adjunto a una bala instanciada y consiste en lo siguiente:

- Recuadro rojo: La condición para que tenga efecto de daño la bala al momento de impactar sobre otro objeto es que este último tenga la etiqueta “Enemigo” o “Enemigo2”.
- Recuadro Anaranjado: Acceder a la variable que representa los puntos vitales del enemigo y decrementar su valor. El valor a decrementar es un valor base propio de la bala establecido en la variable DañoBala más un valor aleatorio entre 1 y 10 (punto flotante).
- Recuadro Verde: al momento de impactar, la bala aplica una pequeña fuerza que empuja al enemigo hacia atrás, además se activa la animación propia del enemigo que simula dolor.
- Recuadro Azul: Si la variable que controla los puntos vitales del enemigo es menor a 0, reproduce un sonido (un grito de muerte del enemigo).

Ahora bien, la manera que utiliza el personaje para atacar es disparando balas, los enemigos atacan a corta distancia y no disparan proyectiles. El enemigo detecta un objeto cuya etiqueta es “Personaje” y comienza a dirigirse hacia el punto (x,y,z) donde se encuentra en ese instante de tiempo. La pregunta es ¿Cómo lo detecta?. Observar la Fig. 3.62 antes de entrar en detalle ayudará a comprender mejor el mecanismo.

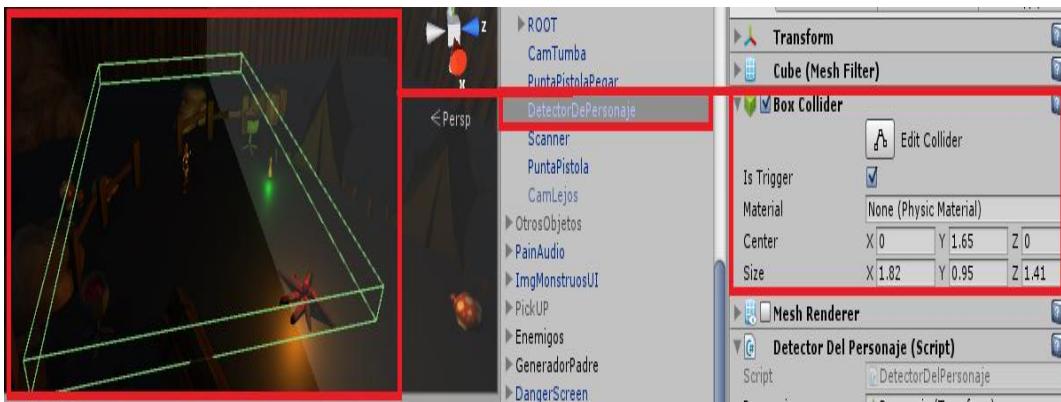


Fig. 3.62 – Objeto hijo del personaje llamado “DetectorDelPersonaje”

Fig. 3.62: El objeto *DetectorDelPersonaje* es un objeto hijo del *Personaje*. El mismo está ubicado exactamente en la posición del padre y tiene la siguiente particularidad, un componente *BoxCollider* inmenso que le rodea (es invisible en el juego) con la propiedad *Is Trigger* activada. La idea de este componente gigante rodeándolo es que, si un objeto cuya etiqueta sea “Enemigo” entra dentro de esa zona, se dispara una acción. Esta acción es la siguiente, buscar en el objeto enemigo el *script* que controla su movimiento de patrullar y desactivarlo. A continuación, indicarle que el punto (x,y,z) destino al cuál se tiene que dirigir es hacia donde se encuentre el personaje actualmente en ese cuadro. Todo esto ocurrirá mientras el objeto enemigo permanezca dentro de la zona envolvente de esa caja con bordes verdes que muestra la Fig. 3.62. A continuación, un fragmento del código adjunto al objeto *DetectorDelPersonaje* (Fig. 3.63):

```
void OnTriggerEnter(Collider colliderImpactado)
{
    if (colliderImpactado.CompareTag("Enemigo"))
    {
        EnemigoPatrullar enemigoPatrullarScript = colliderImpactado.GetComponent<EnemigoPatrullar>();
        //EnemigoMovimientoNavMesh enemigoMovimientoNavMeshScript = colliderImpactado.GetComponent<EnemigoMovimientoNavMesh>();
        NavMeshAgent enemigoNavMeshAgent = colliderImpactado.GetComponent<NavMeshAgent>();

        if (enemigoPatrullarScript != null & enemigoNavMeshAgent != null)
        {
            enemigoPatrullarScript.enabled = false;
            enemigoNavMeshAgent.SetDestination(personaje.position);
        }
    }
}
```

Fig. 3.63 – Fragmento del script adjunto al objeto *DetectorDelPersonaje*

Ahora la función es *OnTriggerStay* a diferencia de *OnTriggerEnter* que tenía la Bala, puesto que la condición se debe cumplir mientras el enemigo permanezca dentro del collider que envuelve al personaje y no solo cuando entra al mismo. El recuadro rojo muestra que se desactiva el componente *script* adjunto al enemigo y luego se indica que debe dirigirse hacia el punto (x,y,z) donde se encuentra el personaje actualmente (*función SetDestination(personaje.position)*).

Finalmente, si el personaje se aleja del enemigo una cierta distancia la cuál este último queda fuera de la zona envolvente, se activará nuevamente el componente script que tenía como función hacer patrullar al enemigo. Ahora la función a utilizar es *OnTriggerExit* y se muestra en la Fig. 3.64:

```
void OnTriggerExit(Collider colliderImpactado)
{
    if (colliderImpactado.CompareTag("Enemigo"))
    {
        EnemigoPatrullar enemigoPatrullarScript = colliderImpactado.GetComponent<EnemigoPatrullarScript>();
        enemigoPatrullarScript.enabled = true;
        enemigoPatrullarScript.enabled = true;
    }
}
```

Fig. 3.64 – Función *OnTriggerExit* perteneciente al mismo script

Una vez que el enemigo persigue al personaje, al acercarse a una distancia más próxima, detiene su marcha y comienza a atacarlo. Esto también es detectado por un *trigger collider*, esta vez adjunto al enemigo mismo y no al personaje. Esto se puede ver en la Fig.3.65 a continuación:



Fig. 3.65 –Vista de un objeto que representa un enemigo (Golem) en las ventanas *Scene*, *Hierarchy* e *Inspector* pertenecientes a la herramienta Unity

Es ahora el *script* *EnemigoAtacar* el cuál maneja esta interacción. Nuevamente es la función *OnTriggerStay* la que se usa, ya que el enemigo atacará al personaje mientras este se encuentre “invadiendo” el espacio del *collider* adjunto al enemigo (Fig. 3.66).

```
void OnTriggerEnter(Collider colliderImpactado)
{
    if (colliderImpactado.gameObject == personaje)
    {
        EnemigoFuncionAtacar();
    }
}//OnTriggerEnter
```

Fig. 3.66 – Fragmento del script EnemigoAtacar adjunto al GameObject Enemigo

El enemigo tiene una velocidad de ataque controlada por una variable que cumple la función de *timer*, atacando al personaje cada cierto tiempo. De no haber esto, el enemigo restaría puntos vitales del personaje cada cuadro y lo mataría casi instantáneamente. Es ahora el enemigo quién accede a la variable pública del *script* adjunto al personaje que controla sus puntos vitales (Fig. 3.67).

```
timer = 0f;
enemigoAnimacionesScript.EnemigoAnimacionesAtacar(true);
personajeVidaScript.personajeVidaActual = personajeVidaScript.personajeVidaActual - enemigoDaño - Random.Range(1f, 10f);

personajeUIScript.personajeSlider.value = personajeVidaScript.personajeVidaActual;
personajeUIScript.PintarPantallaRoja();
```

Fig.3.67 – Función EnemigoFuncionAtacar dentro del script EnemigoAtacar

3.3.6 Personaje y la UI

Normalmente, al jugar un videojuego 3D, también aparecen textos e imágenes en 2D. Esto es la *UI* o *User interface* (en español, la interfaz de usuario) y puede tener o no interacción directa con los controles de entrada del juego:

- Ejemplo *UI* interactiva: Un menú.
- Ejemplo *UI* no interactiva: Una barra de vida de un personaje.

Se vió anteriormente que este proyecto constaba con una serie de 14 escenas en 2D, como por ejemplo un menú. Sin embargo, las escenas 3D también tienen elementos de la *UI*, como por ejemplo la barra de vida del personaje, la estamina, la cantidad de balas y la referencia de los controles. La Fig. 3.68 muestra lo descriptorecientemente.

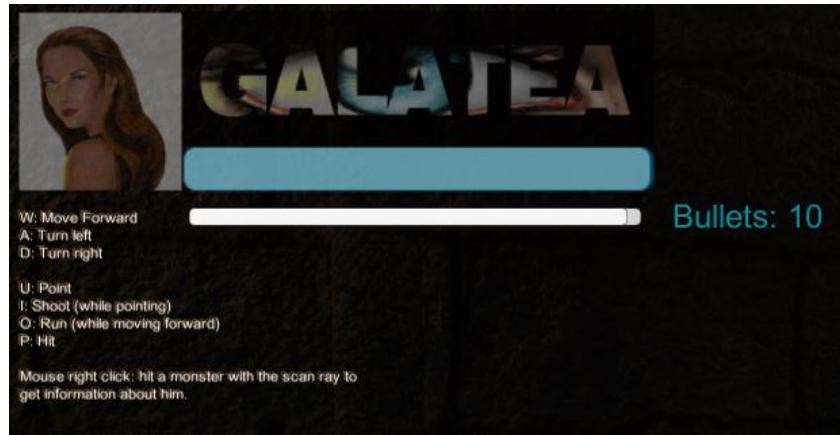


Fig. 3.68 – UI visible en todas las escenas 3D

Los elementos *UI* del juego son también objetos instanciados en escena. En Unity, este tipo de objetos es del tipo *Canvas*. En la Fig. 3.69 se muestra el objeto *Canvas* correspondiente a la *UI* visible en la Fig. 3.68:

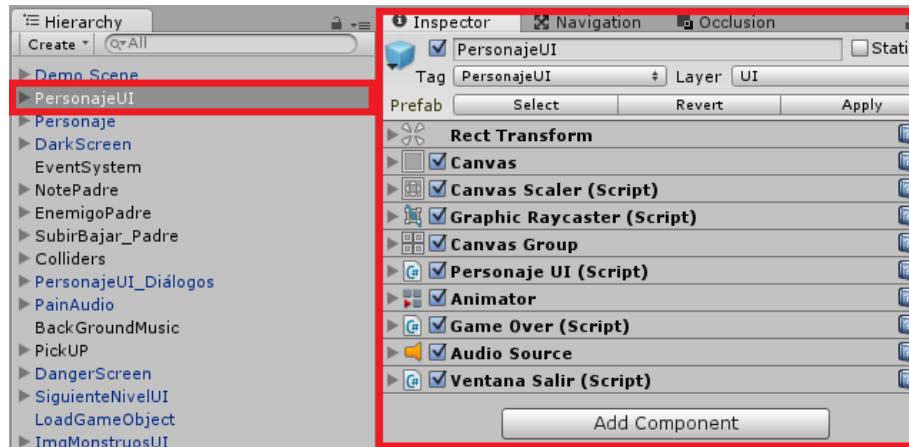


Fig. 3.69 – Objeto Canvas PersonajeUI

Este tipo de objeto (*Canvas*) no posee un componente *Transform*, puesto que no tiene una posición (x,y,z) dentro del espacio de la escena. La posición y el escalado es respecto a la pantalla y no al “mundo”, por lo que, su posición en este caso es en un eje de coordenadas (x,y) y no (x,y,z). La Fig. 3.70 muestra el objeto *PersonajeUI* con sus respectivos hijos:

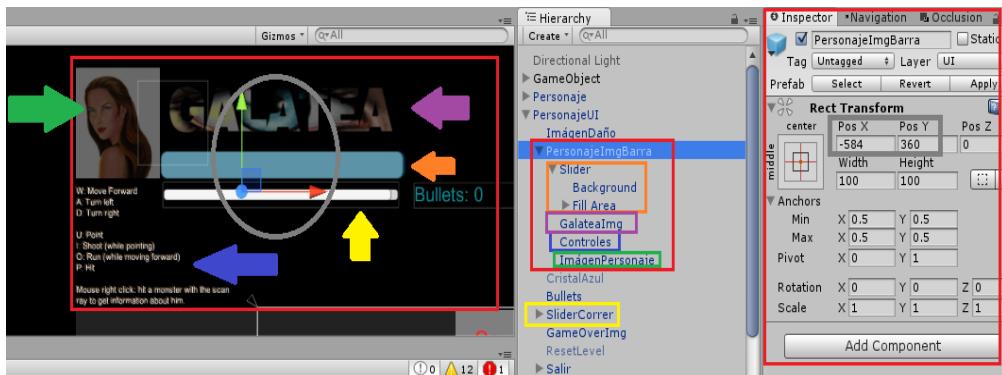


Fig. 3.70 – Objeto Canvas PersonajeUI

Detalles de la Fig. 3.70:

- Recuadro rojo: Objeto *Canvas* hijo *PersonajeImgBarra*, el cuál a su vez tiene otros hijos (también objetos de tipo *Canvas* claro).
- Recuadro y círculo gris: La posición en pantalla donde se encuentra ubicado el objeto *PersonajeImgBarra*.
- Recuadro Anaranjado: Objeto *Slider*. Es la barra de vida celeste que representa los puntos vitales del personaje.
- Recuadro violeta: Imagen (objeto tipo *Canvas*).
- Recuadro verde: Imagen (objeto tipo *Canvas*).
- Recuadro amarillo: Otro objeto *Slider*. En este caso es hijo del objeto padre *PersonajeUI*, pero no hijo de *PersonajeImgBarra*.

Algunos objetos *Canvas* pertenecientes a la *UI* tienen interacción con el personaje también, tal es el caso de ambos *Sliders* que aparecen en la Fig. 3.71. Uno representa los puntos vitales y el otro la estamina del personaje y sus respectivos valores son controlados desde una variable. También hay otros objetos *Canvas* similares que registran valores de variables como las balas del personaje y la cantidad de cristales que se consigue (aplicado solo en dos niveles). La particularidad de estas variables es que son estáticas, por lo que pertenecen a la clase y no la instancia y su valor siempre no variará de una escena a otra ya que al cargar una nueva escena, los objetos de la escena anterior son destruidos.

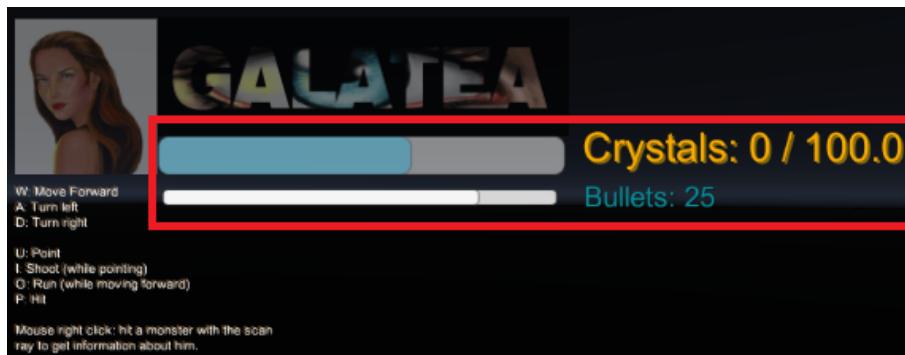


Fig. 3.71 – El recuadro rojo muestra los valores actuales de algunas variables

Cuando el enemigo ataca al personaje, además de acceder a la variable pública que controla sus puntos vitales, también accede a la variable dentro del *script* del objeto *PersonajeUI* que tiene el *slider* que representa los mismos (Fig. 3.72).

```

timer = 0f;
enemigoAnimacionesScript.EnemigoAnimacionesAtacar(true);
personajeVidaScript.personajeVidaActual = personajeVidaScript.personajeVidaActual - enemigoDaño - Random.Range(1f, 10f);

personajeUIScript.personajeSlider.value = personajeVidaScript.personajeVidaActual;
personajeUIScript.PintarPantallaRoja();

if (personajeVidaScript.personajeVidaActual < 100 && personajeVidaScript.personajeVidaActual > 10) personajeVidaScript.pai

```

Fig. 3.72 – Fragmento del código EnemigoAtacar de un enemigo

Así, a la hora de gastar balas, recoger cristales y correr, los valores de las variables se verán reflejados en la *UI*. Esto por supuesto es necesario para el jugador.

La *UI* también se hace presente en otras situaciones también, como las siguientes:

- El personaje tiene menos de 20 puntos de vida: Cada cierto tiempo, un parpadeo de una imagen color rojo y un aviso indicando peligro aparecen (Fig. 3.73).
- Se presiona la tecla “*Escape*”: El juego se pausa abriendo una ventana consultando si se desea salir (Fig. 3.74).



Fig. 3.73 – El personaje tiene menos de 20 puntos de vida

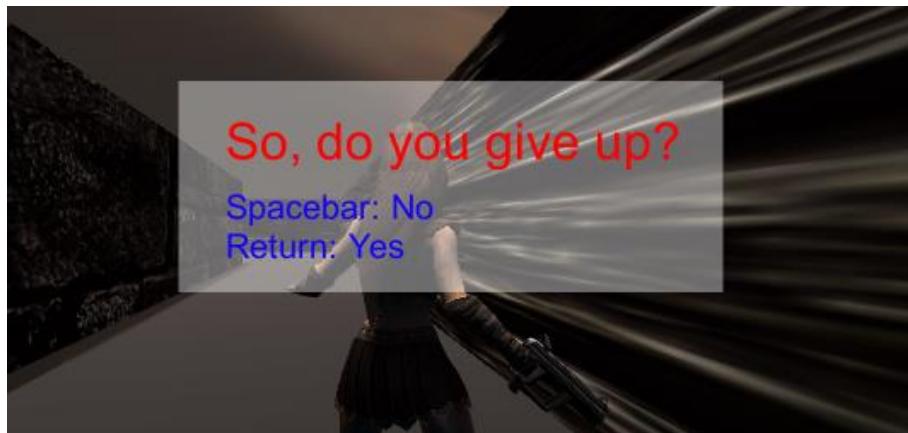


Fig. 3.74 – La tecla “*Escape*” fue presionada, aparece una ventana interactiva

3.3.7 Items

En el juego también hay ciertos items que cumplen determinadas funciones. Esto es, objetos dentro de una escena ubicados en algún punto (x,y,z) del universo que, al colisionarlos, pueden modificar el valor de una variable, mostrar un mensaje en la *UI* o salvar la partida accediendo a un archivo binario de texto. La particularidad que tienen es que luego de ser colisionados

y responder a la condición sujeta en el *script* adjunto, se destruyen (pues, ya no se necesitan en escena).

Cada uno de estos objetos tiene un modelo 3D coherente de acuerdo a la acción que tiene lugar. Por ejemplo, un objeto cuyo modelo 3D es una pistola, aumentará el valor de la variable balas (Fig. 3.75).

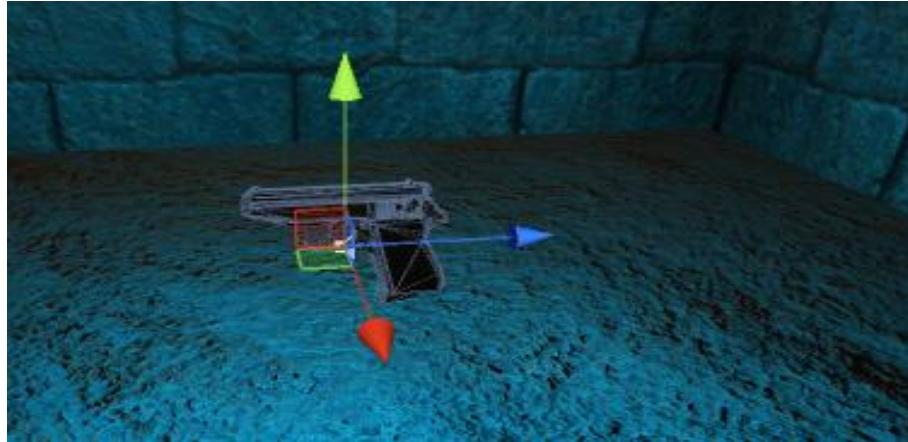


Fig. 3.75 – Modelo 3D de una pistola

Nuevamente, el concepto de *triggers* o disparadores se hace presente en la función *OnTriggerEnter* (Fig. 3.76), como así también la condición que la etiqueta del objeto que invadió su espacio sea “Personaje”, pues, un *zombie* no utiliza balas, y la referencia a la variable no podrá ser accedida.

```
void OnTriggerEnter(Collider personajeCollider)
{
    //Debug.Log("Dentro del collider");
    //Debug.Log(personajeCollider);
    if (personajeCollider.CompareTag("Personaje"))
    {
        PersonajeBalas.cantidadBalas = PersonajeBalas.cantidadBalas + 5;
        sonidoCristal.Play();
        luz.enabled = false;
        Destroy(gameObject, 0.7f);
    }//if

}//OnTriggerEnter
```

Fig. 3.76 – Fragmento del código adjunto al objeto

Se puede apreciar también en la Fig. 3.76 que luego de un cierto tiempo el objeto se autodestruirá. La razón por la cuál se puso un retardo para esta acción de colisionar con un objeto item es para que se reproduzca un *clip* de audio adjunto al mismo indicando que se ha colisionado.

De igual manera, un objeto sumará a la variable que representa los puntos vitales del personaje un valor al ser colisionado. El algoritmo es el mismo salvo la variable involucrada. Para este caso, dentro de los modelos 3D

gratuitos disponibles en el *Unity Asset Store*, una hamburguesa lo representa (Fig. 3.77).



Fig. 3.77 – Modelo 3D de una hamburguesa

Ahora bien, hay items los cuáles solo tienen la función de mostrar algún mensaje en pantalla por medio de la *UI* una vez colisionados. La finalidad de estos es mantener al jugador inmerso en la historia, aunque esta sea algo confusa y extraña. Previamente se mencionó que voces extrañas aparecían todo el tiempo, algunas de ellas pertenecientes a la gente del hospital, otras a memorias guardadas en la mente del personaje (Fig. 3.78) y otra extraña con tono aterrador.

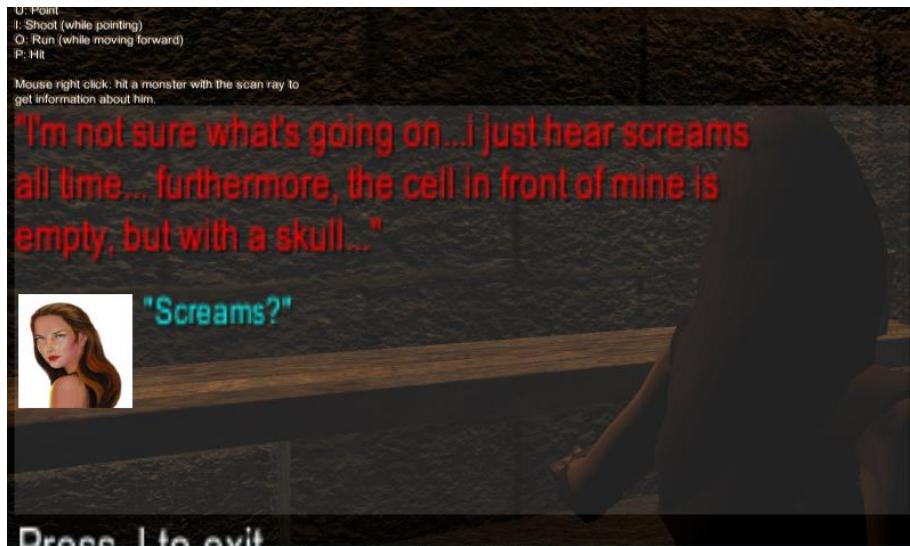


Fig. 3.78 – Un texto aparece en pantalla luego de colisionar un objeto

El mensaje que muestra la Fig. 3.78 es propio de un modelo 3D que representa un libro (Fig. 3.79). Este es usado para todos estos tipos de mensajes ocultos y raros (voces de la gente en el hospital y recuerdos de su memoria).



Fig. 3.79 – Modelo 3D de un libro

Mientras que, para las voces aterradoras, el modelo elegido es un bastón de Nigromante (Fig. 3.80). Lo único que difiere del libro es el sonido al momento de ser colisionado y que los mensajes son provocativos y aterradores.

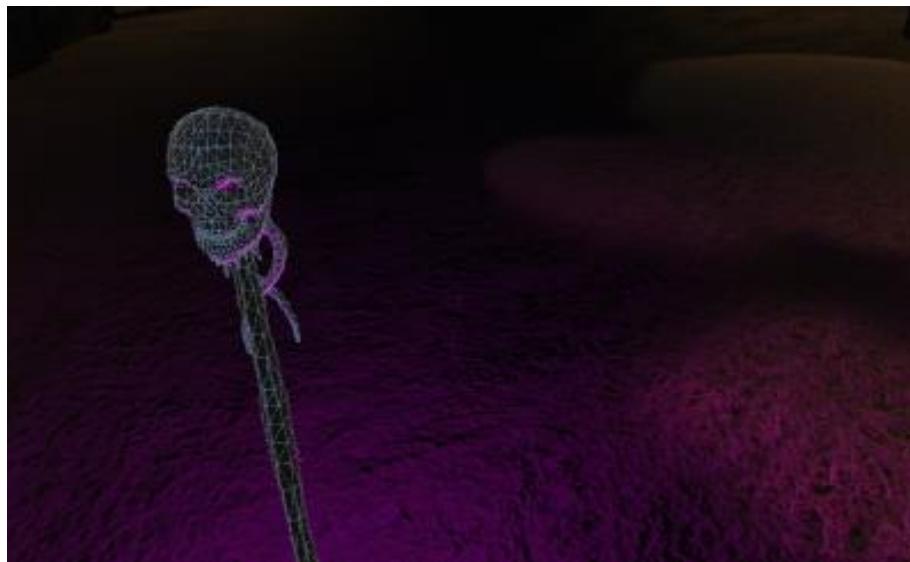


Fig. 3.80 – Modelo 3D, bastón de nigromante

Al comienzo de cada nivel, se agregó un ítem el cuál, al colisionar con el mismo, automáticamente salvará la partida en ese nivel. Es decir, la próxima vez que se ejecute la aplicación, al apretar la tecla ‘F1’, el juego arrancará desde el último nivel el cuál se colisionó con este objeto.

La idea principal del modelo 3D que representaría el objeto era un pergamino o un *disquete*, pero no había algo similar disponible gratuitamente, por lo que el modelo 3D elegido es el que muestra la Fig. 3.81:



Fig. 3.81 – Objeto el cuál, al colisionar con el mismo, salva la partida en el nivel actual

Luego de colisionar con el objeto, un mensaje en pantalla aparecerá indicando implícitamente que se ha salvado la partida (Fig. 3.82).



Fig. 3.82 – Mensaje al colisionar con el objeto

Salvar la partida es guardar y cargar datos del juego usando serialización, utilizando variables estáticas para conservar los datos cuando se cambia de escena. Algo no menos importante es aclarar que este método para guardar y cargar datos del juego funciona en todas las plataformas excepto en el reproductor Web.

El objeto tiene adjunto un script que permite serializar los datos del juego, es decir, convertirlo a un formato que puede ser salvado y cargado más adelante. A continuación un fragmento del código en la Fig. 3.83.

```

using UnityEngine;
using System.Collections;

using UnityEngine.SceneManagement;

using System;
using System.Runtime.Serialization.Formatters.Binary;
using System.IO;

public class Save_Load_Scene : MonoBehaviour
{
    public string nombreEsteNivel_MonoBehavior;

    void OnTriggerEnter(Collider colliderImpactado)
    {
        if (colliderImpactado.CompareTag("Personaje"))
        {
            SaveLevel();
        }
    }
}

```

Fig. 3.83 – Fragmento del código

De la Fig. 3.83:

- *System. Runtime. Serialization.Formatters.Binary* permite usar las capacidades de serialización del sistema operativo dentro del *script*.
- *System.IO* permite escribir y leer desde nuestra computadora o dispositivo móvil. En otras palabras, esta línea permite crear archivos y poder leerlos más tarde.

El objeto, como el resto de los items, está ubicado en algún punto (x,y,z) del universo de una escena. Entonces como primer paso, se debe identificar la escena en la que se encuentra actualmente. Para ello, una variable pública tipo cadena es necesaria puesto que cada escena puede ser identificada por un número entero o una cadena de caracteres. Entonces, una vez ubicado el objeto en escena, la cadena de caracteres de la variable perteneciente al *script* del objeto tiene que coincidir con el nombre de la escena actual. La siguiente imagen (Fig. 3.84) ilustra lo mencionado recién:

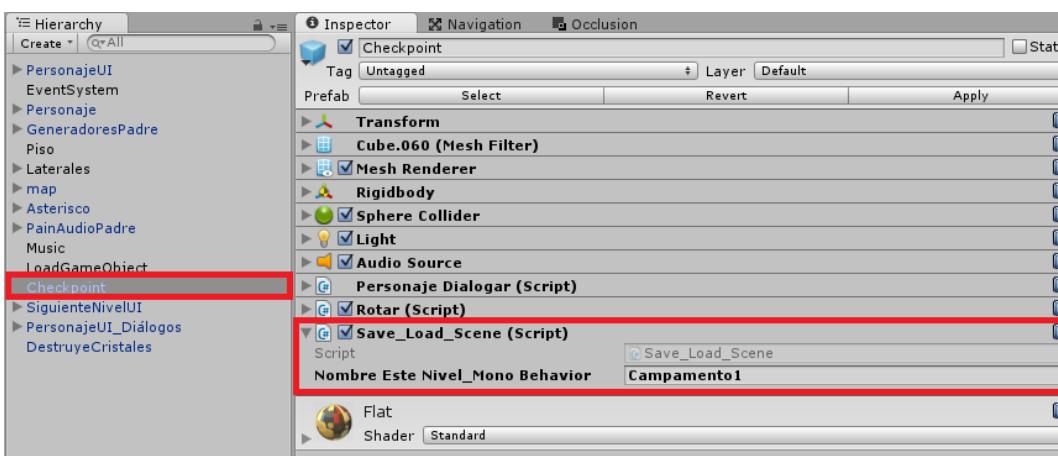


Fig. 3.84 – El objeto Checkpoint

En la Fig. 3.84, el objeto está ubicado en un punto (x,y,z) del espacio de la escena llamada “Campamento1”. El recuadro rojo de la derecha muestra uno de sus componentes adjuntos, este es el script Save_Load_Scene, el cuál tiene la variable pública tipo cadena NombreEsteNivel_MonoBehaviour cuyo conjunto de caracteres es “Campamento1”, es decir, coincide con el nombre de la escena actual en la que está instanciado.

Una vez cumplida la condición que muestra el recuadro rojo de la Fig. 3.85, la función SaveLevel es llamada para guardar los datos del juego.

```
public class Save_Load_Scene : MonoBehaviour
{
    public string nombreEsteNivel_MonoBehavior;

    void OnTriggerEnter(Collider colliderImpactado)
    {
        if (colliderImpactado.CompareTag("Personaje"))
        {
            SaveLevel();
        }
    }
}
```

Fig. 3.85 – Fragmento de código del Script adjunto

Dentro del *script*, una clase serializable es creada. La clase contendrá una variable tipo cadena la cuál guardará el nombre de la escena en un archivo binario para luego poder acceder a ella (Fig. 3.86).

```
[Serializable]
class DatosJugador
{
    public string nombreEsteNivel_SerializableClass;
}
```

Fig. 3.86 – Clase serializable

Se puede ver a continuación en la Fig. 3.87 la función SaveLevel(): primero se crea un archivo binario, se copia el conjunto de caracteres de la variable que contiene el nombre de la escena actual en la variable cadena de la clase serializable (recuadro rojo) y luego se cierra el archivo.

```

public void SaveLevel()
{
    BinaryFormatter formatoBinario = new BinaryFormatter();
    FileStream archivoGuardar = File.Create(Application.persistentDataPath + "/DatosJugador.dat");

    DatosJugador datosJugadorVariable = new DatosJugador();
    datosJugadorVariable.nombreEsteNivel_SerializableClass = nombreEsteNivel_MonoBehavior;

    formatoBinario.Serialize(archivoGuardar, datosJugadorVariable);
    archivoGuardar.Close();

} // SaveLevel

```

Fig. 3.87 – Función SaveLevel

Para cargar el nivel, un mensaje informativo se muestra en la primera escena del juego en modo aventura como se puede ver en la siguiente imagen (Fig. 3.88):

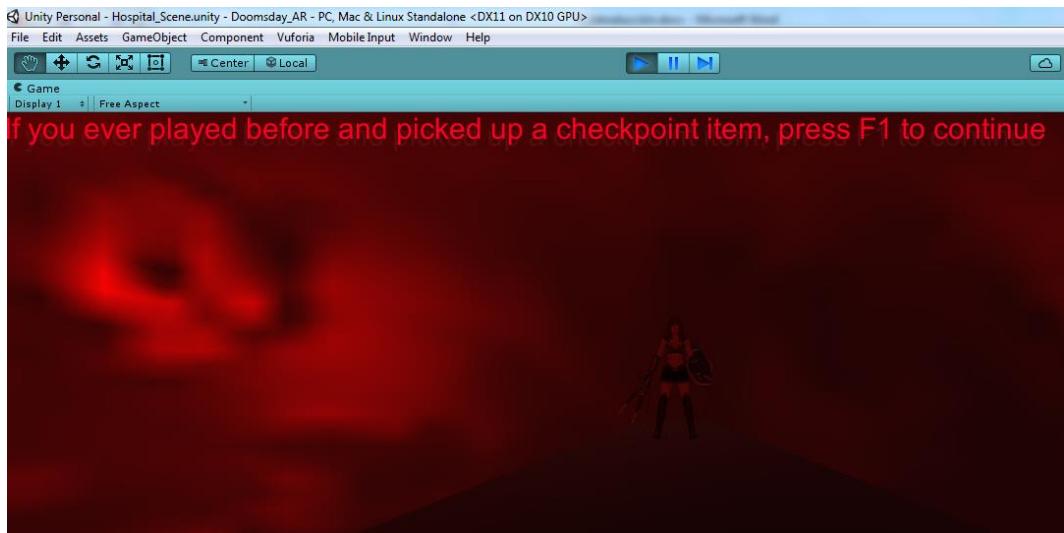


Fig. 3.88 – Mensaje informativo, presionar tecla F1 para cargar el último nivel jugado

La condición de la tecla ‘F1’ pulsada o no es controlada cuadro a cuadro dentro de la función *Update* de Unity (Fig. 3.89).

```

// Update is called every frame, if the MonoBehaviour is enabled
public void Update()
{
    if (Input.GetKeyDown(KeyCode.F1)) LoadLevel();
} // Update

```

Fig. 3.89 – Función Update del script, donde se realiza el control de la tecla F1 presionada

Una vez pulsada, la función *LoadLevel* será llamada (Fig. 3.90).

```

public void LoadLevel()
{
    if (File.Exists(Application.persistentDataPath + "/DatosJugador.dat"))
    {
        BinaryFormatter formatoBinario = new BinaryFormatter();
        FileStream archivoCargar = File.Open(Application.persistentDataPath + "/DatosJugador.dat", FileMode.Open);

        DatosJugador datosJugadorVariable = (DatosJugador)formatoBinario.Deserialize(archivoCargar);

        archivoCargar.Close();

        SceneManager.LoadScene(datosJugadorVariable.nombreEsteNivel_SerializableClass);
    }
}

```

Fig. 3.90 – Función LoadLevel

Fig. 3.90: La primera condición a cumplirse es verificar la existencia del archivo. En caso de éxito, este se abre y se procede a utilizar la función antes vista `SceneManager.LoadScene` para cargar la escena cuyo nombre coincide con el conjunto de caracteres de la variable perteneciente a la clase serializable (recuadro rojo).

3.3.8 Escenarios

Cada escena 3D representa un nivel jugable. En estos niveles, se necesita ubicar en el espacio de coordenadas (x,y,z) objetos que cumplan el rol de escenario tanto para el personaje como para los enemigos. Así, pisos y paredes son requeridos para el propósito.

Para armar los escenarios, combinaciones varios objetos primitivos de Unity como los cubos fueron posicionados y rotados en escena para formarlos (Fig. 3.91). También se utilizaron modelos 3D descargados (Fig. 3.92) y otras combinaciones.

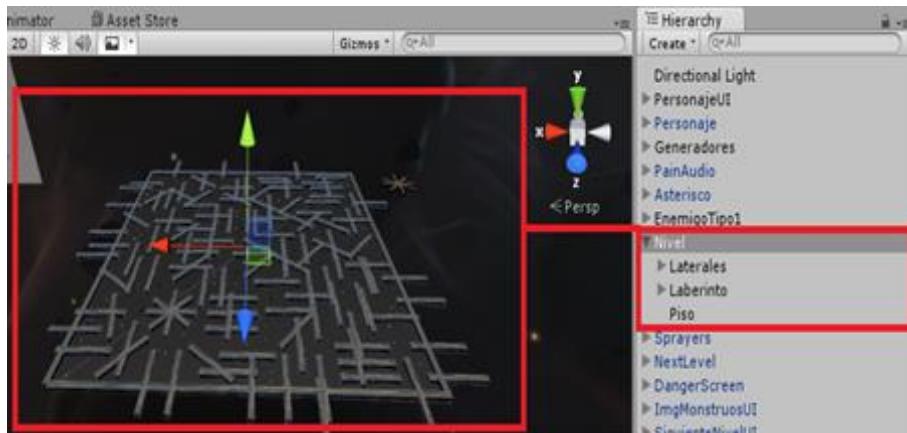


Fig. 3.91 – Escenario de la escena llamda Laberinto

La Fig. 3.91 muestra un escenario formado solo por objetos 3D primitivos de Unity. Tanto el piso como las paredes son cubos esculpidos de tal manera que conforman el diseño del mismo.

También hay escenarios cuyos modelos 3D fueron descargados del Asset Store de Unity, como muestra la Fig. 3.92 a continuación.



Fig. 3.92 – Escenario de la escena llamada Tumba

La idea original siempre fue descargar todos los niveles para darle mejor calidad al juego, pero hubo que descartar muchos de estos debido al enorme peso de los assets. Además, aparecieron incompatibilidades molestas entre los modelos 3D descargados y el motor debido a que este se actualiza a menudo. Algunos modelos 3D gratis son muy antiguos y pierden parte de su funcionalidad en las nuevas versiones.

3.4 Fase 4: Beta

Repasando el concepto de las tres 5s:

- 5 segundos vista: a los 5 segundos de juego, el jugador debe sentir que entiende los controles y que le divierte utilizarlos.
- 5 minutos vista: a los 5 minutos de juego, el jugador debe haber experimentado una pequeña misión del juego y tiene que haber encontrado interesante el universo.
- 5 horas vista: a las 5 horas de juego, el jugador debe haber entendido el conflicto narrativo y le divierte solucionarlo.

Para este tópico, es necesario siempre la colaboración de los *Testers* (terceros que prueban el juego). Gracias a ellos se corrigieron algunos errores dentro del juego, puesto que a medida que se iba avanzando en el proyecto, se mostraba al público versiones antiguas del mismo, de allí surgían opiniones y detecciones de errores las cuales se las anotaban, analizaban y tomaban decisiones.

Vale aclarar aquí se tiene en cuenta los 5 segundos vista y los 5 minutos vista, puesto que el juego no es tan largo para cubrir las 5 horas vista. A continuación, los puntos de acuerdo a las opiniones más comunes y la decisión que se tomó respectivamente:

Arte del juego: Hoy en día, los juegos tienen un alto grado de calidad gráfica, es algo común al probar un juego nuevo que las imágenes 2D, modelos 3D, efectos de partículas, etc. tengan muy buenos diseños, puesto que las

empresas tienen gente dedicada a eso (los artistas). La opinión constructiva común fue la de mejorar por ejemplo la imagen de fondo del menú principal, proponiendo algo que llame más la atención. Pues bien, no se tomó ninguna acción al respecto en este tema, puesto que este proyecto tiene fines académicos de ingeniería y no comerciales, sumado al hecho de que esa tarea requiere de alguien dedicado al arte 2D, lo cuál no es este caso. Sin embargo, si se pudo tomar en cuenta el momento del sonido, ya que un tercero se encargó de llevar a cabo esta tarea. Entonces, la sugerencia de una buena música de ambientación pudo ser atendida.

Jugabilidad: Al principio, el personaje principal se desplazaba a una cierta velocidad uniforme dada por una variable en un *script*. Esto, no le gustó a prácticamente el 90% de la gente que probó versiones anteriores, en efecto, fue la opinión más común, por lo que inmediatamente se añadió la opción de correr. Otras opiniones (entre varias) como agregar animaciones al monstruo a la hora de ser impactado por una bala y algún indicador alevoso cuando se es atacado fueron tratadas mediante efectos de sistemas de partículas y 2D.

Errores: Paredes falsas, se podían atravesar. Cabe aclarar que se debió mucho al modelo 3D descargado, por lo que hubo que trabajarlos un poco y modificar algunas propiedades que estos traían por defecto. Agregar componentes físicos a la entidad fue la solución.

Finalmente, podría considerarse el comentario más importante (puede incluirse en jugabilidad también) el hecho de querer comenzar a jugar en el modo historia desde el nivel en el que abandonó el juego previamente y no desde el principio siempre. Entonces se decidió atender esta petición creando un sistema de creación y gestión de partidas, es decir, guardar y cargar datos del juego usando serialización, utilizar variables estáticas para conservar los datos cuando se cambie de escena o nivel.

3.5 Fase 5: Cierre

Luego de pasar la fase beta y corregir, modificar y agregar ciertos detalles, finalmente se decidió dar por finalizado el juego.

Sección 2

Aplicación para Android

Capítulo 4

Realidad Aumentada

4.1 Introducción

La realidad aumentada es una aplicación interactiva que combina la realidad (universo físico), con información virtual (imágenes 3D, sonidos, videos, texto, sensaciones táctiles) en tiempo real y de acuerdo al punto de vista del usuario, enriqueciendo la experiencia visual. El concepto de “realidad aumentada” puede ser aplicado a un gran número de usos. De forma general y bajo una definición expandida se puede entender que se refiere a cualquier aumento de las capacidades perceptivas y de acción del ser humano.

La diferencia principal entre Realidad Virtual (RV) y Realidad Aumentada (RA) es que, mientras la RV implica inmersión del participante en un mundo totalmente virtual, la RA implica mantenerse en el mundo real con agregados virtuales. La información virtual tiene que estar vinculada espacialmente al mundo real de manera coherente, lo que se denomina registro de imágenes. Por esto se necesita saber en todo momento la posición del usuario, con respecto al mundo real.

Las necesidades de un sistema de RV y RA pueden dividirse:

- Dispositivos de entrada:
 - Mecanismos de entrada de comandos y datos
 - Tracking
- Dispositivos de salida:
 - Salida visual
 - Salida auditiva
 - Salida táctil o *haptics*
- Aplicación en tiempo real:
 - Visualización del mundo virtual (en RV), o mixto (en RA), de acuerdo a la posición y orientación del participante.
 - Interactividad.

Para este proyecto:

- Dispositivo de entrada: Tracking.
- Dispositivo de salida: Visual (teléfono móvil).
- Visualización del mundo mixto de acuerdo a la posición y orientación del participante.

La Fig. 4.1 ilustra un ejemplo de realidad aumentada donde el tipo de salida es visual. La cámara de un dispositivo móvil captura el video de una escena real y, al momento de enfocar una determinada imagen 2D en el mundo, objetos virtuales son visualizados en la pantalla del dispositivo móvil (Mario Bros y otros detalles).



Fig. 4.1 - Aplicando RA mediante el uso de un dispositivo móvil de visualización

Las diferentes partes involucradas en una aplicación de realidad aumentada, donde la salida es visual (ejemplo Fig. 4.1) son:

- Captura de la escena real: la escena real se captura mediante una interfaz de visión. El video capturado puede utilizarse para *tracking* basado en visión, es decir, basado en el análisis de la imagen mediante algoritmos de visión.
- *Tracking* del participante: El *tracking* es el proceso de seguimiento de un objeto en movimiento, es decir, la estimación de la posición y orientación del mismo en cada instante. En aplicaciones de RA, se debe realizar el seguimiento o *tracking* del participante para determinar su posición y orientación en el mundo real. Generalmente, en aplicaciones de realidad virtual y algunas aplicaciones de realidad aumentada, se realiza el *tracking* de la cabeza para realizar la visualización del mundo virtual o mixto de acuerdo al punto de vista del participante. En otras, puede necesitarse el *tracking* de objetos que sostiene el participante. Dependiendo del tipo de aplicación será la precisión que se necesite para determinar la posición del usuario y en consecuencia el tipo de *tracking* que se realice. Para realizar el *tracking* basado en visión, es necesaria la captura de la escena real.
- Generador de la escena virtual: se tiene un mundo virtual, y con la información de la posición y orientación del participante se genera una vista acorde del mismo.
- Combinación del mundo virtual y la escena real: en los casos en los que existe una visión directa del mundo real, la combinación se realiza directamente en el ojo del participante. En otro caso, se realiza un video, resultado de la escena capturada y la escena sintética proyectada.

La Fig.4.2 muestra el diagrama de las partes involucradas mencionadas anteriormente:

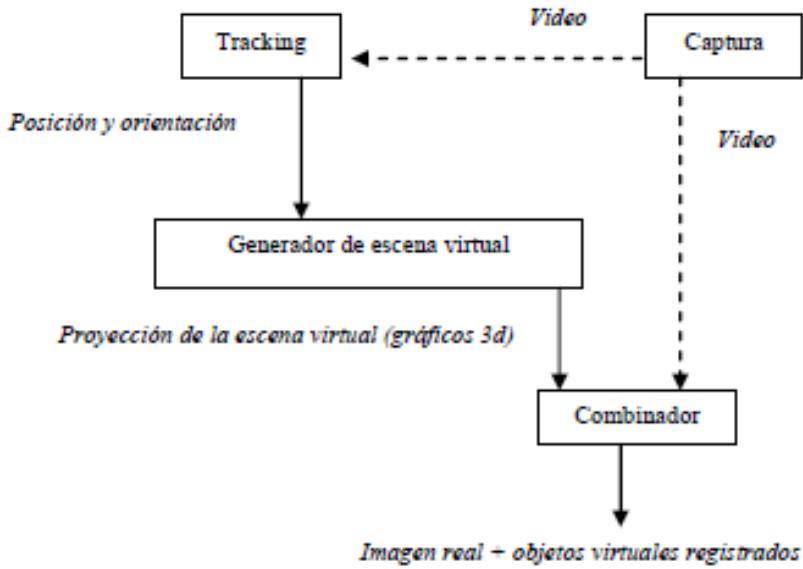


Fig. 4.2 - Diagrama de una aplicación de RA

Particularizando la Fig. 4.2 para un dispositivo móvil (dispositivo de visualización utilizado en el proyecto):

- Captura: Cámara captura el video real.
- *Tracking*: De la cámara. Basado en visión o mediante sensores (GPS y brújula digital; sensores iniciales como acelerómetros y giroscopios).
- Generador de escena virtual: Software para móviles.
- Combinador: Generación y visualización de video combinando la captura de la proyección de la escena sintética.

Hacer una revisión de los juegos que incorporan realidad aumentada es una manera muy buena de describir cómo ha ido evolucionando tanto la tecnología como la aproximación del concepto en sí de realidad aumentada en los últimos años. En el año 2000 algunas universidades comenzaron a ver el potencial que podía tener el uso de la realidad aumentada y para su investigación comenzaron a crear réplicas de juegos para el ordenador o las videoconsolas usando esta tecnología. Un juego clásico muy conocido y replicado de este modo es PacMan, que fue implementado por la National University of Singapore, de manera que el jugador podía ser, bien un fantasma o el propio Pac-Man y el laberinto eran las propias calles de Singapur. Para poder jugar, el usuario tenía que disponer de un ordenador portátil, unas gafas (que permitían ver la realidad y los datos del juego), GPS, Bluetooth, Wifi, infrarrojos y sensores.

4.2 Idea de la Aplicación

La aplicación para dispositivo móvil es independiente al ejecutable para PC. Sin embargo, mientras se juega (con el teclado y el mouse) en la computadora, en algunas ocasiones el dispositivo móvil forma parte de la jugabilidad también. Sus funciones son 2:

- Identificar el monstruo en frente y brindar información del mismo.
- Mostrar el escenario de la escena la cuál se está jugando actualmente (aplicado en 2 niveles).

Las imágenes a continuación describen las funcionalidades mencionadas (Fig. 4.3 y Fig. 4.4).



Fig. 4.3 – Captura de pantalla de la serie animada Pokémon.



Fig. 4.4 – Mini mapa de un nivel en el juego Resident Evil 3

Fig. 4.3: En un dibujo animado japonés llamado Pokémon, el personaje principal de la historia utilizaba un aparato (un dispositivo móvil llamado Pokédex), el cuál, al enfocar su lente en un animal o criatura perteneciente a la serie (llamados *pokémon*), obtenía del animal un breve resumen de sus características principales (por ejemplo, las zonas donde habitaba).

Fig. 4.4: En varios tipos de juegos, es muy común ver en algún margen de la pantalla, o por ejemplo al pausar el juego, un mini mapa del nivel cuál se está jugando, sobre todo en juegos largos de aventura, acción o survival horror que utilizan cámaras en primera y tercera persona.

Capítulo 5

Especificación de Requisitos de Software (ANSI/IEEE 830-1998)

5.1 Análisis de requisitos del sistema

Esta especificación tiene como objetivo analizar y documentar las necesidades funcionales que deberán ser soportadas por el sistema a desarrollar. Para ello, se identificarán los requisitos que ha de satisfacer el nuevo sistema, mediante técnicas de educación de requisitos, el estudio de los problemas de las unidades afectadas y sus necesidades actuales. Además de identificar los requisitos se deberán establecer prioridades, lo cual proporciona un punto de referencia para validar el sistema final que compruebe que se ajusta a las necesidades del usuario.

5.2 Identificación de los usuarios participantes

En el ámbito del desarrollo del software se identifican el siguiente grupo de usuarios:

- Grupo de Usuarios Finales: formado por el conjunto de usuarios capaces de acceder a la información brindada por el sistema. En este caso se trata de toda persona que esté jugando al juego para PC “Force of Will” y haya descargado la aplicación en su dispositivo móvil previamente.

5.3 Catálogo de requisitos del sistema

El objetivo de la especificación es definir en forma clara, precisa, completa y verificable todas las funcionalidades y restricciones del sistema que se desea construir. Será el canal de comunicación entre las partes implicadas. Esta documentación estará sujeta a revisiones, hasta alcanzar su aprobación. Una vez aprobado servirá de base al equipo para el desarrollo del nuevo sistema. Esta especificación se ha realizado de acuerdo al estándar “IEEE Recommended Practice for Software Requirements Specifications (IEEE/ANSI 830-1998)”.

Este sistema llevará el nombre de FOWAPP.

5.4 Objetivos y alcance del sistema

El principal objetivo del sistema a desarrollar consiste en detectar, a partir de una fuente de video en tiempo real, distintas imágenes planas mostradas en el juego para PC. Luego, mediante algoritmos de procesamiento de imagen, el sistema deberá ser capaz de mostrar los elementos virtuales también en

tiempo real de acuerdo a cada imagen detectada en la fuente. El sistema a desarrollar debe ser abierto y escalable, permitiendo incorporar nuevas funcionalidades en el futuro.

5.5 Definiciones, acrónimos y abreviaturas

Definiciones:

- *Target*: en este trabajo se denomina *target* a toda aquella imagen 2D plana.
- *Image Targets*: son las imágenes objetivos a ser detectadas.
- *Framework*: es un conjunto estandarizado de conceptos, prácticas y criterios para enfocar un tipo de problemática particular que sirve como referencia, para enfrentar y resolver nuevos problemas de índole similar.
- *Middleware*: o lógica de intercambio de información entre aplicaciones (*interlogical*) es un software que asiste a una aplicación para interactuar o comunicarse con otras aplicaciones, o paquetes de programas, redes, hardware y/o sistemas operativos.

Acrónimos:

- API: *Application Programming Interface* (Interfaz de programación de aplicaciones).
- SDK: *Software Development Kit* (Kit de Desarrollo de Software).
- COCOMO: *COnstructive COst MOdel* (Modelo Constructivo de Costos).
- RAD: *Rapid Application Development* (Desarrollo Rápido de Aplicaciones).
- IDE: *Integrated Development Enviroment* (Entorno de desarrollo integrado).
- GPU: *Graphics Processor Unit* (Unidad de procesamiento gráfico).
- S.O: Sistema Operativo.

Abreviaturas:

- App: *Application* (aplicación).

5.6 Descripción General

Esta sección presenta una descripción general del sistema con el fin de conocer las funciones que debe soportar, los datos asociados, las restricciones impuestas y cualquier otro factor que pueda influir en la construcción del mismo. La función que debe realizar el sistema es la siguiente:

- Reconocimiento de imágenes planas: el sistema debe permitir visualizar, en tiempo real, el video que la cámara del dispositivo está tomando y mostrar en pantalla el resultado de la detección de imagen, que será el contenido virtual programado para cada caso.

5.7 Requisitos funcionales

Detección de imágenes y visualización de elementos virtuales:

- Introducción: el sistema debe permitir al usuario visualizar por la pantalla del móvil, en tiempo real, la imagen mostrada en el juego de PC y la información virtual asociada a ella.
- Entrada: conjunto de imágenes 2D que son mostradas en la pantalla de la computadora en la cual se está ejecutando el juego.
- Proceso: una vez que el usuario haya iniciado la aplicación y se encuentre en la pantalla principal de la misma, deberá presionar el botón “Start”. Al presionar este botón, el sistema abre la cámara del dispositivo. La instancia de la cámara se encarga de que cada fotograma capturado por la cámara digital se pase al *tracker* y se envíe automáticamente al convertidor de imágenes. La instancia del conversor de formato de pixel realiza la conversión entre el formato con el que trabaja la cámara a un formato adecuado para el renderizado con el cual trabaja la librería gráfica (utilizada por el SDK de realidad aumentada), OpenGL ES 2.0 para manejar el contenido virtual. El *tracker* contiene los algoritmos de visión computacional para detectar y seguir (*detect & track*), las características naturales de las imágenes en los fotogramas capturados por la cámara. Los resultados se almacenan en un objeto de estado que es utilizado por el procesador de vídeo de fondo el cual hace la combinación del contenido real con el virtual.

Mientras el usuario mantenga el *target* en foco, el sistema realizará todo el proceso descripto, renderizará el resultado de la imagen “combinada” y verá la realidad (puesto que la cámara siempre permanece abierta hasta que no se sale de esta funcionalidad) junto a los objetos virtuales asociados a la escena que han sido previamente programados para cada *target*.

- Salida: se muestra en pantalla el contenido virtual asociado al *target* que está siendo capturado por la cámara y detectado por el sistema.

5.8 Suposiciones y Dependencias

- Suposiciones: se asume que los requisitos en este documento son estables una vez que sean aprobados por el administrador del Sistema.
- Dependencias: actualmente no existen dependencias.

5.9 Requisitos de usuario y tecnológicos

- Requisitos de usuario: los usuarios serán todos aquellos que posean acceso al sistema, es decir, todos aquellos jugadores del juego “Force of Will” para PC que además hayan descargado e instalado la aplicación en su móvil. Las interfaces serán intuitivas, fáciles de usar y amigables, de manera tal que no sea requisito necesario realizar una capacitación para hacer un correcto uso de las mismas.
- Requisitos tecnológicos: los usuarios deberán contar con un dispositivo móvil (*tablet*, *smartphone*, etc.) que funcionen con sistema operativo Android versión 2.3.3 Nivel de API 9 y posean cámara de fotos integrada de 2 Megapíxeles.

5.10 Requisitos de interfaces externas

- Interfaz de *hardware*: dispositivo móvil con cámara de fotos integrada de 2 Megapíxeles.
- Interfaz de *software*: sistema operativo Android versión 2.3.3 Nivel de API 9 (Gingerbread).
- Interfaces de usuarios: las interfaces de usuario deben ser intuitivas, fáciles de usar y amigables.

Las ventanas deberán seguir una progresión lineal, es decir, si el usuario realiza una acción y quiere volver atrás, que sea posible.

Las interfaces de la aplicación deben ser lo más claras posibles, minimalistas, con el objeto de que al usuario no le resulten complicadas o confusas debido a, por ejemplo, demasiados botones en una sola ventana, o a la cantidad de elementos virtuales y contenidos 3D presentes al mismo tiempo en pantalla.

5.11 Requisitos de rendimiento

Un requisito indispensable para la aplicación se trata de dar una respuesta en tiempo real a todos los eventos que ocurran, tanto el movimiento de la cámara, como sobre la representación de la escena al ser reconocida.

El tiempo de respuesta de la aplicación en el cual detecta una imagen para mostrar el contenido virtual asociado a ella no debe ser superior a 1 segundo. Estos resultados se obtienen de las pruebas que se hicieron con *Smartphone*.

Al tratarse de una aplicación de Realidad Aumentada, tiene una fuerte dependencia sobre qué es lo que se capta en cada instante y por lo tanto se pretende que la recreación de los objetos añadidos al mundo físico sea en tiempo real y sin cortes o retardos. Así pues, se ha trabajado en busca de la mayor eficiencia de las funcionalidades que se han proporcionado a la aplicación.

En términos generales, se ha buscado el desarrollo de una aplicación de Realidad Aumentada cuyo funcionamiento esté dotado de la mayor sencillez e intuitividad posible y huyendo de las pantallas sobrecargadas.

La aplicación sólo se instanciará una vez por terminal, limitando el número de usuarios posibles por teléfono a uno. La cantidad de información que manejará debe ser mínima, dadas las limitadas capacidades de los teléfonos. Debe ser capaz de ejecutarse en un terminal con las siguientes características:

- Android 2.3.3 Nivel de API 9.
- 128 MB de RAM.
- Procesador de 200MHz. ARM9 o mejor.

Estos son los requisitos mínimos en cuanto a rendimiento del terminal para que corra Android correctamente y con los cuales se puede trabajar en el Framework y el IDE usados. Se ha probado la aplicación en dos terminales Android con características distintas y superiores a las mínimas nombradas.

5.12 Requisitos de desarrollo

El ciclo de vida será el de Ciclo de vida en Espiral o Modelo Iterativo basado en Prototipos, debiendo orientarse hacia el desarrollo de un sistema flexible que permita incorporar de manera sencilla cambios y nuevas funcionalidades.

5.13 Restricciones de diseño

- Ajuste de estándares: No se han definido.
- Seguridad: No se han definido.
- Política de respaldo: Está definida por el desarrollador del sistema.
- Política de borrado: Está definida por el desarrollador del sistema.

Capítulo 6

Selección del Modelo de Ciclo de Vida y Gestión del Proyecto

6.1 Introducción

El ciclo de vida es la transformación que el producto *software* sufre a lo largo de su vida, desde que nace hasta que muere. El resultado de las transformaciones que sufre el producto *software* a lo largo de su ciclo de vida representa en esencia el producto mismo y se denomina estado.

Un ciclo de vida determina el orden de las fases de un proceso *software* y establece los criterios de transición de una fase a la otra. El proceso *software* es una colección de actividades que comienzan con la identificación de una necesidad y concluye con el retiro del *software* que satisface dicha necesidad.

Al comienzo de un proyecto *software* se debe elegir el ciclo de vida que seguirá el producto a construir. El modelo de ciclo de vida elegido llevará a encadenar las tareas y actividades del proceso *software* de una determinada manera. Se debe tener en cuenta que algunas tareas serán realizadas una vez y otras deberán realizarse más de una vez. El ciclo de vida apropiado se elige en base a la cultura de la corporación, el dominio del problema, la comprensión de los requisitos y la volatilidad de los mismos.

Un proyecto sin estructura es un proyecto inmanejable, no puede ser planificado ni estimado ni mucho menos alcanzar un compromiso de costos y tiempo. La idea de buscar ciclos de vida que describan las actividades a realizar para transformar el producto surgen de tener un esquema que sirvan como base para:

- Planificar
- Organizar
- Asignar personal
- Coordinar
- Presupuestar
- Dirigir

6.2 Selección de Metodología de Desarrollo de Software

En esta sección se definirá la metodología a utilizar para el desarrollo de este *software* para móviles. La elección de metodología es determinante durante

el desarrollo puesto que, dependiendo de ella, habrá más cargas de trabajo durante ciertas fases y menos en otras.

Hay también condicionantes a la hora de elegirla, por ejemplo la experiencia del desarrollador o la disponibilidad para realizar reuniones con el cliente o el supervisor del proyecto. A continuación se hace un análisis detallado de todos estos factores para llegar a la mejor elección.

Para definir la metodología usada para desarrollar este *software* se necesita una perspectiva de alto nivel. El objetivo es tanto desarrollar una aplicación informática en un tiempo limitado como desarrollarlo de la manera más práctica posible.

Debido a lo expuesto, la metodología elegida debe dar más peso a la obtención de prototipos en fases prematuras de desarrollo.

La metodología *Rapid Application Development* o RAD, es la que mejor se ajusta a las necesidades. RAD tiene una característica que es la descentralización de la responsabilidad, por lo que se adapta bien al ser este un proyecto de una sola persona; esta metodología es idónea ya que enfatiza la fase de programación, minimizando el tiempo de planificación, permitiendo al desarrollador centrarse en la programación.

A continuación se expone un diagrama que muestra el ciclo de desarrollo de la metodología RAD.

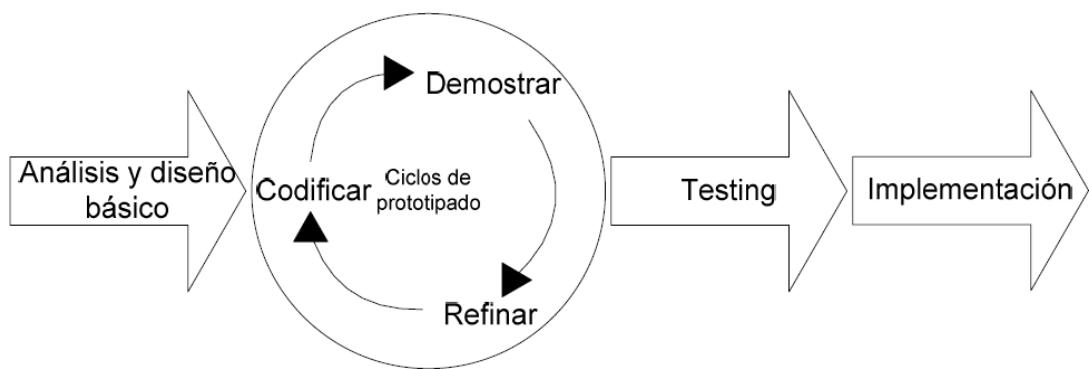


Fig. 6.1 – Diagrama metodología RAD

Fases de RAD:

- Modelado de gestión: el flujo de información entre las funciones de gestión se modela de forma que responda a las siguientes preguntas: ¿Qué información conduce el proceso de gestión? ¿Qué información se genera? ¿Quién la genera? ¿A dónde va la información? ¿Quién la procesó?
- Modelado de datos: el flujo de información definido como parte de la fase de modelado de gestión se refina como un conjunto de objetos de datos necesarios para apoyar la empresa. Se definen las características (llamadas atributos) de cada uno de los objetos y las relaciones entre estos objetos.

- Modelado de proceso: los objetos de datos definidos en la fase de modelado de datos quedan transformados para lograr el flujo de información necesario para implementar una función de gestión. Las descripciones del proceso se crean para añadir, modificar, suprimir, o recuperar un objeto de datos. Es la comunicación entre los objetos.
- Generación de aplicaciones: RAD asume la utilización de técnicas de cuarta generación. En lugar de crear software con lenguajes de programación de tercera generación, el proceso DRA trabaja para volver a utilizar componentes de programas ya existentes (cuando es posible) o a crear componentes reutilizables (cuando sea necesario). En todos los casos se utilizan herramientas automáticas para facilitar la construcción del software.
- Pruebas de entrega: Como el proceso RAD enfatiza la reutilización, ya se han comprobado muchos de los componentes de los programas. Esto reduce tiempo de pruebas. Sin embargo, se deben probar todos los componentes nuevos y se deben ejercitar todas las interfaces a fondo.

Ventajas:

- El uso de herramientas case ahorra tiempo.
- Los entregables pueden ser fácilmente trasladados a otra plataforma.
- El desarrollo se realiza a un nivel de abstracción mayor.
- Visibilidad temprana.
- Mayor flexibilidad.
- Menor codificación manual.
- Mayor involucramiento de los usuarios.
- Posiblemente menos fallas.
- Posiblemente menor costo.
- Ciclos de desarrollo más pequeños.
- Interfaz gráfica estándar.

6.3 Selección de un Modelo de Ciclo de Vida

Aunque el RAD ha sido el enfoque principal que se le ha dado al proceso de desarrollo del software, también se ha utilizado otro paradigma auxiliar como método de resolución de problemas. Este paradigma ha sido el modelo iterativo basado en prototipos. Esta metodología es llamada también Ciclo de Vida en espiral (el modelo de espiral es el modelo teórico del prototipado evolutivo).

El Ciclo de Vida en espiral o modelo iterativo basado en prototipos va a constar de las siguientes etapas:

- Identificación de requisitos: en esta fase se estudiarán los requisitos que debe cumplir el producto software, basándose en las necesidades que se desean cubrir. Para ello también hay que tener en cuenta las limitaciones de la plataforma, puesto que condicionarán en gran medida las posibilidades del software. La identificación de requisitos ha quedado definida en el Capítulo 5 - Especificación de Requisitos de Software (ANSI/IEEE 830-1998).
- Programación: implementación de los prototipos necesarios para cumplir los requisitos. Muchos de estos prototipos serán desecharables, puesto que se desarrollarán con la idea de comprobar el funcionamiento del sistema.
- Pruebas: de manera que se garantice que el software cumple los requisitos marcados. Los prototipos que finalmente se vayan a implementarse en el software final deben realizar correctamente las funciones para las que fueron desarrollados. Estas pruebas serán de validación y verificación es decir que lo que haga es lo que se quería y que lo haga bien.

Usando este modelo se generan dos tipos de prototipos: los desecharables y los evolutivos. Se muestra en la Figura 6.2 el gráfico que ilustra el modelo utilizado para desarrollar prototipos.

El prototipo desecharable es uno escrito de manera muy rudimentaria y se utiliza para programar las partes más complejas del código. Después se desecha. El prototipo evolutivo suele incluir lo que se ha aprendido haciendo prototipos desecharables. Es decir, un prototipo evolutivo es riguroso y se desarrolla usando bases sólidas (obtenidas mediante prototipos desecharables) y el objetivo último es incluirlo en el sistema.

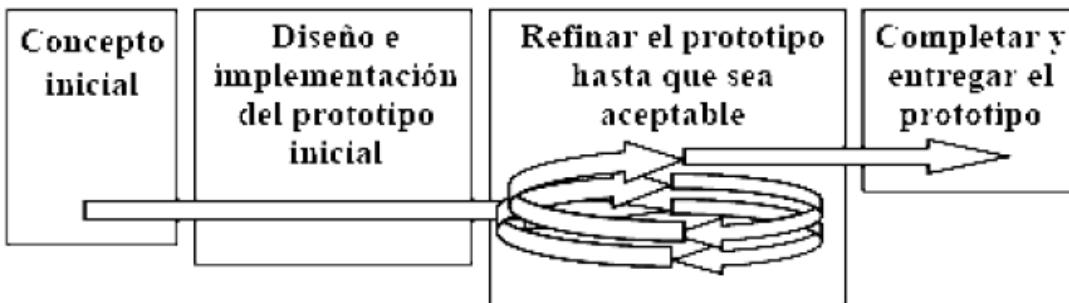


Fig. 6.2 – Modelo de prototipado evolutivo

Capítulo 7

Gestión del Proyecto

7.1 Introducción

La gestión del proyecto se refiere a la utilización de técnicas y actividades de gestión requeridas para obtener un producto *software* de alta calidad, de acuerdo a las necesidades de los usuarios, dentro de un presupuesto y con una planificación de tiempos establecidos previamente. El propósito de la gestión de proyecto es proporcionar la información necesaria para controlar el producto software; describe el plan global usado para el desarrollo del mismo.

La planificación del producto software se refiere a la identificación de actividades, hitos y entregas de un proyecto, por lo tanto, se debe bosquejar un plan para guiar el desarrollo hacia las metas del mismo.

El avance de la aplicación para Android se realizará periódicamente y se refinará antes del comienzo de cada iteración.

7.2 Procedimiento

Puntos de Función:

Los Puntos de Función permiten estimar el tamaño del sistema a partir de sus requerimientos. Pretende medir la funcionalidad entregada al usuario independientemente de la tecnología utilizada para la construcción y explotación del software.

El análisis de Puntos de Función se realiza teniendo en cuenta cinco parámetros básicos:

- Datos Lógicos Internos (ILF, *Internal Logic File*)
- Datos de Interface Externa (EIF, *External Interface File*)
- Salidas Externas (EO, *External Output*)
- Entradas Externas (EI, *External Input*)
- Consultas Externas (EQ, *External Query*)

Datos Lógicos Internos (ILF, *Internal Logic File*): grupo de datos relacionados lógicamente e identificables por el usuario, que residen enteramente dentro de los límites del sistema y se mantienen a través de las Entradas Externas.

Datos de Interfase Externa (EIF, *External Interface File*): grupo de datos relacionados lógicamente e identificables por el usuario, que se utilizan solamente para fines de referencia. Los datos residen enteramente fuera de los límites del sistema y se mantienen por las Entradas Externas de otras aplicaciones.

Salidas Externas (EO, External Output): un proceso elemental con componentes de entrada y de salida mediante el cual datos simples y datos derivados (que se calculan a partir de otros datos) cruzan la frontera del sistema desde adentro hacia afuera. Adicionalmente, las Salidas Externas pueden actualizar un Archivo Lógico Interno.

Entradas Externas (EI, External Input): un proceso elemental en lo que ciertos datos cruzan la frontera del sistema desde afuera hacia adentro.

Consultas Externas (EQ, External Query): un proceso elemental con componentes de entrada y de salida donde un Actor del sistema rescata datos de uno o más Archivos Lógicos Internos o Archivos de Interfaz Externos. Los datos de entrada no actualizan ni mantienen ningún archivo (lógico interno o de interfaz externo) y los datos de salida no contienen datos derivados (es decir, los datos de salida son básicamente los mismos que se obtienen de los archivos). Dentro de éste tipo de transacción entran los listados y las búsquedas de los sistemas.

Las salidas externas y consultas externas se diferencian en que las salidas externas producen valor agregado (por ejemplo agrupan datos) mientras que las consultas externas solo toman el contenido de archivos internos y lo presentan (por ejemplo un listado).

Para poder realizar la estimación se identifican las funcionalidades que ofrece el sistema y se puntúa a cada una de ellas. La siguiente tabla (tabla 7.1) muestra el puntaje que corresponde a cada funcionalidad dependiendo de la complejidad, estos valores se encuentran estandarizados por IFPUG (*International Function Point Users Group*).

Tabla 7.1 – Valores estándar IFPUG.

Tipo / Complejidad	Baja	Media	Alta
(EI) Entrada externa	3 PF	4 PF	6 PF
(EO) Salida Externa	4 PF	5 PF	7 PF
(EQ) Consulta externa	3 PF	4 PF	6 PF
(ILF) Archivo lógico interno	7 PF	10 PF	15 PF
(EIF) Archivo de interfaz externo	5 PF	7 PF	10 PF

A continuación se asigna un nivel de complejidad para cada componente, estos niveles dependen de factores como por ejemplo número de campos no repetidos, número de archivos a ser leídos, creados o actualizados, etc. A mayor número de factores mayor número de complejidad. Los umbrales para pasar de un grado a otro son particulares a este producto software.

Teniendo en cuenta lo anterior mencionado se listan las funcionalidades que ofrece el sistema detallando el tipo, la cantidad, la complejidad y los puntos de función que corresponde a cada una de ellas.

Tabla 7. 2 – Funcionalidades del sistema y su complejidad.

Funcionalidad	Tipo	Complejidad	Puntaje
Detección de imágenes 2D y visualización de elementos virtuales	EO	Alta	7
Total Puntos de Función sin ajustar (PFSA)			7

Notas:

- La especificación del sistema no posee ILF debido a que no guarda ningún tipo de dato.
- La especificación del sistema no posee EIF debido a que no tiene una aplicación previa existente de la cual obtener los datos necesarios ni consulta información alguna a ningún sistema externo.

Ajuste de Puntos de Función:

Una vez obtenidos los puntos de función se debe realizar un ajuste, teniendo en cuenta las características generales del sistema y su grado de influencia en el mismo. Cada una de éstas características aporta un valor entre 0 y 5, de acuerdo a la importancia que tenga en el sistema.

El TDI (*Total Degree of Influence*), es el ajuste de complejidad técnica y es un número que se obtiene de considerar 14 factores de complejidad y sumando los mismos.

Comunicación de datos: hay una facilidad de comunicación de datos ya que la aplicación corre en una terminal por usuario. Puntaje: 3.

Procesamiento distribuido: No corresponde. Puntaje: 0.

Objetivos de rendimiento: El tiempo de respuesta y volumen de procesamiento son ítems críticos. Rendimiento y requisitos de diseño han sido definidos previamente contemplando que los tiempos de respuesta dependen de la tecnología empleada. Puntaje: 4.

Configuración del equipamiento: Son necesarias especificaciones mínimas de procesador para que la aplicación corra. Puntaje: 3.

Tasa de transacciones: No existe un periodo pico de transacciones. Puntaje: 0.

Entrada de datos en línea: El fuerte de la aplicación es la interactividad con el usuario. Puntaje: 5.

Interfaz con el usuario: Se brinda facilidad en el manejo de las funciones del sistema sin exigir al usuario conocimientos previos al mismo. Puntaje: 4.

Actualizaciones en línea: No existen ILF. Puntaje: 0.

Procesamiento complejo: Existen procesos matemáticos matriciales complejos en la aplicación. Puntaje: 5.

Reusabilidad del código: Se utiliza codificación reutilizable dentro de la aplicación. Puntaje: 2.

Facilidad de instalación: Ninguna consideración especial fue establecida por el usuario. Puntaje: 0.

Facilidad de operación: No se definieron, por parte del usuario, necesidades especiales de operación o de respaldo. Puntaje: 0.

Instalaciones múltiples: La aplicación funcionará bajo entornos idénticos de Hardware y Software. Puntaje: 2.

Facilidad de cambios: No existe ninguna especificación por parte de los usuarios en este sentido. Puntaje: 0.

Tabla 3.4.2.3 – Factor de ajuste.

Factores	Valores
Comunicación de Datos	3
Función Distribuida	0
Rendimiento	4
Configuración Fuertemente Usada	3
Frecuencia de Transacciones	0
Entrada de Datos On-Line	5
Eficiencia del Usuario Final	4
Actualización On-line	0
Procesos Complejos	5
Reutilización	2
Facilidad de Instalación	0
Facilidad de Operación	0
Instalación en distintos lugares	2
Facilidad de Cambio	0
TDI	28

Cálculo de Factor de Ajuste (AF):

$$AF = (TDI * 0,01) + 0,65$$

$$AF = (28 * 0,01) + 0,65$$

AF = 0,93

Cálculo de Puntos de Función Ajustados (PFA):

$$PFA = PFSA * AF$$

$$PFA = 7 * 0,93$$

$$PFA = 6,51$$

$$PFA \approx 7$$

Con los puntos de función obtenidos, pasamos a estimar las líneas de código (LOC, *Lines of Code*). Para ello, estimamos que nuestro sistema estará constituido en un 100% en un lenguaje orientado a objetos (C#) según la página <http://www.qsm.com/FPGearing.html>:

$$LOC/FP (C\#) = 54$$

Cálculo de LOC:

Sabiendo que:

$$RATIO = LOC/PFA$$

$$LOC = RATIO * PFA$$

$$LOC (C\#) = 54 * 6,51$$

$$LOC (C\#) = 351,54$$

$$LOC = 351,54$$

Líneas de código en miles (KLOC):

$$KLOC = 0,351$$

Método de COCOMO:

El método de COCOMO nos permitirá estimar el coste, esfuerzo y tiempo que insumirá la elaboración de nuestro proyecto. En éste caso utilizaremos el método en su modo orgánico y modelo básico ya que es el que se ajusta al proyecto a desarrollar:

Modo orgánico: un pequeño grupo de programadores experimentados desarrollan software en un entorno familiar. El tamaño del software varía desde unos pocos miles de líneas (tamaño pequeño) a unas decenas de miles (medio).

Modelo básico: Se utiliza para obtener una primera aproximación rápida del esfuerzo y hace uso de los siguientes valores de constantes para calcular distintos aspectos de costes:

a = 2,40.

b = 1,05.

c = 2,50.

d = 0,38.

Estos valores son para las fórmulas:

Esfuerzo en meses-hombre para llevar adelante el producto software:

$$MM = a * KLOC^b$$

Tiempo de desarrollo del producto software: $TDEV = c * MM^d$

Personas necesarias para realizar el producto software: $CosteH = MM/TDEV$

Haciendo los cálculos correspondientes tenemos que:

$$MM = 0,8$$

$$TDEV = 2,3$$

$$CosteH = 0,34 \approx 1$$

7.3 Planificación del Proyecto

7.3.1 Introducción

La gestión efectiva de un proyecto de software depende de planificar completamente el progreso del proyecto. El gestor del proyecto debe anticiparse a los problemas que puedan surgir, así como preparar soluciones a esos problemas. Un plan, preparado al inicio de un proyecto, debe utilizarse como un conductor para el proyecto. Este Plan Inicial debe ser el mejor posible de acuerdo con la información disponible. Éste evolucionará conforme el proyecto progrese y la información sea mejor.

7.3.2 Plan del Desarrollo

El plan del proyecto fija los recursos disponibles, divide el trabajo y crea un calendario de trabajo. El desarrollo se lleva a cabo basándose en fases. En cada fase se tienen distintos esfuerzos, y se debe tener en cuenta que muchas de las actividades a realizar se pueden hacer en paralelo.

▪ Fase 1: Gestión y Planificación Inicial.

En esta fase se estudia y planifica la realización del proyecto. Se establecen los objetivos y el alcance.

- Estudio de Objetivos: se establecen los objetivos que se quieren lograr con la realización del proyecto.
 - Determinación del Alcance: se fija el trabajo que será necesario para lograr los objetivos establecidos.
 - Planificación: Organización global de las fases del proyecto y la estimación del tiempo dedicado a cada una de ellas.
- **Fase 2: Estudio de alternativas.**
- En esta fase se hace un estudio de las alternativas que existen actualmente para la realización de aplicaciones Android y para implementar la tecnología de realidad aumentada.
- Realización del estudio: Se reúne la mayor cantidad posible de información para poder decidir qué herramientas se escogerán para la realización del proyecto y cómo se utilizarán.
 - Elección: se escoge la opción que mejor se amolda a las necesidades del proyecto.
- **Fase 3: Formación.**
- Estudio de las tecnologías y herramientas que se utilizan en el proyecto. Consistente en la lectura de documentación y en la realización de tutoriales para familiarizarse con el entorno de desarrollo y lenguaje de programación.
- Principalmente se ha estudiado el *framework* que se ha elegido, el sistema Android y el modo en el que se podrían desarrollar las aplicaciones para la realización del proyecto mediante la creación de ejemplos básicos.
- **Fase 4: Diseño y pruebas.**
- Implementación de la aplicación en la que se usarán las técnicas acordes a la elección hecha en la Fase 2.
- Diseño: diseño de la lógica y los módulos necesarios para implementar la aplicación.
 - Implementación: una vez realizado un correcto diseño de la aplicación se procede a desarrollar el código para obtener la aplicación completa.
 - Validación: se prueba el funcionamiento de la aplicación mediante testeo.
- **Fase 5: Informe Final.**
- Recopilación de toda la información obtenida, utilizada y generada así como la elaboración del documento escrito el proyecto.

Capítulo 8

Modelización y Diseño

8.1 Actores

Un actor interactúa con el sistema, siendo el actor un usuario u otro sistema, los actores identificados son:

- Usuario/s

8.2 Diagrama de contexto

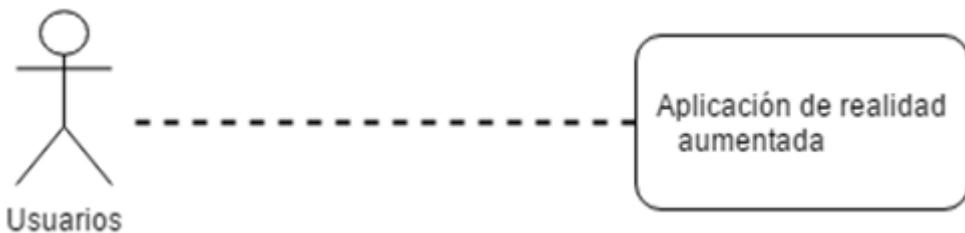


Fig. 8.1 – Diagrama de contexto

8.3 Listado de casos de uso

Casos de uso identificados:

- 1) Detección de imágenes y visualización de elementos virtuales.

8.4 Diagrama de casos de uso

Se muestra un diagrama de casos de uso general. Luego se va a explorar a cada uno de ellos en profundidad.

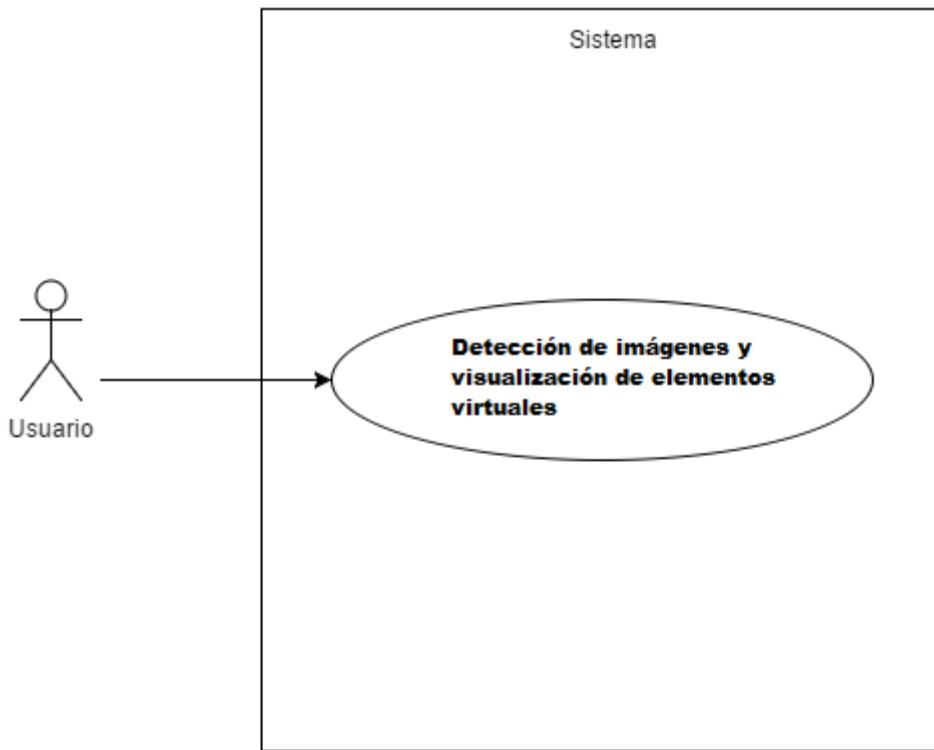


Fig. 8.2 – Diagrama de casos de uso

8.5 Descripción Textual de los Casos de Uso

Tabla 8.1 – Caso de uso: Detección de imágenes 2D y visualización de Realidades Aumentadas

CU_01: Detección de imágenes y visualización de elementos virtuales	
Fecha de creación	01/04/2016
Fecha de modificación	01/04/2016
Versión	1
Resumen	Permite al usuario tener la cámara del dispositivo lista para enfocar los <i>targets</i> que forman parte del juego para PC y así poder visualizar, en tiempo real, la información virtual asociada.
Personas involucradas y metas	Usuarios: quieren hacer uso de la funcionalidad de la aplicación que permite ver la descripción del enemigo o nivel del juego mediante realidad aumentada.
Precondiciones	El usuario debe haber iniciado la aplicación y esta estar en ejecución en la pantalla principal.

Postcondiciones	Ninguna.						
Escenario principal	<ol style="list-style-type: none"> 1. Un usuario desea hacer uso de la funcionalidad de la aplicación que permite ver la descripción del enemigo o nivel mediante realidad aumentada. 2. El usuario presiona el botón “Start”. 3. El sistema abre la cámara del dispositivo y comienza a tomar fotogramas. 4. Se analizan los fotogramas con algoritmos de visión y se busca si hay coincidencias en “Recursos Targets”. 5. Si el usuario ha enfocado un <i>target</i> perteneciente a la aplicación, el sistema asocia y posiciona el contenido virtual “sobre” ese <i>target</i>, combinando lo real con lo virtual. 6. El sistema renderiza y muestra en la pantalla del dispositivo, en tiempo real, la imagen resultante de la combinación. 						
Escenario alternativo	<ol style="list-style-type: none"> 5. a. Si el usuario no enfoca ningún <i>target</i> perteneciente a la aplicación, el sistema simplemente deja ver por pantalla lo que la cámara está tomando. 						
Requisitos de interfaz de usuario	Dispositivo móvil.						
Requisitos funcionales	<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding: 5px;">Tiempo de respuesta</td> <td style="padding: 5px;">La interfaz debe responder dentro de un tiempo máximo de 1 segundo en detectar cierto <i>target</i>.</td> </tr> <tr> <td style="padding: 5px;">Concurrencia</td> <td style="padding: 5px;">No hay concurrencia. Cada usuario tiene su propia versión de app instalada en su móvil.</td> </tr> <tr> <td style="padding: 5px;">Disponibilidad</td> <td style="padding: 5px;">100%.</td> </tr> </table>	Tiempo de respuesta	La interfaz debe responder dentro de un tiempo máximo de 1 segundo en detectar cierto <i>target</i> .	Concurrencia	No hay concurrencia. Cada usuario tiene su propia versión de app instalada en su móvil.	Disponibilidad	100%.
Tiempo de respuesta	La interfaz debe responder dentro de un tiempo máximo de 1 segundo en detectar cierto <i>target</i> .						
Concurrencia	No hay concurrencia. Cada usuario tiene su propia versión de app instalada en su móvil.						
Disponibilidad	100%.						

8.6 Diagrama de Actividad

Para cada caso de uso se realiza su diagrama de actividad.

- 1) Caso de uso: Detección de imágenes y visualización de elementos virtuales



Fig. 8.3 – Diagrama de actividad 1

8.7 Diagrama de Secuencia

Para la lectura de los diagramas se emplearán entidades que representan a los subsistemas en lugar de clases.

- 1) Caso de uso: Detección de imágenes y visualización de elementos virtuales

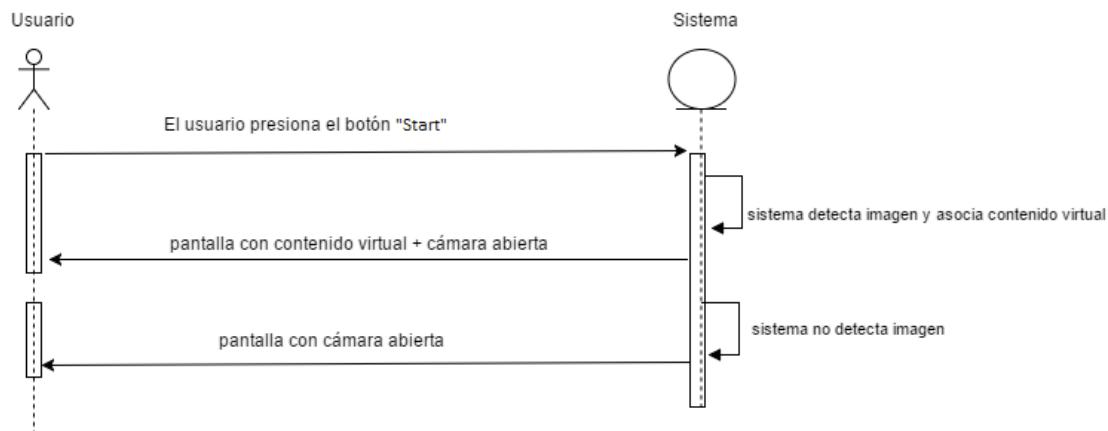
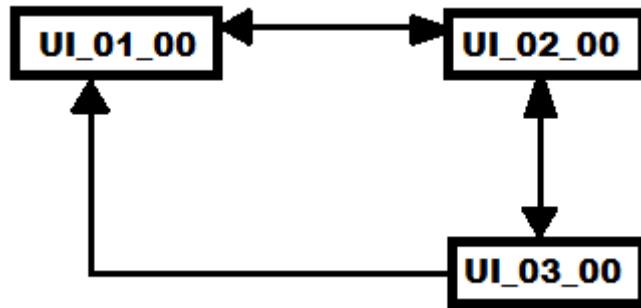


Fig. 8.4 – Diagrama de secuencia 1

8.8 Identificación de Roles

Rol Usuario: tiene permisos para realizar todas las funciones del sistema.

8.9 Guiones y Escenarios: Diagrama de Transición de Escenarios



Flg. 8.5 – Diagrama de Transición de Escenarios

8.10 Tabla de Transición de Escenarios

Tabla 8.2 – Tabla de transición de escenarios.

Escenarios	IU_01_00	IU_02_00	IU_03_00
IU_01_00	IU_01_01/02	IU_01_03	-
IU_02_00	IU_02_01/02/04	-	IU_02_03
IU_03_00	IU_03_01	IU_03_02	-

8.11 Descripción de escenarios y objetos de escenarios

- Escenario IU_01_00: Inicio de aplicación



Fig. 8.6 – Descripción Escenario IU_01_00

Si la aplicación, en cualquier momento de ejecución, se para por algún error interno, volverá a este escenario IU_01_00.

En este escenario al presionar el botón IU_01_03, que es el ícono que funciona como acceso directo de la app FOWAPP, llevará hasta el escenario IU_02_00.

Si el usuario presionar el botón IU_01_01 (botón *Home*), o IU_01_02 (botón *back*), se volverá inmediatamente al escenario IU_01_00.

- Escenario IU_02_00: Pantalla principal

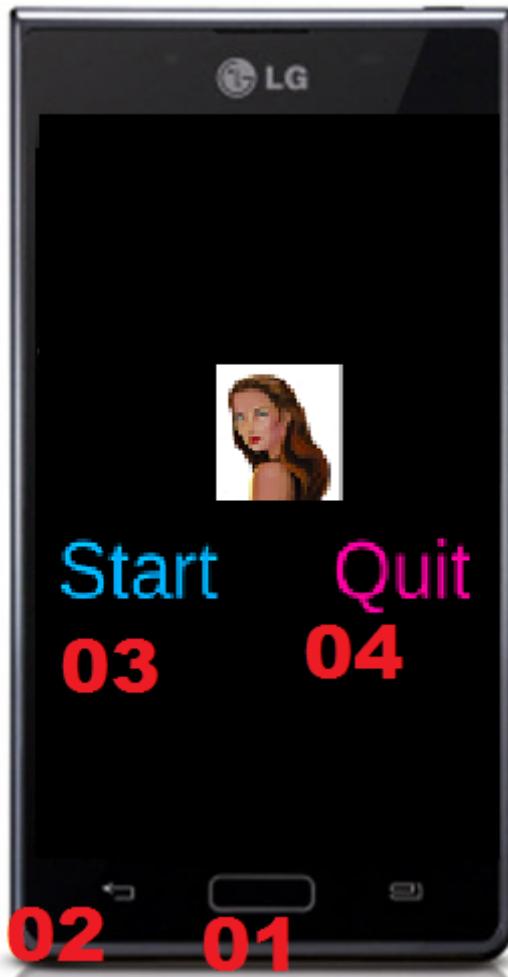


Fig. 8.7 – Descripción Escenario IU_02_00

En este escenario al presionar el botón IU_02_03 llevará al Escenario IU_03_00.

Si se aprieta el botón IU_02_02 (botón *back*), o el botón IU_02_04, o en cualquier momento el botón IU_02_01 (botón *Home*), hará regresar al escenario IU_01_00.

- Escenario IU_03_00: Pantalla para detectar las imágenes y asociarlas a elementos virtuales. Es simplemente la cámara del dispositivo abierto que deja ver el mundo físico y según lo que detecte, mostrará en pantalla los correspondientes contenidos virtuales. Por lo tanto lo que cambiará será

solamente la información virtual que se muestre luego del reconocimiento de imagen.



Fig. 8.8 – Descripción Escenario IU_03_00

El botón *Home* IU_03_01 hará volver al escenario IU_01_00 pero la aplicación quedará en estado de ejecución en segundo plano o *background*, es decir, que si se vuelve a presionar el acceso directo de la app FOWAPP (botón IU_01_03), la aplicación se abrirá en la última pantalla en la que se haya estado antes de presionar el botón *Home*. El botón *Back* IU_03_02 hará regresar al escenario IU_02_00.

Capítulo 9

Implementación y Pruebas

9.1 Herramienta Principal para el Desarrollo de la aplicación

9.1.1 Introducción

El *SDK* (Kit de Desarrollo de Software) Vuforia para Unity permite a los desarrolladores crear aplicaciones de realidad aumentada fácilmente usando el motor Unity.



Fig. 9.1 – Logo oficial de Vuforia

Vuforia es una plataforma de software que permite a las aplicaciones añadir fácilmente la funcionalidad de la visión artificial avanzada, lo que permite reconocer imágenes y objetos, o reconstruir ambientes en el mundo real. Si se está construyendo un juego, Vuforia tiene toda la funcionalidad y el rendimiento para satisfacer las necesidades. Las características claves incluyen la capacidad de reconocer y realizar un seguimiento de imágenes, objetos, texto, marcadores y reconstruir entornos. La Fig. 9.2 muestra las distintas opciones que hay para descargar del kit de desarrollo de software Vuforia.

Vuforia 5.5 SDK

Use the Vuforia SDK to build Android and iOS applications for mobile devices and digital eyewear. Apps can be built with Android Studio (Java/C++), XCode (C++), and Unity, the cross-platform game engine.



[Download for Android](#)

vuforia-sdk-android-5-5-9.zip (6.99 MB)



[Download for iOS](#)

vuforia-sdk-ios-5-5-9.zip (17.10 MB)



[Download for Unity](#)

vuforia-unity-5-5-9.unitypackage (34.50 MB)

Fig. 9.2 – SDK Vuforia (versión 5.5) disponibles en su web oficial

Hay tres componentes principales en la plataforma Vuforia:

1. El motor Vuforia: el motor es la biblioteca de cliente que está estéticamente vinculada a la aplicación. Este servicio está disponible a través del *SDK* del cliente y es compatible con Android y también iOS. Puede usar Android Studio, Xcode o Unity para construir aplicaciones.
2. Herramientas: Vuforia proporciona herramientas para la creación de *Targets*, gestión de bases de datos de destino y la protección de las licencias de aplicación. El escáner de objetos Vuforia (disponible para Android), ayuda a digitalizar fácilmente objetos 3D en un formato de destino que es compatible con el motor Vuforia. El *Target Manager* es una aplicación web en el portal de desarrolladores que permite crear bases de datos de *Targets* para su uso en el dispositivo y la nube (para un gran número de *Targets*). También existe otra herramienta para el caso de los desarrolladores de aplicaciones *see-through*.
3. Nube de Reconocimiento: Vuforia también ofrece un servicio de reconocimiento de la nube para cuando la aplicación necesita reconocer un gran conjunto de imágenes o si la base de datos se actualiza con frecuencia. La *API* de *Vuforia Web Services* permite administrar estas grandes bases de datos de imagen en la nube de forma eficiente y le permite automatizar los flujos de trabajo mediante la integración directa en sus sistemas de gestión de contenidos.

Las capacidades de reconocimiento y seguimiento de Vuforia se pueden utilizar en una variedad de imágenes y objetos:

- Los *Image Target*: son imágenes planas.
- Los *Multi-Target*: se crean utilizando más de una imagen de destino y se pueden organizar en formas geométricas regulares (por ejemplo, cajas) o en cualquier disposición arbitraria de superficies planas.

- Los *Cylinder Targets*: son imágenes envueltas sobre los objetos que son aproximadamente de forma cilíndrica (por ejemplo, botellas de bebidas, tazas de café, etc).
- El Reconocimiento de texto permite desarrollar aplicaciones que reconocen palabras de un diccionario de aproximadamente 100.000 palabras en inglés.
- Vuforia puede reconocer y realizar un seguimiento de una gama más amplia de objetos 3D también.
- Además de dirigir el reconocimiento, Vuforia proporciona un conocimiento y comprensión del entorno físico del usuario. Terreno Inteligente es una tecnología innovadora que puede reconstruir el entorno físico del usuario como una malla 3D. Permite a los desarrolladores crear una nueva clase de juegos y experiencias de visualización realista de productos, en los que el contenido puede interactuar con los objetos físicos y superficies en el mundo real.

9.1.2 Image Target



Fig. 9.3 – Reconocimiento de una imagen plana la cuál muestra una tetera como objeto virtual (una vez reconocida esta)

El *SDK* de Vuforia detecta y rastrea las características que se encuentran de forma natural en la propia imagen mediante la comparación de estas características naturales con una base de datos de *Image Targets* conocida.

Los *Image Target* pueden ser creados con el *Vuforia Target Manager* usando JPG o PNG imágenes en RGB o escala de grises. El tamaño de las imágenes de entrada debe ser de 2 MB o menos. Las características extraídas de estas imágenes se almacenan en una base de datos, que luego se puede descargar y se envasa junto con la solicitud. La base de datos puede ser utilizada por Vuforia para las comparaciones en tiempo de ejecución. Los *Image Target* deben ser vistos bajo una luz brillante y moderadamente difusa, la superficie de la imagen debe ser uniformemente iluminada.

Hay dos fases en el desarrollo con *Image Targets*. Primero hay que diseñar (o seleccionar) la imagen destino y luego subirla al *Vuforia Target Manager* para el procesamiento y evaluación de la misma. Los *Image Targets* pueden

usar cualquier imagen plana que proporcione suficiente detalle como para ser detectada por el *SDK Vuforia*.

9.1.3 Subir y manejar los *Image Targets*

Los *Image Target* se pueden incorporar en las aplicaciones utilizando una base de datos de dispositivos embebidos o una base de datos de la nube en línea:

- Bases de datos de dispositivos: Crear *Image Targets* con el Administrador de destino para usar en bases de datos de dispositivos.
- Bases de datos en la nube (Fig. 9.4): Crear *Image Targets* para usar en bases de datos de reconocimiento de la nube. La nube de reconocimiento de Vuforia permite a los desarrolladores alojar y administrar los *Image Targets* en línea, es ideal para aplicaciones que utilizan muchas de los *Targets* que deben actualizarse con frecuencia.

Las Aplicaciones de reconocimiento de la nube utilizan las mismas *APIs* para el manejo de *Image Targets* que las aplicaciones que utilizan bases de datos de dispositivos. La diferencia significativa entre las aplicaciones de reconocimiento de la nube y las de la base de datos de dispositivos es que las aplicaciones de reconocimiento de la nube utilizan la clase *TargetFinder* que gestiona la ejecución de consultas y manejo de resultados de la consulta.

Los *Image Target* se crean utilizando el *Vuforia Target Manager* que procesará las imágenes para generar una representación de los datos de sus características. También proporcionará una calificación de detección esperada del *Target* y el *Tracking*.

<input type="checkbox"/> Target Name	Type	Rating	Status	Date Modified
<input type="checkbox"/> 400x400_Tower2	Single Image	★★★★★	Active	Apr 10, 2016 05:09
<input type="checkbox"/> 400x400_Tower	Single Image	★★★★★	Active	Apr 10, 2016 05:03
<input type="checkbox"/> 400x400SinCabeza	Single Image	★★★★★	Active	Apr 10, 2016 04:55
<input type="checkbox"/> 400x400_Galatea	Single Image	★★★★★	Active	Apr 10, 2016 04:55
<input type="checkbox"/> 400x400_Skeleton	Single Image	★★★★★	Active	Apr 10, 2016 04:54
<input type="checkbox"/> 400x400_Spider	Single Image	★★★★★	Active	Apr 10, 2016 04:54
<input type="checkbox"/> 400x400_Zombie	Single Image	★★★★★	Active	Apr 10, 2016 04:54
<input type="checkbox"/> 400x400_Golem	Single Image	★★★★★	Active	Apr 10, 2016 04:53

Fig. 9.4 – Los Image Target del juego subidos a la base de datos en la nube con sus respectivas clasificaciones

9.1.4 Atributos de un Image Target ideal

Los *Image Target* que posean los atributos siguientes permitirán el mejor rendimiento de detección y *Tracking* (seguimiento) desde el *SDK Vuforia*:

- Rica en detalles (ejemplo: escena de la calle, grupo de personas, collages y mezclas de elementos y escenarios deportivos). Fig. 9.5.
- Buen contraste (ejemplo: regiones brillantes y oscuras, y bien iluminado).
- No debe haber patrones repetitivos (ejemplo de imágenes no recomendables: un campo cubierto de hierba, la fachada de la casa moderna con ventanas idénticas, un tablero de ajedrez).
- El formato debe ser de 8 y 24 bits PNG y JPG (RGB o escala de grises), menos de 2MB de tamaño.

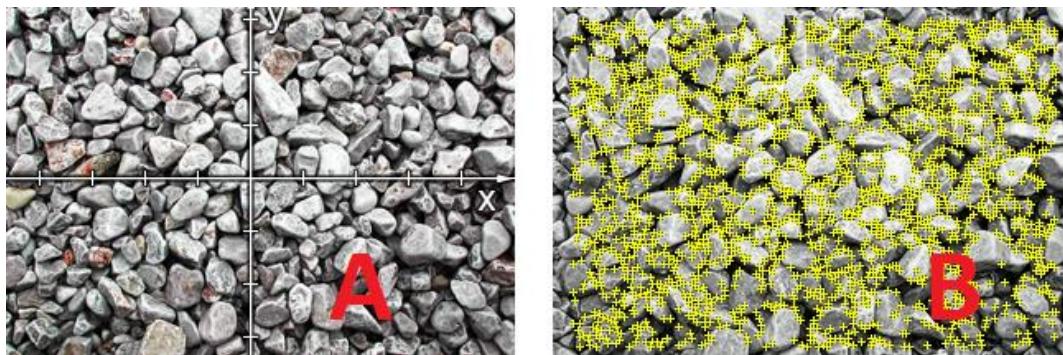


Fig. 9.5 – A: Esta imagen se introduce en el *Target Manager*. B: Se muestran las características naturales que el *SDK Vuforia* utiliza para detectar el *Image Target*

9.1.5. Características y puntuaciones de los Image Target

Una calificación define lo bien que una imagen puede ser detectada y rastreada utilizando el *SDK Vuforia*. Esta calificación se muestra en el *Image Target* para cada Target subido a través de la web de la *API*. La calificación puede variar de 0 a 5 estrellas para cualquier imagen dada. Cuanto más alta sea la calificación, mayor será la capacidad de detección y seguimiento que contiene. Un valor de cero indica que un objetivo no se sigue en absoluto por el sistema AR, mientras que un número de 5 estrellas indica que una imagen es fácilmente rastreada por el sistema de RA (Fig. 9.6).

Imagen subida	Imagen analizada	Grado de la estrella

Fig. 9.6 – La imagen con mejor calificación tiene más características (puntos amarillos)

Más detalles a tener en cuenta:

- El contraste local: El contraste local es a menudo difícil de detectar con el ojo. Las formas orgánicas, detalles redondos, imágenes borrosas o de alta compresión a menudo no proporcionan suficiente riqueza de detalles para ser detectado y rastreado correctamente.
- Evitar los modelos repetidores: Como se mencionó anteriormente, aunque algunas imágenes contienen suficientes características y buen contraste, los patrones repetitivos dificultan los resultados de detección. Para obtener los mejores resultados, es preferible usar una imagen sin motivos repetidos. Un tablero de ajedrez es un ejemplo de un patrón repetido de que no se puede detectar, ya que los pares de 2x2 cuadrados blancos y negros tienen exactamente el mismo aspecto y no se pueden distinguir por el detector.

9.1.6 Vuforia Target Manager

El *Vuforia Target Manager* es una herramienta basada en la web que permite crear y gestionar bases de datos de destino en línea. También puede administrar las asociaciones de bases de datos de claves de licencia utilizando el Gestor de destino.

Se utiliza el *Vuforia Target Manager* para:

- Crear dispositivos y bases de datos de la nube.
- Asignar a las bases de datos de licencia de claves.
- Añadir *Targets* a bases de datos.
- Editar y eliminar *Targets*.
- Administrar bases de datos.
- Etc.

Para comenzar a trabajar con el *Target Manager*, se necesita una cuenta de desarrollador Vuforia (el registro es gratis). Todas las bases de datos creadas en el *Target Manager* deben estar asociadas a una clave de licencia (una vez activada la cuenta, se pueden crear licencias propias).

Luego de tener la clave de licencia activada, se puede:

- Crear una base de datos.
- Añadirle *Targets*.
- Descargar el package para Unity de la base de datos y utilizarla en el motor.
- Actualizar en cualquier momento dichas bases de datos creadas.

9.1.7 Arquitectura de la aplicación para Android usando Vuforia

La Fig. 9.7 muestra la arquitectura general de funcionamiento de la aplicación usando Vuforia, la misma resume lo explicado anteriormente desde el comienzo de la sección 2 hasta ahora.

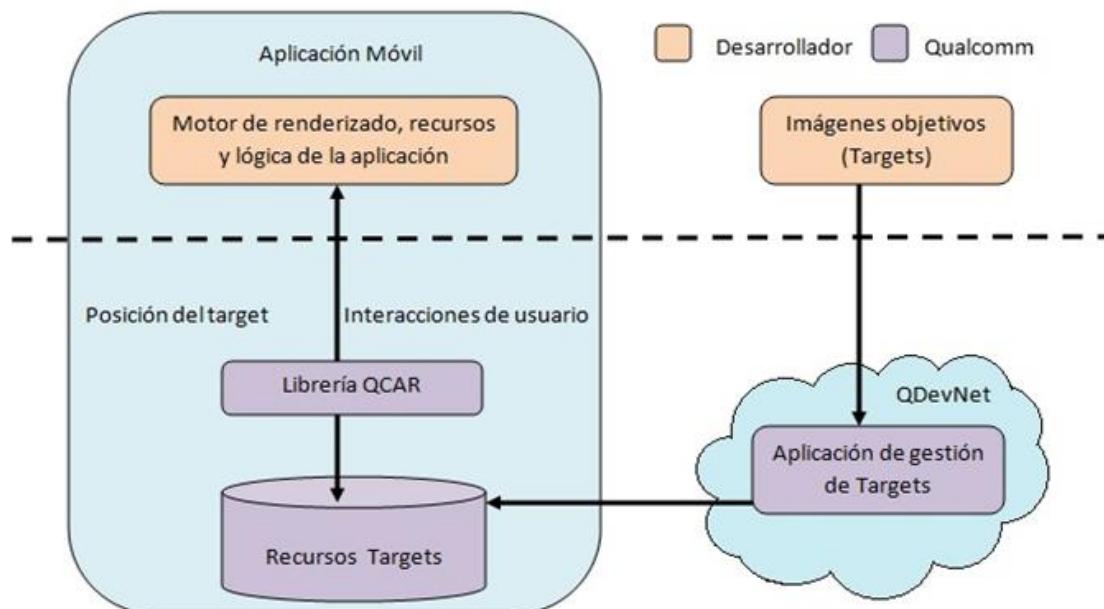


Fig. 9.7 – Arquitectura general de funcionamiento de la aplicación usando Vuforia

Luego, en la Fig. 9.8 se pueden ver los componentes esenciales de una aplicación RA basada en Vuforia:

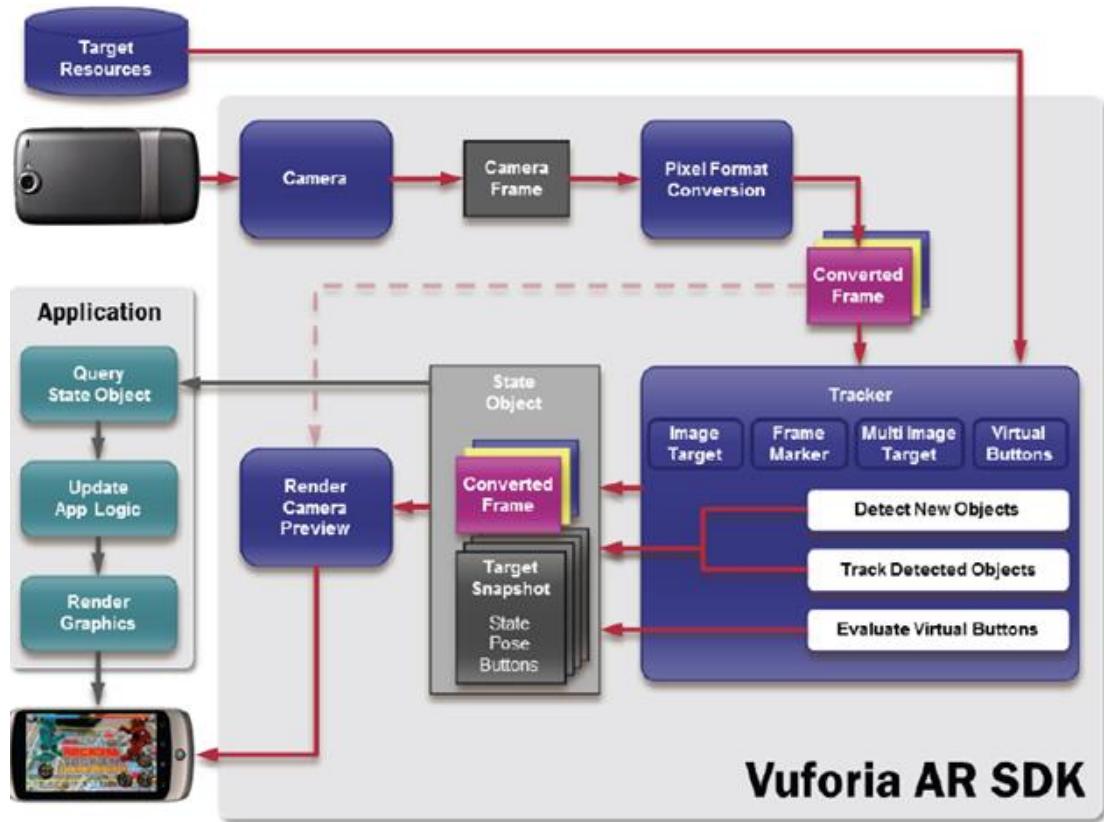


Fig. 9.8 – Arquitectura de Vuforia AR SDK

A continuación se describirán términos referentes a la arquitectura de clases básicas de Vuforia. Muchos de estos componentes se tratan de componentes *singletons* (clases que tienen una sola instancia).

- **Cámara (singleton):** la cámara se asegura que cada *frame* es capturado y pasado eficientemente al *Tracker*. El desarrollador solo ha de decir a la instancia de la cámara con los métodos oportunos cuándo ha de comenzar y terminar de capturar. La imagen es entregada automáticamente a un dispositivo dependiente del formato y dimensiones de la imagen.
- **ImageConverter (singleton):** se encarga de convertir la imagen capturada por la cámara en un formato aceptable para que OpenGL ES la renderice y trate. Esta conversión también incluye una reducción del muestreo para tener la imagen de la cámara en diferentes resoluciones para diferentes procesos.
- **Tracker (singleton):** es la parte encargada de detectar y rasterizar la imagen captada mediante algoritmos computacionales de visionado. Partiendo de la imagen captada por la cámara, se preocupa de detectar nuevos marcadores, imágenes objetivos y de evaluar posibles botones virtuales. El resultado es almacenado en un objeto usado por el renderizador de video y que puede también ser accesible desde el código.

- del desarrollador. Puede cargar distintos conjuntos de datos, patrones de reconocimiento, pero solo uno puede estar activo al mismo tiempo.
- Renderizador de Video de Fondo (*singleton*): el renderizador de video de fondo realiza la tarea de analizar la imagen almacenada por la cámara en el objeto de estado. El rendimiento de la representación de vídeo de fondo está optimizado para cada dispositivo específico.
 - *Trackables*:



Fig. 9.9 – Vista de alto nivel de arquitectura de Vuforia

Los *trackables* constituyen la clase base que representa todo objeto real que puede ser seguido en seis grados de libertad por el SDK Vuforia. Como se ve en la Fig. 9.9, hay diferentes tipos de *trackables*. Cada *trackable*, cuando es detectado y seguido, tiene un nombre, un identificador, estado e información sobre su posición. Tanto si es del tipo *Image Targets*, *Multi Targets* o *Markers*, son *trackables* que heredan propiedades de esta clase base (clase *Trackable*). Los *Trackables* se actualizan cada vez que se procesa un *frame* y los resultados se pasan a la aplicación en el objeto de estado (*State*).

Parámetros:

- 1) Tipo de *trackable*: Enumeración (enum) que define el tipo de *trackable*:
 - UNKNOWN_TYPE: *Trackable* de tipo desconocido.
 - IMAGE_TARGET: *Trackable* de tipo *ImageTarget*.
 - MULTI_TARGET: *Trackable* de tipo *MultiTarget*.
 - MARKER: *Trackable* de tipo *Marker*.
- 2) Nombre/identificador de *trackable*: *String* que define de forma única al *Trackable* dentro de la base de datos de *Targets*.
- 3) Estado del *trackable*: Cada *trackable* tiene una información de estado asociada con él en el objeto *State*, el cual se actualiza cada vez que se procesa un *frame*. El estado viene caracterizado por una enumeración:
 - UNKNOWN: El estado del *trackable* es desconocido. Este estado es devuelto normalmente antes de la inicialización del *tracker*.
 - UNDEFINED: El estado del *trackable* no está definido.

- NOT_FOUND: El *trackable* no se encuentra, por ejemplo, esto sucede cuando el *trackable* no está en la base de datos.
 - DETECTED: El *trackable* se ha detectado en el fotograma actual.
 - TRACKED: El *trackable* se ha seguido en el fotograma actual.
- 4) Posición del *trackable*: La posición actual válida de un *trackable* en estado DETECTED o TRACKED se devuelve como una matriz 3x4 en orden de filas mayor. El SDK Vuforia proporciona herramientas para convertir estas matrices de posición en matrices llamadas GL model_view matrix y para proyectar puntos en tres dimensiones desde la escena 3D a la pantalla del dispositivo.

Vuforia emplea un sistema de coordenadas cartesianas a derechas. Cada *Image Target* y *Frame Marker* define un sistema de coordenadas local con (0,0,0) en el centro (medio) del *Target*, constituyendo el origen del sistema de coordenadas. +X va hacia la derecha, +Y hacia arriba y +Z hacia fuera del *trackable* (en la dirección desde la cual puede ser visto).

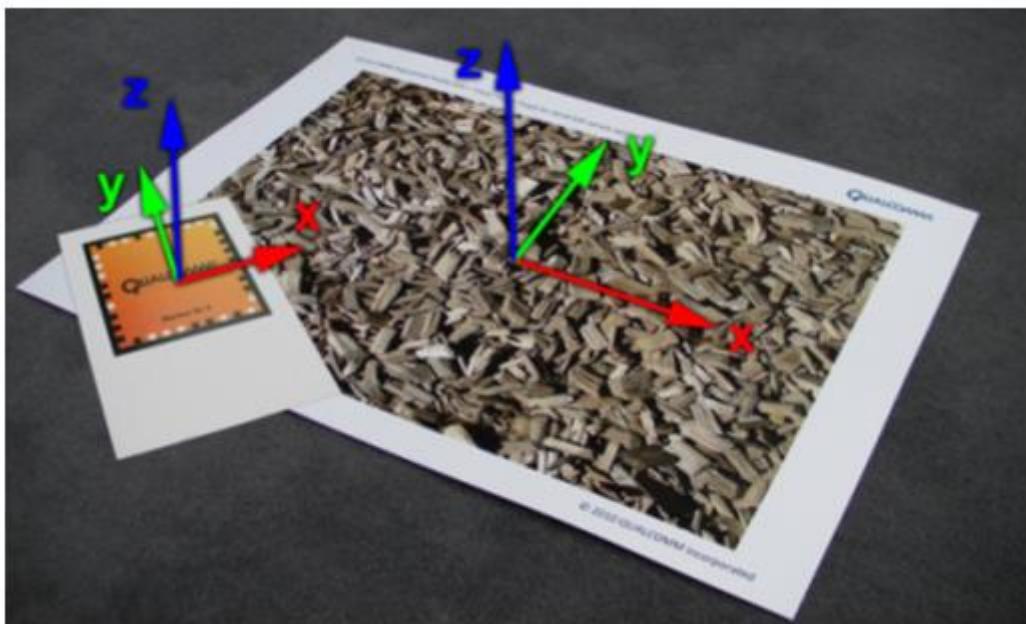


Fig. 9.10 – Sistema de coordenadas de *trackables* (*Marker* e *ImageTarget*)

El origen del sistema de coordenadas local de un *Multi Target* viene definido por sus componentes. Sus partes, constituidas por *Image Targets* son transformadas respecto a este origen. La posición del *Multi Target* devuelta por el sistema, es la posición de este origen, independientemente desde qué parte individual del *Multi Target* esté siendo seguida. Esta característica permite que un objeto geométrico (por ejemplo, una caja) pueda ser seguida continuamente con las mismas coordenadas, incluso si otras partes de un *Image Target* están visibles en la escena y son capturadas por la cámara.

9.2 Otras herramientas

9.2.1 Android SDK



Fig. 9.11 – Logo oficial Android Studio

El *SDK* de Android, incluye un conjunto de herramientas de desarrollo. Las plataformas de desarrollo soportadas incluyen GNU/Linux, Mac OS X 10.5.8 o posterior, y Windows XP o posterior. Una aplicación Android está compuesta por un conjunto de ficheros empaquetados en formato .apk.

El *SDK* soporta también versiones antiguas de Android, por si los programadores necesitan instalar aplicaciones en dispositivos ya obsoletos o más antiguos. Las herramientas de desarrollo son componentes descargables, de modo que una vez instalada la última versión, pueden instalarse versiones anteriores y hacer pruebas de compatibilidad.

9.2.2 Java JDK

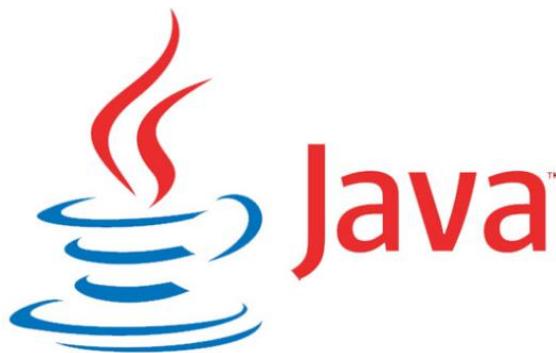


Fig. 9.12 – Logo oficial Java

Primero, hay que recordar que el lenguaje de programación para el sistema operativo Android se trata de un sublenguaje de Java llamado Dalvik, pese a que hoy en día se le conoce más por su aplicación en Android que por el propio lenguaje, por lo que comúnmente se le llama Android. Así pues, el *JDK* de Java es necesario para el desarrollo de esta aplicación.

El *JDK*, *Java Development Kit*, se trata de un conjunto de herramientas de software para la creación del programa en lenguaje Java. Existen distribuciones para todos los sistemas operativo mayoritarios.

9.2.3 OpenGL ES



Fig. 9.13 – Logo de OpenGL ES

OpenGL ES es una librería gráfica que crea una interfaz de bajo nivel flexible y potente entre el software y la aceleración de gráficos.

Android incluye soporte para un alto rendimiento de gráficos 2D y 3D con *Open Graphics Library* (OpenGL), en concreto, la *API* de OpenGL ES. OpenGL es una *API* de gráficos multiplataforma que especifica una interfaz de *software* estándar para el *hardware* de procesamiento de gráficos 3D. OpenGL ES (*OpenGL for Embedded Systems*) es una variante simplificada de la especificación OpenGL destinada para dispositivos embebidos. Android es compatible con varias versiones de la API de OpenGL ES, 1.0, 1.1, 2.0, 3.0 y 3.1.

En este trabajo se utiliza OpenGL ES 2.0 ya que es esta versión la que utiliza el sdk Vuforia para manejar contenido gráfico, además de ser compatible hacia atrás con las versiones 1.0 y versión 1.1.

9.3 Desarrollo de la aplicación

La versión de Vuforia para Unity usada para este proyecto es la 5-0-10. Actualmente, la nueva versión disponible para Unity es la 5-5-9.

Antes de comenzar, se repetirá lo mencionado anteriormente en el capítulo 4.2 Idea de la aplicación:

La aplicación para dispositivo móvil es independiente al ejecutable para PC. Sin embargo, mientras se juega (con el teclado y el mouse) en la computadora, en algunas ocasiones el dispositivo móvil forma parte de la jugabilidad también. Sus funciones son 2:

- Identificar el monstruo en frente y brindar información del mismo.
- Mostrar el escenario de la escena la cual se está jugando actualmente (aplicado en 2 niveles).

Las imágenes a continuación describen las funcionalidades mencionadas (Fig. 9.14 y Fig. 9.15).



Fig. 9.14 – Captura de pantalla de la serie animada Pokémon.

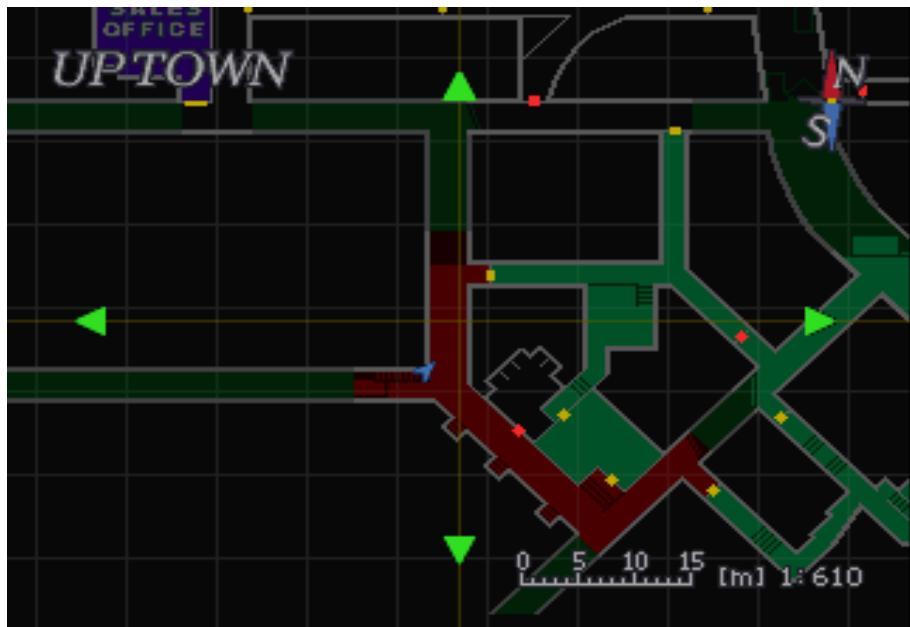


Fig. 9.15 – Mini mapa de un nivel en el juego Resident Evil 3

Fig. 9.14: En un dibujo animado japonés llamado Pokémon, el personaje principal de la historia utilizaba un aparato (un dispositivo móvil llamado Pokédex), el cuál, al enfocar su lente en un animal o criatura perteneciente a la serie (llamados *pokémon*), obtenía del animal un breve resumen de sus características principales (por ejemplo, las zonas donde habitaba).

Fig. 9.15: En varios tipos de juegos, es muy común ver en algún margen de la pantalla, o por ejemplo al pausar el juego, un mini mapa del nivel cuál se está

jugando, sobre todo en juegos largos de aventura, acción o survival horror que utilizan cámaras en primera y tercera persona.

Habiendo analizado estas 2 imágenes previas, se procede a explicar la mecánica del juego de este proyecto. Al hacer click derecho con el ratón, un rayo será lanzado en dirección del eje Z, con origen en el punto (x,y,z) donde se encuentra el personaje actualmente. Si el rayo impacta en un enemigo, el juego se pausará y se mostrará luego de una breve animación, una imagen 2D del monstruo impactado (Fig. 9.16).

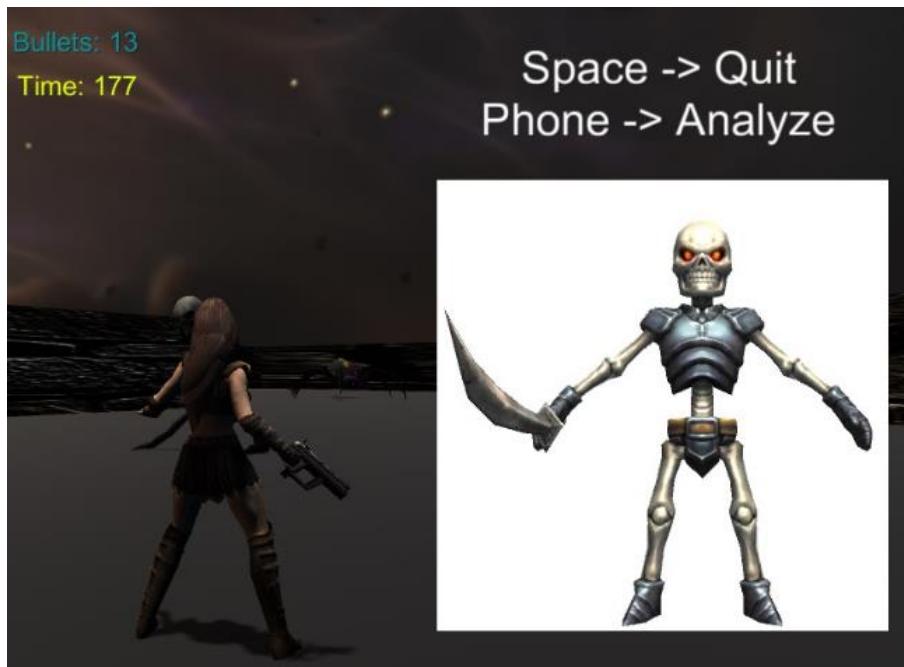


Fig. 9.16 – El rayo lanzado impactó a un enemigo. En este caso, el esqueleto

Al enfocar con el lente del dispositivo móvil sobre la imagen generada en pantalla (una vez ejecutada la aplicación y estando en el escenario IU_03_00), el dispositivo móvil la reconoce y objetos virtuales aparecen en el campo de visión del lente. Así como sucedía en la serie animada japonesa Pokémon, el dispositivo móvil muestra en pantalla el modelo 3D del monstruo animado con un texto que describe al mismo, como ser el daño que este causa al atacar, su velocidad de movimiento o en qué niveles se lo puede encontrar por ejemplo (Fig. 9.17).



Fig. 9.17 – El objeto que se ve en el dispositivo móvil una vez detectada la imagen

Vale mencionar que el monstruo no se encuentra estático, sino que el modelo está animado ya que tiene adjunto una máquina de estados con sus respectivos clips de animaciones (Fig. 9.18).

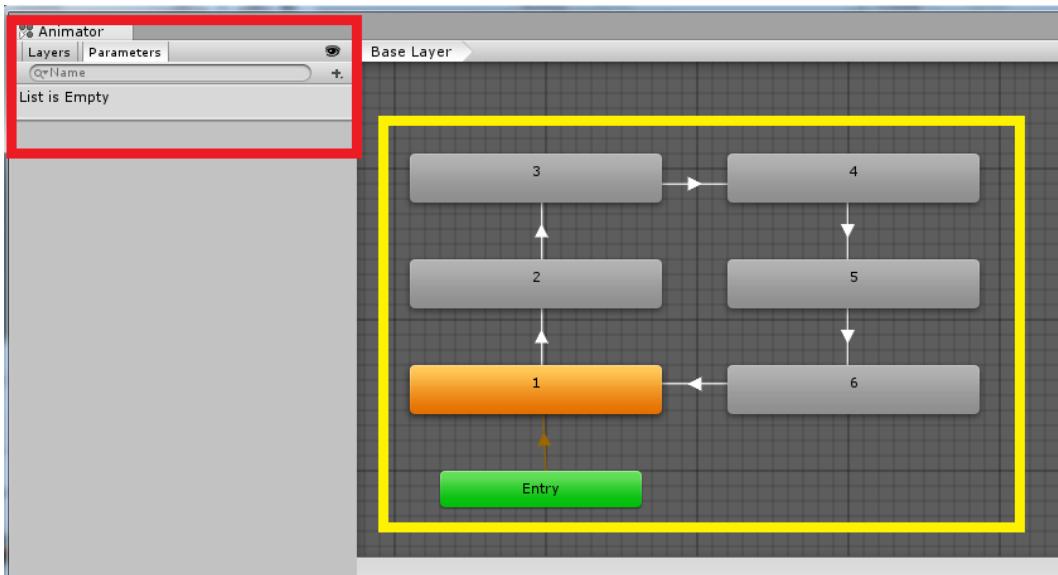


Fig. 9.18 – Máquina de estados del enemigo

Se puede apreciar en el recuadro amarillo de la Fig. 9.18 un bucle infinito entre los estados, por lo que el modelo 3D estará animado todo el tiempo al detectar la imagen. En el recuadro rojo, la lista de parámetros que condicionan la transición de un estado a otro está vacía, por lo que una vez finalizado el clip de animación de un estado, automáticamente se transicionará a otro sin necesidad de cumplirse condición alguna.

El ejemplo mostrado del esqueleto (Fig. 9.17) también se aplica para el resto de los enemigos, como para el jefe final y para el personaje. Lo único que varía es el modelo 3D a usar y el contenido del texto que se muestra, la máquina de estados sigue siendo la misma para todos, pero con sus respectivos *clips* de animaciones pertenientes al modelo en cuestión.

Antes de seguir avanzando con la aplicación para Android, es necesario volver un momento a la aplicación para PC para analizar un poco el rayo scanner que el jugador dispara al hacer *click* derecho en el ratón (Fig. 9.19).

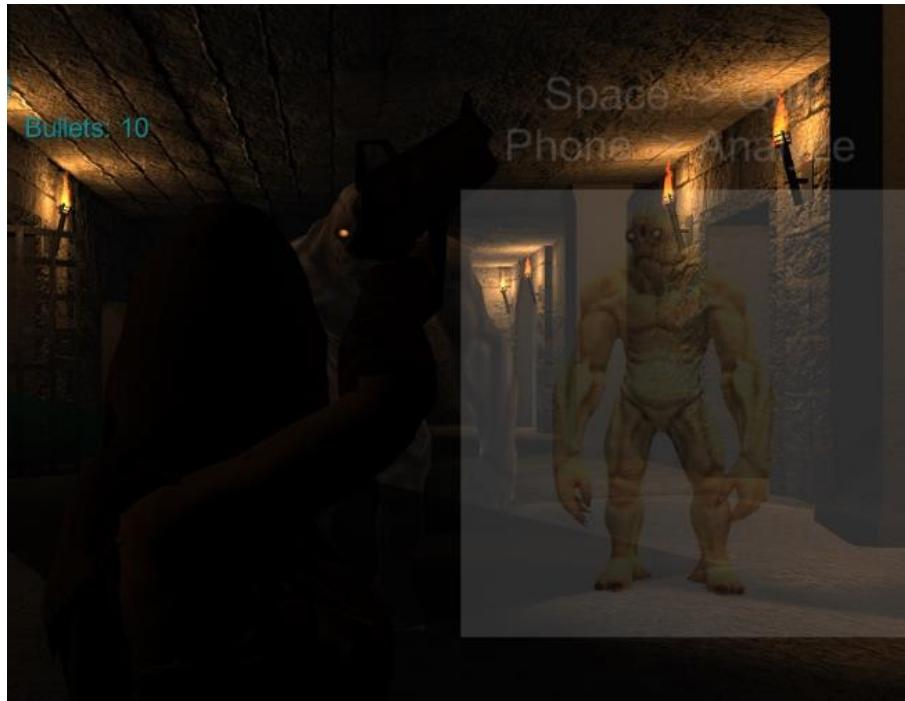


Fig. 9.19 – Instante en el que se utiliza el Scan Ray y este impacta un monstruo. La animación está en curso

El rayo tiene como inicio la posición (x,y,z) actual del personaje, pues, un hijo del mismo tiene los componentes (entre ellos un *script*) encargados de llevar esta tarea a cabo (Fig. 9.20).

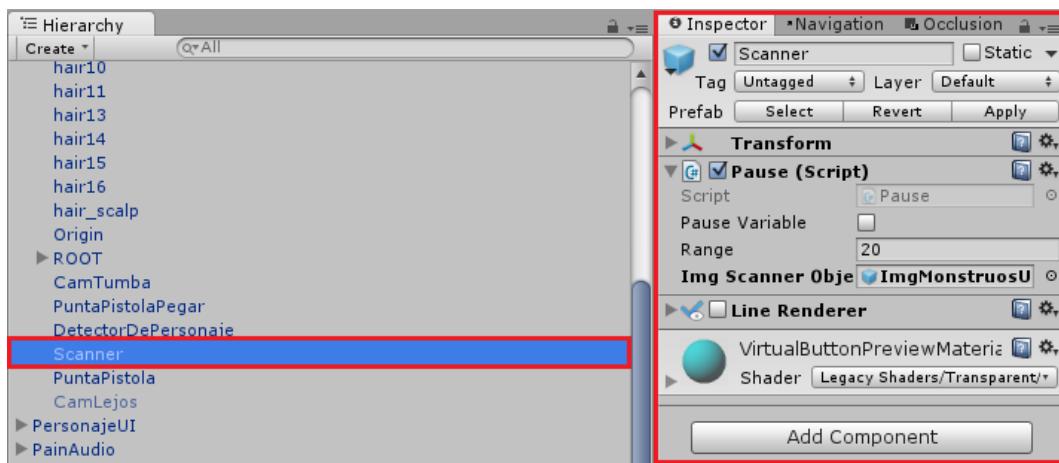


Fig. 9.20 - Objeto Scanner

El *script* adjunto (Pause) que se observa en la Fig. 9.20 tiene las instrucciones en el código para lanzar el rayo, detectar al enemigo y luego mostrar la imagen 2D en pantalla. Se vió anteriormente que Unity utiliza un sistema de etiquetas para distinguir un objeto (o grupo de objetos)

determinados. La pregunta es, si todos los enemigos tienen la etiqueta “Enemigo”, ¿cómo poder diferenciar un zombie de un esqueleto?. La respuesta a la misma son las capas o layers en Unity. Los enemigos tienen todos la misma etiqueta “Enemigo”, pero tienen diferentes capas (Fig. 9.21).

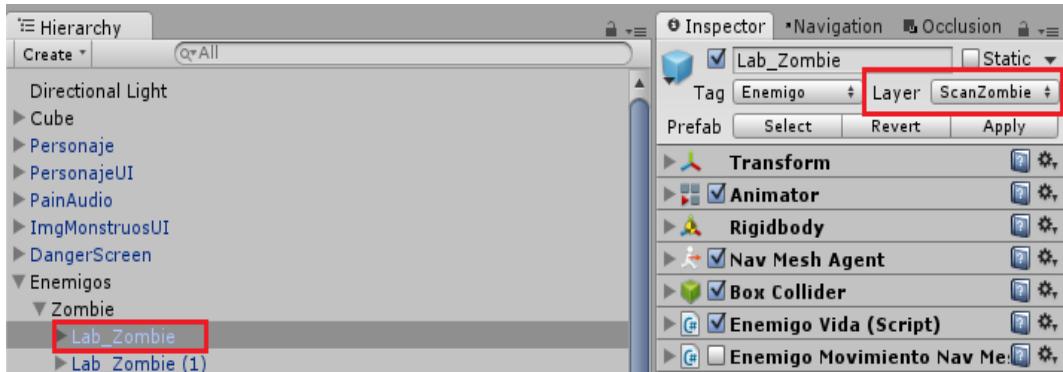


Fig. 9.21 – Objeto Zombie con la capa ScanZombie seleccionada

Así, dentro del script que controla el rayo, se crea una variable para cada capa diferente (Fig. 9.22).

```
// Awake is called when the script instance is being loaded
public void Awake()
{
    // Create a layer mask for the Scan layer.
    maskMonster = LayerMask.GetMask("ScanMonster");
    maskZombie = LayerMask.GetMask("ScanZombie");
    maskSpider = LayerMask.GetMask("ScanSpider");
    maskSkeleton = LayerMask.GetMask("ScanSkeleton");
    maskBoss = LayerMask.GetMask("ScanBoss");

    // Set up the references.
    gunLine = GetComponent<LineRenderer>();

    animadorScanner = imgScannerObject.GetComponent<Animator>();

    impactMonster = false;

    animadorPersonaje = GetComponentInParent<Animator>();
}
//Awake
```

Fig. 9.22 – Función Awake dentro del script Pause

Entonces, cada vez que se dispare el rayo, se evaluará si la información devuelta al impactar con un objeto corresponde a alguna de las capas (Fig. 9.23). En caso afirmativo, la imagen del monstruo correspondiente será mostrada en pantalla.

```

else if (Physics.Raycast(shootRay, out shootHit, range, maskZombie))
{
    // Para el Zombie
    animadorScanner.SetBool("ZombieScan", true);
    gunLine.SetPosition(1, shootHit.point);

    impactMonster = true;
} //if

else if (Physics.Raycast(shootRay, out shootHit, range, maskSpider))
{
    // Para la araña
    animadorScanner.SetBool("SpiderScan", true);
    gunLine.SetPosition(1, shootHit.point);

    impactMonster = true;
} //if

```

Fig. 9.23 – Fragmento del script Pause adjunto al objeto hijo Scanner

Volviendo a la aplicación para dispositivo móvil, la otra funcionalidad que se mencionó anteriormente es para analizar los escenarios en 2 de los niveles 3D jugables, esto es, un mini mapa de referencia en la pantalla del dispositivo.

Uno de los 2 niveles mencionados, tiene un escenario bastante extenso, sumado al hecho de una niebla que dificulta la visión a lo lejos y el jugador puede tardar mucho tiempo (además de aburrirse) hasta poder encontrar el objeto llave para concluir el nivel. Es aquí cuando el complemento de la realidad aumentada es necesario, y, para este caso, no se debe disparar un rayo sino pulsar la tecla “M” (la condición es controlada cuadro a cuadro en la función *Update*). Se puede ver lo mencionado en la Fig. 9.24.

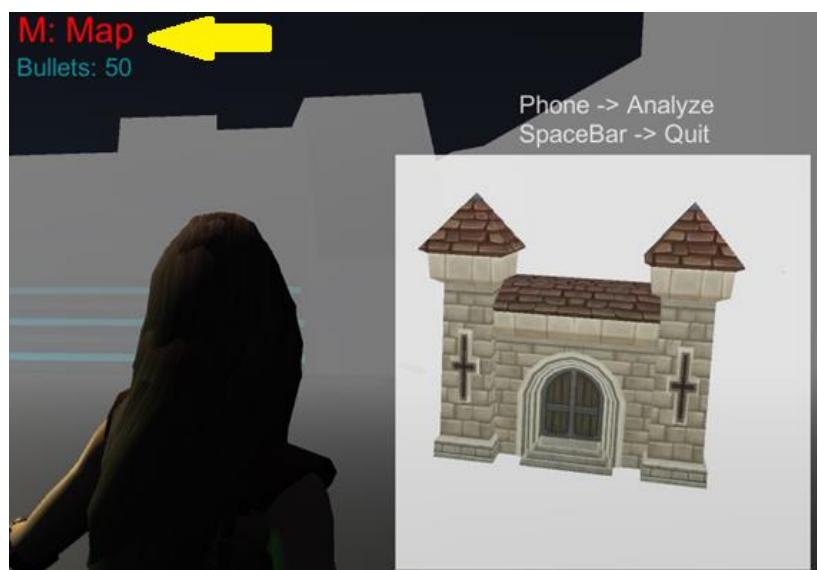


Fig. 9.24 – Imagen 2D luego de pulsar la tecla “M”

Se observa en la Fig. 9.24 que, en ese nivel en particular, un mensaje informativo (M: Map) puede visualizarse desde la *UI*. Luego, de la misma forma que anteriormente se vió con los enemigos, una imagen 2D aparece. Si ahora el lente del dispositivo móvil es enfocado sobre la imagen, el resultado que se obtiene es el modelo 3D del escenario perteneciente al nivel que se está jugando actualmente (Fig. 9.25).

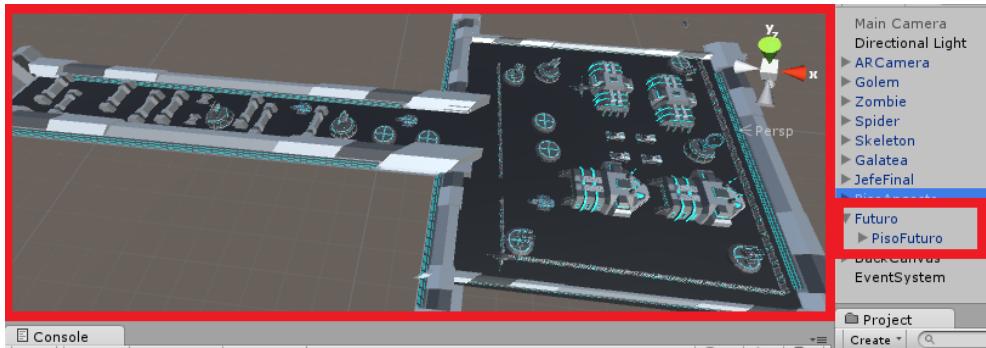


Fig. 9.25 – Modelo 3D del escenario utilizado para un nivel del juego

Los escenarios, a diferencia de los enemigos, no tienen máquinas de estado, pues, no son animados. Tampoco tienen texto informativo, puesto que la finalidad es brindar al jugador un campo de visión completo del mismo.

Como se vió hasta ahora, se utilizó el kit de desarrollo de Vuforia para el reconocimiento y seguimiento de imágenes planas 2D (*Image Targets*) creando objetos virtuales en el mundo real. Vuforia tiene 3 componentes de la plataforma para trabajar con el reconocimiento y seguimiento, el utilizado en este proyecto es el servicio de reconocimiento en la nube, creando una base de datos con las imágenes (*Image Targets*) que se desean utilizar (Fig. 9.26).

Targets (8)					
	Target Name	Type	Rating	Status	Date Modified
	400x400_Tower2	Single Image	★★★★★	Active	Apr 10, 2016 05:09
	400x400_Tower	Single Image	★★★★★	Active	Apr 10, 2016 05:03
	400x400SinCabeza	Single Image	★★★★★*	Active	Apr 10, 2016 04:55
	400x400_Galatea	Single Image	★★★★★	Active	Apr 10, 2016 04:55
	400x400_Skeleton	Single Image	★★★★★	Active	Apr 10, 2016 04:54
	400x400_Spider	Single Image	★★★★★	Active	Apr 10, 2016 04:54
	400x400_Zombie	Single Image	★★★★*	Active	Apr 10, 2016 04:54
	400x400_Golem	Single Image	★★★★*	Active	Apr 10, 2016 04:53

Fig. 9.26 - Los *Image Target* del juego subidos a la base de datos en la nube con sus respectivas clasificaciones

La Fig. 9.26 muestra la base de datos *online* creada para este proyecto, se puede observar 8 *Image Targets* subidos. El recuadro rojo es el botón que permite descargarla y luego importarla a Unity. Este servicio que ofrece Vuforia es bastante útil y sirve también para predecir que tan bien responderá al reconocimiento la imagen subida. El número de estrellas indica si la imagen es apropiada o no para utilizarla (mientras más estrellas, mejor). Cómo se mencionó antes, esta herramienta basada en la web se llama *Vuforia Target Manager* y permite crear y gestionar bases de datos en línea. Para comenzar a trabajar con el *Target Manager*, se necesita una cuenta de desarrollador Vuforia (el registro es gratis). Todas las bases de datos creadas en el *Target Manager* deben estar asociadas a una clave de licencia (una vez activada la cuenta, se pueden crear licencias propias).

En la siguiente imagen (Fig. 9.27), se observa el contenido del kit de desarrollo de Vuforia para Unity cuando se descarga y se importa en el motor.

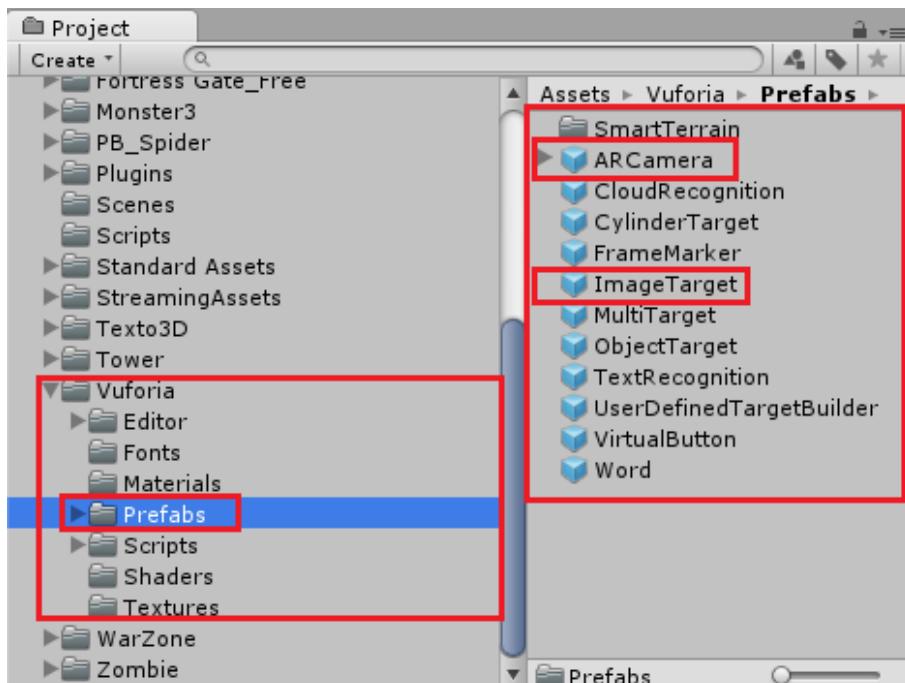


Fig. 9.27 – SDK Vuforia para Unity importado en el proyecto

Dentro de la carpeta *Prefabs* de la Fig. 9.27, se observan 2 Assets recuadrados en rojo. Estos son *ARCamera* e *ImageTarget*, los cuales son los únicos a utilizar, ya que la aplicación se basa en reconocimiento de imágenes planas 2D solamente.

La escena correspondiente a la aplicación para el dispositivo móvil tiene una instancia de *ARCamera* y 8 de *ImageTarget* (una para cada enemigo y nivel). La imagen a continuación (Fig. 9.28) ilustra lo mencionado.

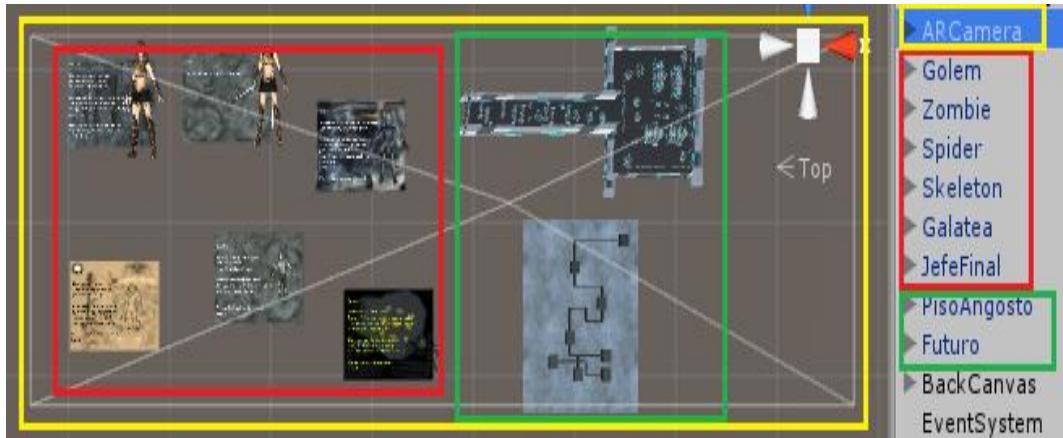


Fig. 9.28 – Escena del proyecto perteneciente al dispositivo móvil

La Fig. 9.28 muestra lo siguiente:

- El recuadro Amarillo encierra el campo de visión de *ARCamera*, esto es, el objeto que tiene el campo de visión en la escena.
- El recuadro rojo son las instancias *Image Target*. A cada uno se le asignaron hijos (un cubo texturizado que será usado como fondo, un modelo 3D y un texto 3D).
- Los objetos del recuadro verde también son *Image Targets*, pero el único hijo que tienen es el modelo 3D del nivel.

Luego, descargada la base de datos del *Vuforia Target Manager* e importada a Unity, se puede asignar a un objeto *ImageTarget* la imagen 2D correspondiente (Fig. 9.29).

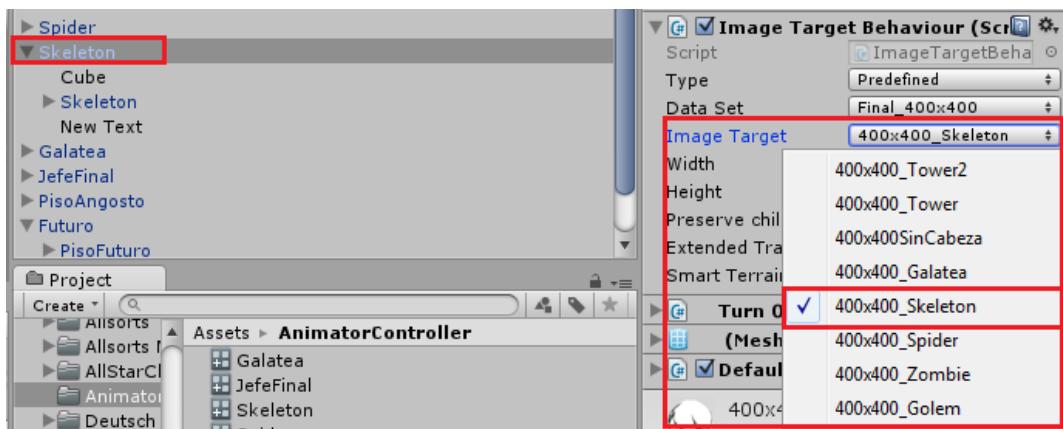


Fig. 9.29 – Objeto ImageTarget perteneciente al esqueleto

Observaciones

A la hora de crear este proyecto, la realidad superó un poco las expectativas. Si bien había una idea en mente, a la hora de poner manos a la obra, hubo un par de limitaciones, todas pertenecientes a las herramientas.

Por el lado de aplicación para Android con RA: cuando se hizo la investigación preliminar, la idea original era usar el lente de la *notebook* o una cámara web, reconocer objetos y luego en base a eso, interactuar con el juego. La página de Vuforia aclara que el kit de desarrollo es para aplicaciones de dispositivos móviles y gafas digitales como muestra la Fig. 10.1.

Vuforia 5.5 SDK

Use the Vuforia SDK to build Android and iOS applications for mobile devices and digital eyewear. Apps can be built with Android Studio (Java/C++), XCode (C++), and Unity, the cross-platform game engine.



[Download for Android](#)

vuforia-sdk-android-5-5-9.zip (6.99 MB)



[Download for iOS](#)

vuforia-sdk-ios-5-5-9.zip (17.10 MB)



[Download for Unity](#)

vuforia-unity-5-5-9.unitypackage (34.50 MB)

[Release Notes](#)

Fig. 10.1 – Captura de pantalla de la página oficial de Vuforia en la sección descargas

Sin embargo, al comenzar con las primeras pruebas, se notó que dentro de Unity se podía hacer uso de la cámara propia de la *notebook* (Fig. 10.2 y Fig. 10.3). Esto claro, mientras se trabaja con el editor sin crear una aplicación.



Fig.10.2 – Grabando un video dentro del editor de Unity en las primeras pruebas de RA. Probando un modelo 3D animado (el enemigo Zombie)



Fig.10.3 – Grabando un video dentro del editor de Unity en las primeras pruebas de RA. Probando un simple cubo

La tapa del libro en las imágenes Fig. 10.2 y Fig. 10.3 fue el primer *Image Target* seleccionado (que además obtuvo 5 estrellas en la clasificación). Luego se intentó hacer correr un ejecutable para escritorio, y apareció el problema en parte ya anunciado (Fig. 10.4).



Fig.10.4 - Ejecutable para escritorio

La Fig. 10.4 muestra que no se pudo inicializar el seguidor (*tracker*) y la base de datos no pude ser cargada. De allí se procedió a combinar el juego con el dispositivo móvil.

Otro dato no menor, es que se invirtió mucho tiempo buscando buen arte para tener un juego vistoso y llamativo, el problema fue que muchos de los modelos eran muy pesados y ralentizaban la producción y la jugabilidad del mismo (Fig. 10.5). Este tipo de software requiere constante testeo para su buen funcionamiento y detección de *bugs*, y crear un ejecutable con esos modelos incluidos tardaba muchísimo tiempo. También, como se mencionó

antes, muchos inconvenientes de compatibilidad aparecieron ya que muchos de los modelos eran de versiones muy antiguas (la mayoría de los modelos gratuitos), y el motor Unity es una herramienta que se actualiza constantemente.



Fig. 10.5 – Ejemplo de un hospital. Este nivel estaba pensado como el nivel inicial

Conclusiones

Se vió qué es un videojuego, sus partes, sus tipos y características. También se vió cómo se diseña un videojuego, y todo lo que hay que tener en cuenta para su creación en la Sección 1.

En la Sección 2 se detalló el desarrollo de la aplicación para Android y cómo se complementa con el Videojuego para PC.

Realizar este tipo de software es muy complejo para una sola persona, pues, como se mencionó anteriormente, es necesario tener un equipo multidisciplinario con roles muy distintos unos de otros si es que se quiere crear un software con propósitos comerciales.

Como posibles mejoras, utilizar otros tipos de *targets* (ejemplo: los códigos QR) para la realidad aumentada, mejorar la física del juego como la inteligencia artificial de los enemigos y optimizar mejor los recursos gráficos del arte del juego.

Expectativa al futuro: poder integrar el área de redes de la carrera, diseñando un juego multijugador por red.

Bibliografía

- Salen, K. & Zimmerman, E. Rules of Play: Game Design Fundamentals. MIT Press. 2003.
- Schuller, D. C# Game Programming For Serious Game Creation. Course Technology. 2011.
- Goldstone, W. Unity Game Development Essentials. Pack Publishing; 2009.
- De Loura, M. Game Programming Gems. Charles River Media. 2000.
- Mullen, T. Prototyping Augmented Reality. Sybex. 2011.
- Abásolo, M.J. et al. Realidad virtual y realidad aumentada en: Interfaces avanzadas. EDUNLP. 2011.
- Acerenza, N. et al. Una Metodología para Desarrollo de Videojuegos, 38º JAIIO - Simposio Argentino de Ingeniería de Software. Págs. 171-176. 2009.
- Odstrcil, M. Apuntes de Clase Ingeniería de Software I y II. 2014.
- Prado, F.G. Diapositivas del seminario Diseño y Creación de videojuegos. 2009.
- Prado, F.G. Diapositivas del curso Creación de videojuegos. 2011.
- Página oficial de Unity: <http://unity3d.com/es/>
- Página oficial de Vuforia: <https://developer.vuforia.com/>
- Página: <http://www.gemserk.com/sum/>
- Grupo en Facebook de Unity: <https://web.facebook.com/groups/unity3d/>
- Página Gamasutra: <http://www.gamasutra.com/>
- Página AiGameDev: <http://aigamedev.com/>
- Hilo del problema de Vuforia con Unity en el foro: <http://answers.unity3d.com/questions/1137259/fatal-error-checkdisallowallocation-allocating-meme.html>. Accedido 28/02/2016.
- Página de Android: <https://www.android.com>. Accedido 14/02/2016.
- OpenGL-ES Tutorial for Android: <http://www.jayway.com/>. Accedido 10/02/2016.
- Tutorial OpenGL: <http://www.learnopengles.com/>. Accedido 18/02/2016.
- Página de metodología RAD: <http://metodologiarad.weebly.com/>. Accedida en febrero de 2016.