

# INTÉRPRETE DE VIDEOJUEGOS DE EADVENTURE PARA UNITY

IVÁN JOSÉ PÉREZ COLADO

MÁSTER EN INVESTIGACIÓN EN INFORMÁTICA. FACULTAD DE INFORMÁTICA  
UNIVERSIDAD COMPLUTESNE DE MADRID



Trabajo Fin Máster en Ingeniería Informática

24 de Mayo de 2016

Director/es y/o colaborador:

D. Baltasar Fernández Manjón

# Autorización de difusión

Iván José Pérez Colado

24 de Mayo de 2016

El/la abajo firmante, matriculado/a en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: “INTÉRPRETE DE VIDEOJUEGOS DE EADVENTURE PARA UNITY”, realizado durante el curso académico 2015-2016 bajo la dirección de D. Baltasar Fernández Manjón en el Departamento de DISIA, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

# Resumen en castellano

Debido a la llegada de nuevas tecnologías, algunas de las herramientas, y productos generados con dichas herramientas, de gran valor, se están quedando obsoletos por la falta de adaptación. Este es el caso de eAdventure, un motor de videojuegos educativos del género aventura gráfica, desarrollado en la Universidad Complutense de Madrid, y programado en Java. En este proyecto se desarrolla una herramienta capaz de interpretar un videojuego generado por eAdventure y ejecutarlo en el motor de videojuegos Unity. Dicho motor, caracterizado por la facilidad de generar aplicaciones multiplataforma, su sencillez, y sus licencias gratuitas, hacen que sea un apropiado nexo de unión entre eAdventure y las nuevas plataformas.

Este intérprete se ha desarrollado de forma colaborativa con el proyecto de Piotr Marzsal "NOMBRE DEL PROYECTO", que reconstruye el editor de eAdventure en Unity, realizando una integración de ambos proyectos, y participando en el desarrollo de algunas partes de su reconstrucción.

## Palabras clave

eAdventure, Unity, Intérprete, Emulador, e-Learning, Videojuegos, Serious Game

# Abstract

English version of the abstract

## Keywords

List of keywords

# Índice general

<b>Índice</b>	<b>I</b>
<b>Agradecimientos</b>	<b>III</b>
<b>Dedicatoria</b>	<b>IV</b>
<b>1. Introducción</b>	<b>1</b>
1.1. Los videojuego educativos y el E-Learning . . . . .	2
1.2. El Objeto de Trabajo . . . . .	3
1.2.1. eAdventure . . . . .	3
1.2.2. Checklist . . . . .	5
1.2.3. El motor de videojuegos Unity . . . . .	7
1.3. Estado del Arte . . . . .	8
1.3.1. Motores de videojuegos . . . . .	8
1.3.2. Motores de desarrollo de Aventuras Gráficas . . . . .	11
1.3.3. eAdventure Android y Mokap . . . . .	15
1.4. Metodología de desarrollo . . . . .	19
<b>2. Comenzando con el Proyecto</b>	<b>22</b>
<b>3. Primera Iteración: Prototipo inicial</b>	<b>26</b>
3.1. Arquitectura del Proyecto . . . . .	26
3.1.1. Controlador Principal de Juego . . . . .	28
3.1.2. Clases de Datos . . . . .	28
3.1.3. Clases de Comportamiento . . . . .	28
3.2. Detalles de la implementación . . . . .	29
3.2.1. Sobre la clase Game . . . . .	30
3.2.2. Sobre las Clases de Datos . . . . .	32
3.2.3. Sobre las Clases de Comportamiento . . . . .	35
3.3. Sobre las Clases de Utilidad . . . . .	39
3.3.1. La clase Texture2DHolder . . . . .	40
3.3.2. La interfaz Sequence: Effect, Conversation y Condition . . . . .	41
3.3.3. La clase Action . . . . .	43
3.4. Acerca de los elementos de Adaptación . . . . .	43
<b>4. Segunda Iteración</b>	<b>45</b>
4.1. Arquitectura del Proyecto . . . . .	45
4.1.1. El nucleo de Ejecución: Runner . . . . .	48

<b>Bibliography</b>	<b>51</b>
<b>A. Title for This Appendix</b>	<b>52</b>
<b>B. Title for This Appendix</b>	<b>53</b>

# Agradecimientos

A mi pc por acompañarme cada día.

# Dedicatoria

Texto...



# Capítulo 1

## Introducción

Cada vez es más frecuente que dispositivos tecnológicos de uso cotidiano, como smartphones o tablets, replacen a los ordenadores personales, y no sólo en los hogares, sino también en los centros educativos, encontrando profesores que realizan clases utilizando este tipo de dispositivos. La plataforma eAdventure, así como los videojuegos generados por esta plataforma, funcionan sobre la maquina virtual Java, la cual cada vez dispone de menos soporte en este tipo de dispositivos. Por lo que, todos aquellos juegos generados en eAdventure, cada vez están disponibles para menos público, y dado que éstos son en su mayoría videojuegos educativos, con contenido de gran valor para la educación, tanto a nivel estudiantil, como de instrucción, concienciación y formación de trabajadores, es importante facilitar que todo el desarrollo realizado sobre eAdventure se pueda explotar y utilizar en cualquier tipo de plataforma.

Este es el caso del videojuego Checklist, creado en eAdventure para formar a personal sanitario, recalcando la importancia de realizar siempre la Lista de Verificación Quirúrgica. Este videojuego fue desarrollado por el equipo de investigación de eUCM y únicamente cuenta con versiones que se pueden ejecutar sobre Windows, Mac OSX y Linux. Checklist es objeto de el trabajo y se reconstruirá sobre Unity, motor de videojuegos muy popular, caracterizado por la amplia portabilidad de los videojuegos creados con dicho motor.

El planteamiento de la reconstrucción de dicho videojuego, lleva a la evolución de este proyecto en algo reutilizable, generando una herramienta capaz de interpretar el contenido

de cualquier juego generado por eAdventure, y permitir jugarlo. Así como la posibilidad de generar dicho juego, en un paquete único ejecutable, y compatible con múltiples plataformas y sistemas operativos.

Gracias a este caso personal y la experiencia portando dicho videojuego, se realizará una explicación acerca de cómo evolucionó este proyecto desde el planteamiento de una reconstrucción de juego, así como sus múltiples iteraciones en el diseño de arquitectura, y su posterior integración junto al proyecto de Piotr Marzsal. En este proceso se exponen todos aquellos problemas que surgen, así como su resolución, enfrentando un cambio de paradigma en el que se pasa de tener una arquitectura orientada a objetos, a una arquitectura orientada a componentes.

## 1.1. Los videojuego educativos y el E-Learning

El e-Learning, también conocido en español con la terminología de Aprendizaje Electrónico, es aquello que, según Martín Hernández, engloba aquellas aplicaciones y servicios que, tomando como base las TIC, se orientan a facilitar el proceso de enseñanza-aprendizaje [1]. Este e-Learning surgió en un primer momento con la intención de facilitar el acceso a la información al mayor número de personas posibles, pero esto fue satisfecho rápidamente por sistemas muy simplificados de e-Learning que básicamente son grandes repositorios de información con unas pocas facilidades.

Este sector, sin embargo, está avanzando para que no haya tanta diferencia con los sistemas de educación tradicionales, intentando reducir la separación que se produce entre un profesor y un alumno cuando se realiza la educación mediante plataformas educativas en línea, añadiendo elementos de monitorización y seguimiento, elementos que fomentan la participación en comunidad como foros de trabajo, o añadiendo elementos de gamificación en los proyectos de aprendizaje que se desarrollan.

No obstante, estos sistemas de educación pese a ser mucho más dinámicos, permitiendo que los usuarios marquen su ritmo de aprendizaje, y presentarse con un enfoque diferente,

que supone un grado de implicación personal en el que es el usuario el que decide realizar un aprendizaje, y no una imposición social, familiar o incluso personal; no dan la motivación suficiente a muchos estudiantes para continuar con la educación y no abandonarla. Es por ello que otro tipo de género de herramientas de e-Learning surgen, donde la motivación para aprender se adquiere a través del uso de la herramienta, y el aprendizaje suele ser una cualidad implícita que se adquiere por la utilización de la misma. Estos son los videojuegos educativos.

El abandono escolar es uno de los problemas más importantes actualmente. Según Prensky, los estudiantes son nativos digitales, personas capaces de interactuar con ricos dispositivos digitales como ordenadores, dispositivos móviles o videoconsolas. Esto difiere con las metodologías tradicionales de educación, en términos de interacción y contenido [2].

Por todo ello, los videojuegos educativos suponen un tema de estudio y desarrollo muy importante dentro de las diferentes tecnologías de e-Learning.

## **1.2. El Objeto de Trabajo**

El objeto de trabajo de este proyecto se ha dividido en tres aspectos. En primer lugar la plataforma de desarrollo de videojuegos educativos eAdventure, la cual comienza a sufrir problemas de portabilidad debido a estar desarrollada en Java. En segundo lugar el videojuego Checklist, desarrollado en eAdventure y que se utiliza para formar a personal sanitario acerca de la utilización de la Lista de Verificación Quirúrgica. Y en tercer lugar, el motor de videojuegos Unity, que en contraposición a eAdventure, es ampliamente conocido por las facilidades que brinda a los desarrolladores para exportar sus proyectos en multitud de plataformas diferentes.

### **1.2.1. eAdventure**

Sobre eAdventure Los videojuegos educativos, a la hora de ser producidos, tienen tres grandes problemas a los que los productores del videojuego deben enfrentarse. Estos tres

problemas son:

En primer lugar, la dificultad de balancear la parte divertida y la parte educativa del videojuego. El correcto balance de ambos aspectos es necesario para el éxito del videojuego, pues, si los estudiantes no se divierten mientras juegan, lo mas probable es que terminen abandonando el juego, lo que echaría por tierra todo el esfuerzo de creación del videojuego; y por otra parte, si todos los esfuerzos son puestos en hacer la parte divertida del juego, es posible que el impacto educativo del mismo sea demasiado pequeño. Diversos autores como Dickey defienden que, los videojuegos enfocados en la historia, como cuentacuentos digitales, o aventuras gráficas point-and-click, son géneros que se adecuan mucho a las necesidades de los videojuegos educativos.

Las aventuras gráficas son un género conocido dentro del mundo de los videojuegos, con grandes títulos como Monkey Island, The Day of The Tentacle o, la franquicia española, Runaway. Tal es el grado de popularidad que tienen estos videojuegos que existen herramientas específicas para la creación de videojuegos de este género, como Adventure Game Studio, DAGE o Open SLUDGE. Sin embargo, todo este tipo de herramientas, pese a ayudar mucho a producir videojuegos de este género, pero no aportan funcionalidades específicas para los videojuegos educativos. De esta manera, eAdventure, aporta un enfoque específico orientado a los videojuegos educativos.

En segundo lugar, el desarrollo de videojuegos es uno de los sectores dentro del desarrollo de software que supone un coste muy grande para su desarrollo, llegando a costar varios millones de dólares. En la última década, han surgido varios proyectos de videojuegos educativos cuyo presupuesto supera los cientos de miles de euros [2]. Sin embargo, con herramientas apropiadas que simplificasen el proceso de desarrollo del juego, y que además fuesen de bajo coste, se podría reducir el presupuesto necesario para desarrollar videojuegos educativos. Es por ello que eAdventure se presenta como una alternativa libre y gratuita para simplificar el desarrollo de videojuegos educativos.

Y finalmente, en tercer lugar, el último problema que tienen los videojuegos, está rela-

cionado con los problemas de distribución de un videojuego una vez que está desarrollado. Es frecuente encontrar que videojuegos están únicamente disponibles en una plataforma o sistema operativo, por lo que gran cantidad de personas no pueden disfrutar de un videojuego, y por consiguiente, los desarrolladores no consiguen que sus videojuegos no tengan el efecto deseado.

Sin embargo, esta herramienta, como se mencionó en la introducción, está desarrollada en Java, y produce Applets standalone, que, pese a no necesitar ninguna instalación, necesitan de la maquina virtual Java para funcionar. La máquina virtual Java ha gozado de mucha popularidad a lo largo de los años, pero actualmente cada vez cuesta más encontrar dispositivos de uso cotidiano que dispongan de ella. Tanto es así que, sistemas operativos como Android, cuyo lenguaje de programación principal es Java, han construido su propia máquina virtual llamada Dalvik [3], la cual está siendo sustituida actualmente por ART [4], y que reemplaza completamente a la maquina virtual Java para la ejecución de programas desarrollados en dicho lenguaje.

Es por ello que el desarrollo analizado en este artículo se encarga de mejorar la capacidad de eAdventure para la distribución de juegos, desarrollando un intérprete capaz de visualizar un juego creado en eAdventure, en el motor de videojuegos Unity.

### **1.2.2. Checklist**

Ante la pregunta ¿Que instrumento reduce el ratio de muertes y complicaciones en cirugía en más de un tercio? Pese a que pueda parecer que son los avances tecnológicos, la calidad de la medicina y de los utensilios, o la higiene en los hospitales, esta no es la respuesta. Estos factores han mejorado la medicina, y han logrado que avance hasta el punto de realizar operaciones que antes considerábamos impensables, ayudando a los médicos a detectar problemas mucho antes de que surjan y a monitorizar situaciones peligrosas. Pero es algo mucho mas sencillo lo que ayuda a evitar muertes y complicaciones, y esto es una simple lista que se realiza antes, durante y después de una operación. Esta lista tiene el

nombre de Lista de Verificación Quirúrgica [5].

La Lista de Verificación Quirúrgica fue desarrollada por la Organización Mundial de la Salud en 2007 buscando identificar las normas mínimas de aplicación universal. Se construyó mediante el estudio de pacientes, cirujanos, anestesiólogos, enfermeras y expertos en seguridad de los pacientes a lo largo de dos años. En si, la lista está formada por simples comprobaciones que llevan muy poco tiempo de realizar, y no debe ser una carga para el equipo quirúrgico, pues ellos ya tienen una labor muy importante que llevar a cabo.

El videojuego Checklist surge para concienciar y educar acerca de la correcta aplicación de la Lista de Verificación Quirúrgica, mostrando, tanto los beneficios de aplicarla correctamente, como las consecuencias de una mala aplicación. Se desarrolló en la UCM, por el grupo de investigación eUCM junto con la Facultad de Medicina de la UCM, el Hospital Doce de Octubre y el Massachusetts General Hospital [6].

Este juego está desarrollado utilizando eAdventure, y por consiguiente es del género Aventura Gráfica, más concretamente, una en Primera Persona, en la que el jugador no maneja a un personaje dentro de un juego, sino que ocupa el lugar del protagonista, quien toma uno de los diferentes roles que se ofrecen y que participan dentro de un quirófano, y realiza participación en una operación en tres situaciones diferentes: Antes de inducir la anestesia al paciente, antes de la primera incisión en la piel, y antes de que el paciente se marche de la sala de operación. Asimismo, dentro del videojuego suceden diferentes eventos que se dan de forma aleatoria, como una incisión mal marcada, o un compañero que no coopera.

El paquete ejecutable está disponible en Windows, Mac OS X y Linux, en versiones de 32 y 64 bits. Adicionalmente, el Applet Standalone se puede encontrar en el repositorio en Sourceforge de eAdventure, y permite la ejecución mediante el uso de la máquina virtual Java. Sin embargo, la ejecución de dicho juego en Tablets Android e iPad no está disponible, por ello se realiza la adaptación de dicho juego a Unity

### 1.2.3. El motor de videojuegos Unity

Los motores de videojuegos son herramientas que permiten y facilitan la creación y desarrollo de un videojuego. Estos, dan la funcionalidad básica de proveer de un entorno gráfico para la representación del videojuego, así como un motor físico con detección de colisiones, junto con un bucle principal que se encarga de procesar y ceder tiempo de procesamiento para cada uno de los elementos del videojuego.

Unity es un motor de videojuegos caracterizado por ser capaz de producir videojuegos para multitud de plataformas, entre las cuales encontramos Windows, OS X y Linux, junto con los mas frecuentes sistemas operativos móviles y de tabletas, como Android e iOS, tanto iPad como iPhone, además de videoconsolas como PlayStation 3, PlayStation Vita, Wii, Wii U, etc. Esta facilidad para producir videojuegos en múltiples plataformas, así como su licencia gratuita para los desarrolladores independientes, y pequeñas startups que no tienen presupuesto para pagar las costosas licencias de los motores de videojuegos, hacen de este motor una opción muy elegida por gran número de desarrolladores que quieren llegar al mayor número de personas con el menor esfuerzo posible.

Asimismo, otras de las características de este motor incluyen la, ampliamente utilizada en videojuegos, arquitectura basada en componentes, en la que los objetos se componen de distinta manera dependiendo de sus necesidades en cada momento, permitiéndoles mutar o adquirir un comportamiento adicional si se requiere; en contraposición a la arquitectura orientada a objetos, en la que los objetos adquieren una funcionalidad al implementar una interfaz o heredar de una clase de una jerarquía superior, así como adquirir funcionalidad mediante la implementación de si misma. Esta arquitectura permite tener objetos más dinámicos, así como facilitar la reutilización de código gracias a la incorporación del mismo componente en varios objetos, sin embargo supone un cambio de paradigma en la programación, y requiere conocer tanto el funcionamiento de Unity, la comunicación entre sus componentes, así como el complejo ciclo de vida que compone cada Tick de juego [7].

En Unity, cada objeto que toma partido dentro de la escena, está constituido por un Ga-

meObject y todas aquellas componentes asociadas, extendiendo a la clase MonoBehaviour, así como una componente Transform que le permite orientarse y deformarse en la escena, y una

## **1.3. Estado del Arte**

Una vez hemos definido correctamente el objeto de trabajo de este proyecto, compuesto por todos aquellos elementos que lo forman, se analizarán otras alternativas a los objetos presentados en el anterior apartado.

Frecuentemente ocurre que no existe una única forma de hacer las cosas, y por ello, existen multitud de herramientas disponibles para el desarrollo de Aventuras Gráficas Point and Click, así como de generación de aplicaciones multiplataforma, motores de videojuegos, y herramientas de learning analytics. Todas estas herramientas, además de ayudar a definir mejor los objetivos del trabajo, buscando generar una herramienta que innove y añada funcionalidad adicional a las existentes, ayudan en el desarrollo del proyecto a contextualizar mejor la aplicación, y a inspirar algunos de los elementos de esta misma aplicación.

### **1.3.1. Motores de videojuegos**

Los motores de videojuegos son aquellas herramientas, o conjunto de herramientas que permiten el diseño, creación, y visualización de un videojuego. La funcionalidad básica de un motor de videojuegos es la de proveer de un entorno gráfico, ya sea bidimensional o tridimensional, que simplifique el proceso de representación de un videojuego, permitiendo en muchos casos interactuar directamente con el entorno gráfico, y pudiendo posicionar los elementos que conforman el videojuego con mayor facilidad. Para completar el ecosistema de generación de videojuegos, estos motores proveen al desarrollador de un motor físico, con detección de colisiones entre los elementos, así como soporte a archivos multimedia como música, vídeos, imágenes, e incluso modelos tridimensionales, y finalmente, permiten al desarrollador programar los comportamientos que los elementos deben tener.



No obstante, estos motores tendrían poca utilidad si únicamente permitieran visualizar los juegos en su interior, por lo que dichos motores tienen un compilador que genera un ejecutable de dicho videojuego. Existen motores especializados en plataformas concretas, como ordenador, y sus diferentes sistemas operativos, como Windows, Mac OSX, o Linux; aunque también existen motores especializados en videoconsolas como PlayStation, XBOX o consolas portátiles de Nintendo. Finalmente, existen otros motores que están caracterizados por su capacidad para generar videojuegos para un amplio catálogo de plataformas, reduciendo mucho los costes de producción y distribución de un videojuego.

Finalmente, uno de los puntos que está cambiando la industria del videojuego a escala mundial, es el acceso que las personas pueden tener a esos motores de videojuegos, y las licencias con las que estos son distribuidos. Desde los primeros motores de videojuegos, que en un comienzo tomaron el nombre de los videojuegos que desarrollaban, como DOOM o Quake, estos eran desarrollados por empresas privadas, y para el beneficio de las mismas, es decir, una empresa desarrollaba un motor, y mantenía su ciclo de vida y modelo de negocio orientado a rentabilizar el esfuerzo y dinero invertidos en dicho motor. Poco a poco, empresas fueron centrando su modelo de negocio en desarrollar un muy potente motor, y vender licencias de dicho motor a los desarrolladores para que ellos explotaran el potencial del motor. Dentro de estos motores se hayan Unreal Engine, Unity o CryEngine. El modelo de negocio de estos motores fue evolucionando, y algunos de ellos ahora disponen de licencias gratuitas para desarrolladores. Y finalmente, existen motores de desarrollo libres, con su código completamente disponible para editarlo si se necesita, y con licencias libres y gratuitas ayudan a los desarrolladores independientes.

Estos motores, aun así, siguen teniendo un nivel de abstracción bastante grande, y han surgido motores que abstraen comportamientos y entidades que componen algunos videojuegos concretos,

## Unreal Engine

Uno de los motores que fueron planteados cuando surgió este proyecto es Unreal Engine, esto es debido a que, al igual que Unity3D, ofrece licencias gratuitas para desarrolladores independientes que no obtienen beneficio, o obtienen poco beneficio de sus juegos. Este motor es conocido por la gran calidad gráfica que se consigue produciendo juegos con el.

Unreal Engine fue desarrollado por Epic Games, inicialmente en 1998 para el desarrollo de videojuego First Person Shooter Unreal. El motor ha ido evolucionando desarrollando videojuegos como Unreal Tournament, Turok, America's Army -videojuego desarrollado por un equipo de desarrolladores de la Marina estadounidense con el objetivo de formar a civiles en armamento y estrategias de combate, y conseguir reclutarlos posteriormente en el ejército de los Estados Unidos-, Gears of War, la saga Bioshock, o la saga Mass Effect.

Unreal Engine está caracterizado por tener una gran capacidad de renderización y calidad gráfica, así como un complejo sistema de materiales, iluminación y sombras, que consiguen, si se aplican correctamente, un resultado muy similar a una imagen real, como en la demo técnica London Apartment, visible en la figura 1.1. Esto, no obstante, no implica que todo lo generado con Unreal Engine tenga una calidad gráfica excelente; únicamente hace más sencillo este proceso.



**Figura 1.1:** *London Apartment, una demo técnica desarrollada con Unreal Engine 4.*

Uno de los factores que influyen en la decisión de elegir qué motor es el mas apropiado para realizar la reconstrucción de eAdventure dentro de el, es la capacidad de producir videojuegos para el mayor número de plataformas posibles. Unreal Engine 4 posee esta capacidad, pues es capaz de generar videojuegos para Windows, XBOX, LINUX, OSX, PlayStation 4, iOS, Android, Ouya, iOS, y JavaScript/WebGL, entre otros.

Sin embargo, en nuestro equipo existen dos factores por los que Unreal Engine 4 fué descartado. En primer lugar encontramos que modificar Unreal Engine 4, pese a que se han incorporado nuevos sistemas de gestión de Plugins en esta última versión, es una tarea extremadamente laboriosa, sobre la que hay disponible escasa documentación, y que todavía se encuentra en estado de desarrollo. Por otra parte, ambos integrantes del equipo tenemos gran experiencia en el uso de Unity, por lo que el proceso de aprendizaje hubiera sido bastante laborioso.

### **1.3.2. Motores de desarrollo de Aventuras Gráficas**

Al igual que los motores de videojuegos añaden una capa más de abstracción para los desarrolladores a la hora de crear videojuegos, existen herramientas que se especializan en géneros de videojuegos concretos para añadir funcionalidades específicas que permiten explotar las posibilidades de dichos géneros, y automatizan otras tareas que en otro tipo de videojuegos necesitarían ser controladas o desarrolladas independientemente. Algunos de estos editores son, RPG Maker para videojuegos del género Role Playing Game, RPG, GameGuru para videojuegos del género First Person Shooter, FPS, o Quintus para videojuegos de Plataformas.

Entre estos géneros que gozan de editores que facilitan el proceso de creación de juegos para su propio género, encontramos las Aventuras Gráficas Point and Click. Existen multitud de editores que facilitan la generación de este tipo de juegos. Algunos mas conocidos que otros, con diferentes características y funcionalidades propias. Otros que, desgraciadamente,

aunque son conocidos por generar aventuras gráficas de renombre, no son públicos, pero de los cuales podemos, mediante el análisis de dichos juegos, extraer funcionalidades, y casos reales de características que tuvieron éxito.

Entre las características que abstraen los motores de desarrollo de aventuras gráficas, encontramos que, el usuario no debe preocuparse de tratar con la cámara, una de las tareas que, si se hace indebidamente, llevará al fracaso al juego, además de ser una de las tareas más complejas de realizar. La mayoría de motores se encargan de facilitar un entorno basado en escenas, personajes y objetos, pudiendo establecer zonas por la que el jugador se moverá, ya sea mediante trayectorias o mediante áreas que se encargan de cambiar la escala del personaje y determinar las zonas por donde el personaje se podrá mover.

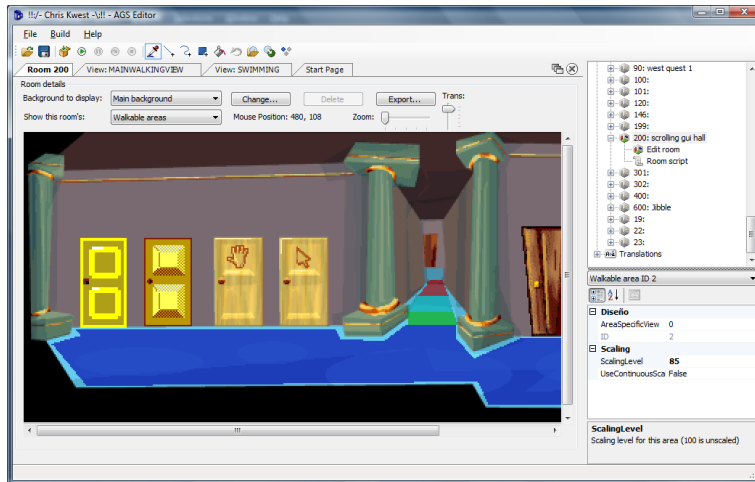
## **Adventure Game Studio**

Dentro de esto de editores, encontramos a el más conocido de todos ellos, Adventure Game Studio, también conocido por sus siglas AGS. Está caracterizado por la increíble personalización que tienen los juegos que se generan con esta herramienta, permitiendo modificar prácticamente todos los elementos que lo conforman, además de dar la oportunidad a los desarrolladores, de generar su propios Scripts e incluirlos en el juego.

Este software no sólo es un gran editor que da casi infinitas posibilidades a los desarrolladores, sino que además está acompañado de una gran comunidad de usuarios, desarrolladores independientes y aficionados, que comparten entre ellos sus proyectos. Todos estos proyectos compartidos, son debidamente catalogados en el repositorio de juegos que provee la propia página web de esta herramienta. Una de las pegas que tiene este editor es que únicamente funciona en Windows, aunque existen alternativas muy similares para linux.

Finalmente mencionar que este motor, pese a ser gratuito para proyectos que no son comerciales, tiene una licencia que, en caso de ser comercializado, establece que el usuario debe encargarse de pagar por todas las librerías que AGS utiliza para funcionar a los desarrolladores de dichas librerías.

En la figura 1.2 se puede observar la vista principal del editor, con una escena en la



**Figura 1.2:** *Una escena en Adventure Game Studio*

parte del centro-izquierda. En la parte superior derecha se ve la lista de escenas disponibles, y cuando estas se abren, se añaden en forma de pestañas al editor.

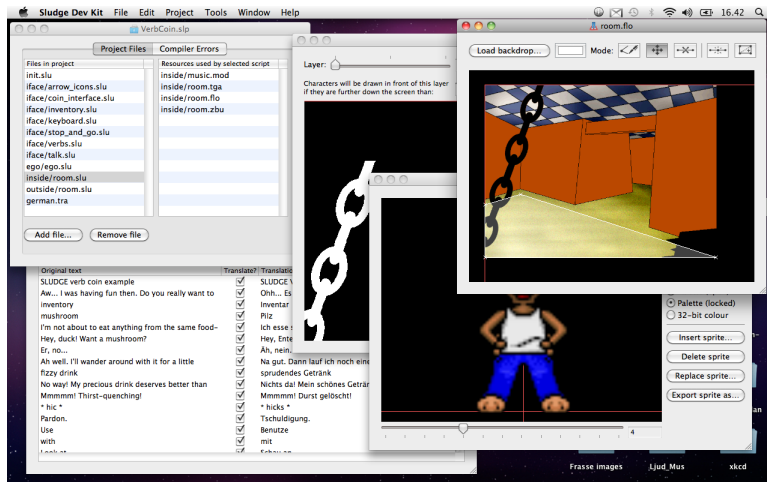
## Open SLUDGE

Otro de los editores de aventuras gráficas más conocidos es Open SLUDGE. Este editor tiene una enorme lista de características disponibles, entre las cuales encontramos no sólo las básicas como personajes, objetos, escenas, o elementos interactivables; sino que además, encontramos características como poder realizar paralajes en fondos de escenas, que además pueden moverse a lo largo de la escena, posibilidad de añadir música y efectos de sonido, uso de timers, y muchas otras características.

No obstante, este editor tiene algunos problemas, como que, por ejemplo, la cámara no sigue al personaje, o no tiene un buen editor de conversaciones, pese a que estas, si que tienen soporte para ser traducidas y localizadas fácilmente. Otro problema de este editor está en que la comunidad de usuarios que lo conforman, y el equipo de desarrollo del editor es bastante escueta. Si bien es cierto que aunque esta comunidad sea pequeña, los usuarios que la conforman ceden recursos de muchos tipos para añadir funcionalidad adicional o mejorar la calidad gráfica del editor.

Como detalle adicional de este editor, y característica de las más importantes, es que, es

multiplataforma, y no solo se puede ejecutar en Windows, OSX y Linux, sino que además, le da al usuario la posibilidad de generar su juego para estas plataformas, en un paquete individual, con todos los recursos incluidos dentro del mismo.



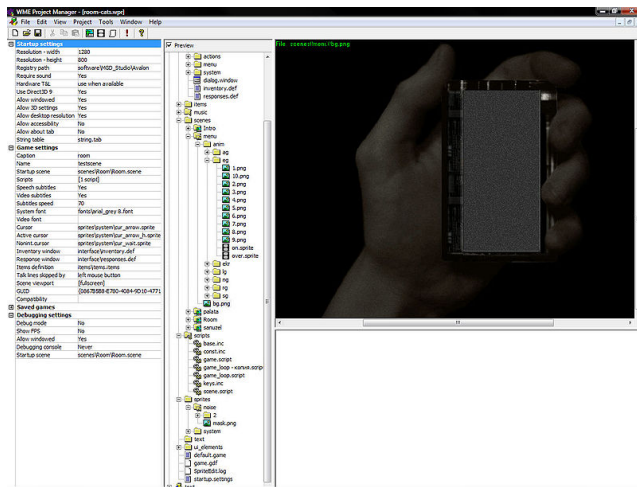
**Figura 1.3:** *Multitud de ventanas de Open SLUDGE ejecutándose en OSX*

En la figura 1.3 se observa como Open SLUDGE se ejecuta en OSX. Las ventanas que se muestran son, por ejemplo, la superior derecha, una previsualización de una escena, con los elementos que hay en ella, y el área dentro de la cual el personaje puede moverse. También, se observa, en el fondo, un diálogo que está siendo traducido del Inglés al Alemán.

## WinterMute Engine

Por último encontramos WinterMute Engine. Este pack de herramientas, cuya primera versión data de enero de 2003, para crear aventuras Point and Click, está caracterizado por ser capaz de crear aventuras gráficas 2D, 2.5D (Personajes en 2 dimensiones, y entornos en 3D, y viceversa), y aventuras generadas completamente en 3D. Al igual que SLUDGE, tiene características estándar como personajes y paralaje en escenas, aunque, frente a este, añade características como editores de interfaz, o soporte a accesibilidad, como poder hacer Text-To-Speech. Sin embargo, una de las características más llamativas de este editor, es la posibilidad de poder generar aventuras gráficas y que estas se ejecuten en iOS, pudiendo

instalarlas en un iPad o iPhone. Finalmente decir que el proyecto se distribuye bajo licencia MIT.



**Figura 1.4:** *Gestor de proyectos de Wintermute Engine*

en la figura 1.4 se muestra una vista del editor WinterMute Engine. La columna central muestra un explorador de archivos que conforman el proyecto, con imágenes y sprites, junto a scripts o otro tipo de archivos. La columna de la izquierda permite configurar el proyecto, y editar características del juego como la resolución del mismo. Y finalmente, la columna de la derecha es una previsualización del archivo seleccionado.

### 1.3.3. eAdventure Android y Mokap

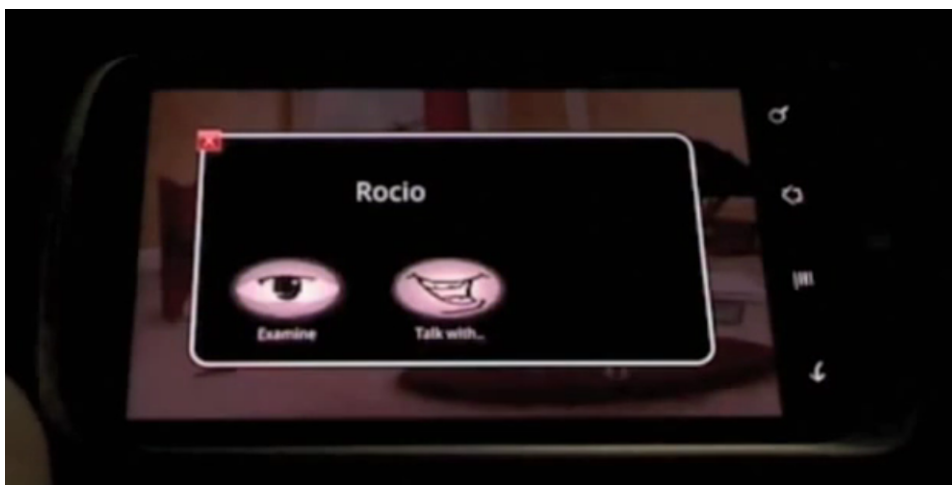
Creado por Javier Torrente e Iván Martínez-Ortiz, y con última versión datada el 30 de enero de 2014, encontramos el proyecto eAdventure Android, el cual tiene dos propósitos. El primero es el de permitir al usuario ejecutar juegos de eAdventure en Android, y el segundo es el de explotar los sensores del dispositivo, como el GPS para la ubicación, y la cámara para lectura de códigos QR. Este proyecto tuvo problemas de continuidad ya que uno de los desarrolladores dejó el equipo y, debido a la dificultad para continuar el proyecto sin su aportación, se decidió que el proyecto evolucionase en otro proyecto llamado Mokap.

El proyecto eAdventure Android añadía características que facilitaban el uso de juegos



**Figura 1.5:** *El videojuego Fire Protocol mostrando la habilidad de hacer Zoom con una lupa e identificar elementos.*

de eAdventure en entornos táctiles, como la capacidad de identificar elementos mediante el uso de una lupa que, si sobrepasaba algún elemento interactuable en la escena, muestra al usuario información acerca de dicho elemento. Esta característica se puede encontrar en la figura 1.5. Además de esta característica, eAdventure android tiene una forma diferente de mostrar el menú, ya que este puede ser difícil de tocar con los dedos en una pantalla pequeña. esto se puede encontrar en la figur a.



**Figura 1.6:** *El videojuego Chocolate Factory mostrando el menú contextual.*

Esta funcionalidad de emulador de eAdventure en Android, es la que se consigue ob-



tener mediante la creación del emulador standalone de uAdventure, capaz de importar y ejecutar cualquier juego; pudiendo ser generado dicho emulador para cualquier plataforma, permitiendo ejecutar los juegos sin problemas de compatibilidad ni necesidad de instalar la Máquina Virtual Java. Dado que eAdventure Android ya no tiene actualizaciones, ni soporte, uAdventure tomará el control de estas necesidades de ahora en adelante.

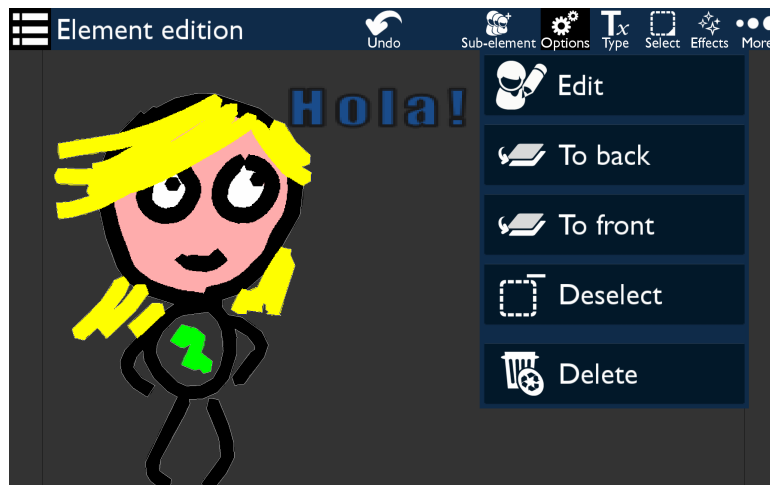
Por otra parte, no existe un editor que permita generar juegos que utilicen la localización mediante GPS, ni la lectura de códigos QR, por lo que, en uAdventure, como trabajo futuro, está pendiente añadir dichas características, y un editor que facilite la creación de videojuegos que los utilicen.

Con la llegada del TFG de Antonio Calvo Morata y Dan Cristian Rotaru, Mokap, eAdventure Android se abandonó para centrar el esfuerzo en el desarrollo de esta herramienta de autoría cuyo objetivo es el de generar videojuegos que puedan ser extremadamente simples, y sin la necesidad de herramientas externas. Surge como una herramienta de prototipado que permita al usuario transformar una idea a algo real de la manera más rápida posible.

En un comienzo, el proyecto se llamó eAdventure Editor y se ejecutaría en Android, este tendría una funcionalidad muy básica, pudiendo generar juegos con escenas, añadir elementos mediante una herramienta para pintar, escritura de texto, o incluyendo imágenes de un repositorio externo. Además, permitiría añadir efectos simples al interactuar con los elementos, como eliminarlos, o realizar transiciones de escenas, así como la posibilidad de añadir condiciones a la hora de ejecutar esos efectos.

Esta primera versión se puede observar en la figura 1.7. El personaje se ha dibujado mediante la herramienta de pincel, que permite modificar su grosor y color. Este personaje, una vez se termina de pintar, se transforma en una figura geométrica rodeando las formas que lo componen con polígonos. Esto facilita la detección de colisiones, para saber si el usuario realiza una interacción con dicho elemento. En la imagen se puede ver un texto generado con la herramienta de generación de textos, el cual pone "hola.<sup>en</sup> color azul con borde negro. Finalmente, en la imagen se observa el menú que permite posicionar los elementos

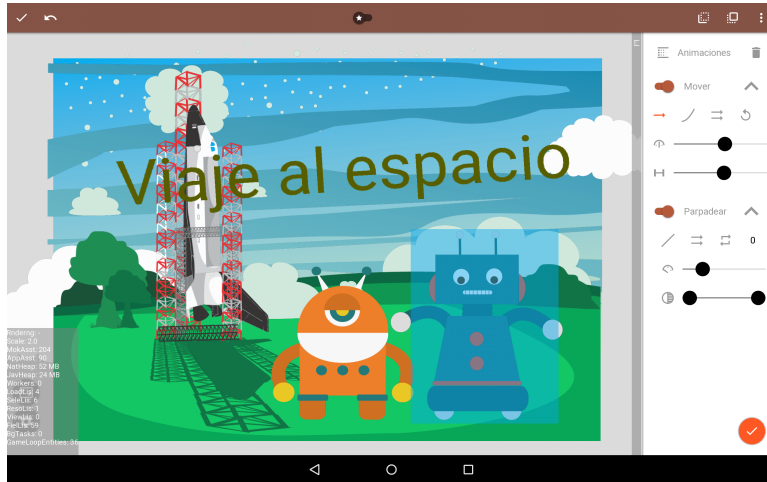
en diferente orden de profundidad en la escena.



**Figura 1.7:** *Primera versión de Mokap, llamada eAdventure Editor en Android.*

Sin embargo, de la primera versión a la última ocurrieron muchos cambios significativos, un completo lavado de cara a la aplicación entera, así como un cambio de nombre, pasando a llamarse Mokap, con un nuevo logotipo. En el desarrollo de esta nueva versión estuvo implicado todo el equipo de e-UCM, y sin hablar de cambios en la infraestructura y el código, la lista de funcionalidades añadidas es muy extensa. Entre estas funcionalidades encontramos capacidades como poder mover elementos en la escena y hacer que parpadeen, ya sea periódicamente, una vez, o un número determinado de veces. El repositorio fue mejorado para permitir incorporar elementos compuestos y animados, así como música y sonidos. Otra de las características que se incorporaron en Mokap es la habilidad de incorporar Shaders a la escena, para conseguir resultados mucho más vistosos.

En la figura 1.8 se observa una escena mucho más elaborada. El fondo de la escena, compuesto por el cielo, los árboles, y el cohete -el cual se mueve- son una forma compuesta descargable desde el repositorio de recursos de Mokap. Los robots que se hayan en el frente son elementos animados, el robot de la izquierda ha sido animado mediante frames, es decir, mediante una secuencia de imágenes, sin embargo, el robot de la derecha utiliza Spine, animación basada en esqueleto. Las animaciones generadas con Spine son mucho más ligeras,



**Figura 1.8:** Última versión de MokuP, mostrando una escena compleja compuesta por múltiples elementos compuestos.

ocupando unos pocos kilobytes, frente al elevado peso que tienen las animaciones generadas con frames, el cual puede llegar a las decenas de megabytes, dependiendo de la resolución y calidad de la imagen. El menú de la derecha muestra los submenús que permiten editar el movimiento y el parpadeo de dicho objeto.

Observando la evolución que han seguido ambos proyectos, extraemos la conclusión de que, la necesidad de desarrollar un intérprete para Android, e incorporarle las características que eAdventure Android introducía a eAdventure, es una necesidad real, ya que, aunque existe un proyecto que satisfacía esta carencia, la realidad es que dicho proyecto no es utilizable, y las posibilidades de retomarlo son mucho menores dado su elevado nivel de complejidad. Por otra parte su evolución, MokuP, tiene un objetivo muy diferente al de uAdventure, y son dos proyectos totalmente diferentes, aunque ambos son herramientas de autoría capaces de generar y ejecutar juegos en Android.

## 1.4. Metodología de desarrollo

Una parte crucial de todo proyecto de ingeniería del software es la metodología de desarrollo que se ha utilizado para su construcción. La metodología determina el proceso de

generación del software, pudiendo ser una metodología más lineal como el Proceso Unificado de Desarrollo de Software, o un desarrollo en Cascada, o metodologías basadas en la iteración, como el modelo en espiral de Boehm, o un desarrollo incremental. Sin embargo, la mayor parte de estas metodologías están pensadas para coordinar a grandes equipos en procesos de desarrollo de larga duración. Por consiguiente, existen metodologías de desarrollo más apropiadas para pequeños equipos que, debido a su configuración de personal y a la rápida comunicación entre miembros, entre otros factores, consiguen que el desarrollo de dicho software sea mucho más ágil.

Asimismo, este proyecto, como muchos de los Trabajos de Fin de Máster, consta de una amplia parte dedicada a la investigación, aprendizaje de nuevas tecnologías, o del desarrollo de prototipos para verificar si esa es la manera más apropiada de hacerlo.

Es por ello que, para desarrollar este proyecto se ha utilizado una metodología de desarrollo ágil, basada en prototipos, en la que se realizan múltiples iteraciones, sobre cada uno de los componentes que conforman el sistema, así como sobre el sistema en sí mismo, realizando tres grandes iteraciones sobre el mismo:

- Una primera iteración para estudiar y analizar los videojuegos de eAdventure, así como para interpretar dichos paquetes e intentar representarlos, aprendiendo de los resultados obtenidos mediante el primer prototipo.
- Una segunda iteración generando un mejor diseño de la aplicación, nuevas interfaces, refactorización de clases y código replicado, así como mejoras gráficas visuales dentro de la aplicación.
- Una tercera iteración, que consiste en poner en funcionamiento el proyecto, separando el proyecto en dos ramas. La primera, en la que se genera un emulador capaz de importar juegos de eAdventure, y permitir al usuario jugarlos, y una segunda que integrando el proyecto de Piotr Marszal junto al intérprete de juegos, transformando ambos proyectos en una única herramienta capaz de generar juegos dentro de Unity. En

esta iteración no sólo se generan estos dos proyectos, sino que, como parte del proceso de integración, se adaptan algunas de las clases, eliminando clases que cumplen con la misma función, y reconstruyendo algunos de los elementos para que sean capaces de trabajar en múltiples condiciones.

Estas tres iteraciones hacen al proyecto evolucionar desde el primer experimento de "hacer funcionar un juego de eAdventure en Unity.<sup>a</sup> "generar un framework completo de creación, interpretación, y ejecución de aventuras point and click basado en eAdventure".

Para dar soporte a la metodología de desarrollo, con mayor frecuencia en la última iteración del desarrollo, se realizaban reuniones periódicas de revisión, generación de tareas para la próxima reunión. Estas reuniones tenían lugar cada 1 o 2 semanas, y estaban formadas por todos los integrantes del desarrollo de ambos proyectos, Baltasar Fernandez Manjón, Iván Martínez Ortiz, Piotr Marszal e Iván José Pérez Colado. Gracias a estas reuniones, la comunicación entre los integrantes ha sido mucho más fluida.

## Capítulo 2

# Comenzando con el Proyecto

Tras obtener el videojuego Checklist en su versión ejecutable, el cual se puede descargar a través del repositorio de eAdventure en Sourceforge, se realizó un estudio. Se completó el videojuego en repetidas ocasiones intentando buscar la aleatoriedad que aparecía descrita en los documentos del proyecto, y analizar a fondo el funcionamiento del mismo.

En una primera instancia parece un videojuego sencillo, con mecánicas muy simples, apenas reglas para el desarrollo, más allá de las que se nos imponen circunstancialmente en alguna de las escenas en las que no podremos avanzar sin tener la atención del equipo. Sin embargo, el juego esconde una mayor complejidad, pues tus decisiones dentro del juego te llevarán a alcanzar un final diferente cada vez que el juego es completado. El análisis de la no linealidad del juego, así como la aleatoriedad de algunos de sus elementos, como la no colaboración de miembros del equipo, o la posibilidad de que algunas de las labores que se comprueban en la Lista de Verificación Quirúrgica estén mal realizadas, hacen que implementación del videojuego no sea tan sencilla al perder la linealidad. Asimismo, el videojuego está compuesto por multitud de recursos de gran variedad de tipos, entre los que encontramos imágenes, animaciones, y textos.

La obtención de los recursos del videojuego se realiza mediante la extracción del paquete ejecutable standalone producido por eAdventure. Dicho paquete es un archivo “.jar”, disponible en el repositorio de SourdeForge de eAdventure, más concretamente en su carpeta Games1. Mediante la extracción de dicho paquete, no sólo se obtienen los recursos del juego.

El paquete “.jar” está compuesto por: Una carpeta “Assets” donde se encuentran los recursos pertenecientes al juego en concreto, una carpeta “gui” que contiene recursos que utilizan los juegos de eAdventure y que son genéricos para todos ellos, como botones, cursores para el ratón, e iconos; además, dentro del paquete se incluye el núcleo de eAdventure, intérprete capaz de ejecutar el juego, junto a multitud de librerías y codecs para la decodificación de vídeo. Finalmente, en la raíz de este fichero se encuentran los archivos de definición del juego, en formato XML junto a las DTD que permiten la interpretación correcta de dichos ficheros. Adicionalmente, este paquete es interpretable por eAdventure, por lo que permite el estudio del videojuego de una forma más visual y aclaratoria. Abrir el paquete con esta plataforma y dedicar un tiempo a analizar cómo se han construido algunos de los elementos que lo forman permite una mejor elección de la estrategia que se llevará a cabo para la implementación de dicho juego en Unity.

Cabe destacar la existencia de ficheros de Animación dentro de la carpeta “/assets/animation/”. Estos ficheros son el formato “.eaa”, y son generados por eAdventure. Aunque en primera instancia parezcan un formato propio de dicha plataforma, estos ficheros son interpretables por un editor de texto, y son descripciones en XML de cómo se debe realizar la animación. Asimismo, la DTD de dicho fichero XML se puede encontrar en la raíz de archivos del videojuego.

Un análisis en profundidad de los ficheros de definición del videojuego, es decir, de los ficheros XML, explica que, dentro de dicho fichero XML podemos encontrar las definiciones de los siguientes elementos: Escenas, Personajes, Objetos, Objetos de Atrezo, estados de la máquina de estado, Macros, Efectos, Acciones... Además, dentro de dicho archivo se especifican todas aquellas condiciones que, dependiendo del estado del juego, causan la representación de escenas de una forma diferente, así como las diferentes respuestas contextuales que podemos obtener al interactuar con algún personaje en un momento concreto, junto a los Grafos de diálogo. Cabe destacar que dichos archivos de definición alcanzan un gran tamaño, siendo, en el caso de Checklist, de una longitud aproximada de 10.000 líneas

de código, sin contar los archivos de animaciones que también definen más elementos del juego.

Encontrar dichos grafos separados del código plantea un dilema a la hora de la implementación. Se nos presentan dos opciones. La primera opción consiste en transcribir los diálogos a mano e incluirlos dentro del código del programa, y la segunda opción consiste en mantener diálogos y textos en general, separados del código del programa. Ambas opciones son válidas y ambas se realizan en la implementación de cualquier tipo de software. Un análisis breve nos lleva a las siguientes conclusiones:

- Transcribir el texto a mano e incluirlo dentro del código del programa es laborioso, y ese tiempo es mucho más valioso invertirlo en realizar un buen diseño de la arquitectura del videojuego. En general, el coste del salario de cada trabajador es más alto cuanto más compleja sea la labor que desempeña, y mayor sea su cualificación, es por ello que el precio de un arquitecto de software y el de un programador no es el mismo. Por ello invertir el tiempo en realizar una labor de más alto nivel, reducirá costes
- Pese a que existen herramientas para la localización de programas mediante el análisis del código en busca de fragmentos de texto para traducir [8], Incluirlo dentro del código del programa, más concretamente dentro del código de cada uno de los personajes, dificulta esta tarea de localización, teniendo que volver a producir el juego una vez se haya traducido y sus elementos se hayan adaptado a la cultura en la cual se va a desplegar dicho programa.
- Finalmente, la obtención de los textos de un archivo externo permite la reutilización del software que desempeñe esa función, para otros proyectos similares, pudiendo realizar la interpretación de textos para otros videojuegos de eAdventure.

Es habitual encontrar los textos de un programa separados del código. Uno de los fundamentos de la construcción de Web consiste en mantener el HTML con únicamente datos de contenido, y CSS con la definición acerca de la representación de dichos datos [10]. Otro



ejemplo es el de los programas desarrollados para Android, los cuales disponen de un archivo llamado “strings.xml”, junto a muchos otros archivos similares, en donde se incita a los desarrolladores a incluir todos los textos del programa, como los títulos de los menús, los nombres de los campos de los formularios, o las descripciones de algunos elementos[9]; y que facilita la traducción del programa cuando un usuario de la Google Play Store de otro país instala dicha aplicación. En la reconstrucción de Checklist sobre Unity se opta por mantener separados el código y los textos, y realizar un análisis con la librería System.Xml de C#, extrayendo el contenido de los diálogos de los ficheros de definición del juego.

Analizando en profundidad el último punto de la anterior lista de conclusiones acerca de mantener el código separado del texto, remarcando la posibilidad de reutilizar dicho software, y pudiendo obtener de los archivos de definición del juego, todas las descripciones de los elementos, junto a los diálogos, efectos, condiciones, macros, personajes, etc. Se plantea la opción de poder generar una herramienta capaz de interpretar cualquier juego de eAdventure, sin ninguna implementación específica, y “reproducir” dicho juego en Unity.

La implementación de dicha herramienta capaz de interpretar cualquier juego es más costosa frente a una implementación centrada y enfocada en Checklist, pero, la realización de dicho esfuerzo ahorrará mucho más esfuerzo en el futuro cuando se desee aprovechar muchos de los otros juegos generados con eAdventure.

Por todo ello, se tomó la decisión de realizar un intérprete de juegos de eAdventure, y, habiendo alcanzado dicho punto, no mucha más información es necesaria para comenzar con el diseño e implementación del videojuego.

# Capítulo 3

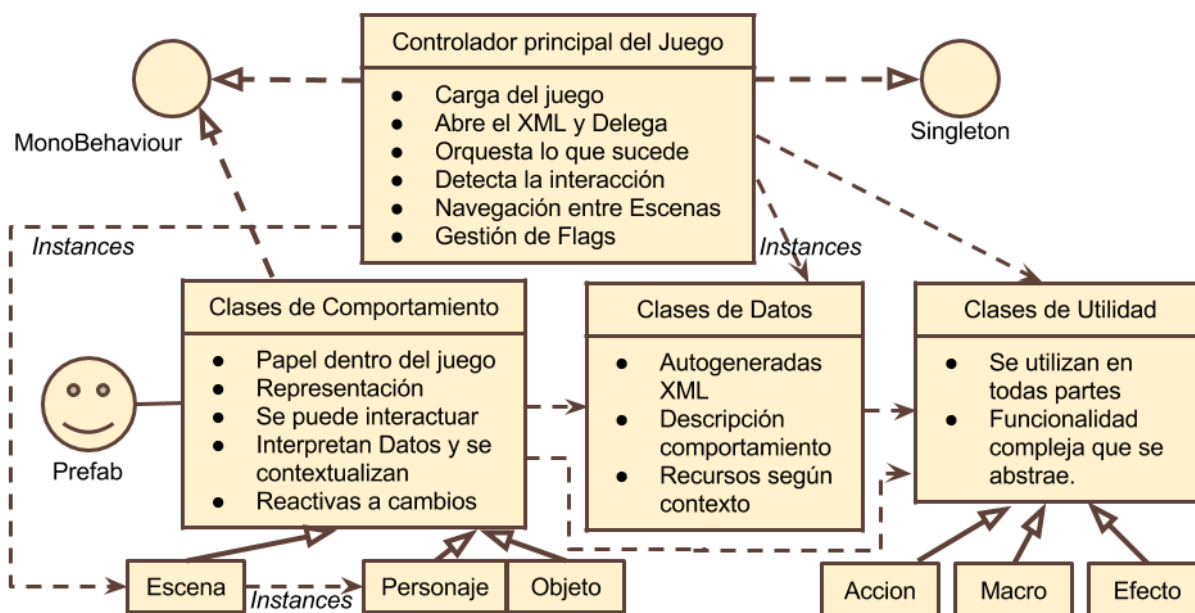
## Primera Iteración: Prototipo inicial

### 3.1. Arquitectura del Proyecto

La arquitectura del proyecto está caracterizada por la dinamicidad y capacidad para adaptarse de sus clases. Permitiendo que tomen un rol u otro dependiendo de las clases que las definan. Es decir, una clase personaje debe adaptarse al rol de personaje que le toque interpretar, y al contexto que le pone dentro de la escena. Además, debe seleccionar el recurso que lo represente que cumpla con las condiciones que se den en ese momento dentro de la escena. Finalmente debe permitir la interacción y ejecución de efectos que están asignados a si mismo.

En esencial, esto se puede explicar con el diagrama de clases muy simplificado de la figura 3. Este diagrama fue generado en esta iteración del proyecto, por lo que su formato es muy diferente al resto de diagramas que se muestran en posteriores iteraciones.

En la figura 3.1 hay varios elementos que no son comunes en un diagrama de clases. El primero de ellos es la "Cara sonriente" que aparece a la izquierda con el texto *Prefab* debajo de el. Este *Prefab* es un elemento en Unity3D que resume el concepto de Objeto Prefabricado, y que, contiene, una representación tridimensional, junto a todas las componentes y comportamientos que definen dicho objeto. Por otra parte la interfaz *MonoBehaviour* no es una interfaz, sino una clase abstracta, y esta, provee a los elementos que la extienden de la habilidad de ser componentes para los objetos de la escena. Dicha clase contiene varios



**Figura 3.1:** Diagrama de clases simplificado de la arquitectura de clases de la primera iteración del proyecto.

métodos como `Start()`, `Awake()` o `Update()` que permiten a estos comportamientos inicializarse, actuar, y en esencia, tener una fracción de tiempo para realizar acciones dentro de la escena de Unity3D.

En la figura 3.1 encontramos cuatro tipos de clases:

- El controlador principal del juego.
- Las Clases de Datos
- Las Clases de Comportamiento: Escenas, Personajes, Objetos, Salidas, etc...
- Las Clases de Utilidad: Acciones, Efectos, Macros, Secuencias, Diálogos, etc...

Todas las clases que se presentan en este primer prototipo han sufrido cambios muy grandes o totales tanto en su código, como en su funcionalidad. Algunas de ellas han desaparecido, surgiendo clases nuevas que realizan su funcionalidad, otras se han transformado

completamente o han sido asumidas por otras, y finalmente, otras han surgido de la generalización y refactorización de código.

### **3.1.1. Controlador Principal de Juego**

El controlador principal de juego, implementado con una clase de nombre `Game`, se encarga de, en primer lugar, comenzar la carga del juego, comenzando a explorar el XML y delegando en las Clases de Datos su propia interpretación, así como orquestar lo que sucede dentro del juego, detecta la interacción por parte del jugador y notifica a aquellos elementos cuando se interactúa con ellos, realiza la navegación entre escenas cuando es necesario, y controla el estado de los interruptores que controlan las condiciones para la contextualización de cada escena en su momento apropiado.

Implementa el patrón Singleton, pues sólo una instancia es necesaria para el control.

### **3.1.2. Clases de Datos**

Las clases de datos se encargan de generarse a partir de un elemento del fichero XML y sirven de descripción para una Clase de Comportamiento para representarse y interactuar de la forma correcta en cada momento. A través del documento XML realizan una lectura del mismo mediante la librería de `c# System.Xml`, y rellenan sus atributos a partir de la especificación en dicho XML. Asimismo implementan funciones útiles como la obtención de un paquete de recursos apropiado según el contexto.

Existe una clase de datos por cada Clase de Comportamiento, aunque también existen clases de datos auxiliares mucho más pequeñas y que apenas aportan funcionalidad, siendo meros contenedores de datos.

### **3.1.3. Clases de Comportamiento**

Las clases de comportamiento son aquellas clases que juegan un papel dentro del videojuego y participan de forma activa en la representación del mismo. Dentro de esta categoría encontramos a las Escenas, los Personajes, los Objetos, las Salidas de las escenas, así como

los objetos de atrezzo y todos aquellos elementos que tienen una representación virtual en el videojuego.

Cada una de estas clases heredar  de la clase de Unity *MonoBehaviour*, que indica, como se explica en la secci n 3.1, que dicha clase es un comportamiento y se puede asignar como componente a un elemento de la escena. Cada una de ellas tiene un objeto de datos que debe interpretar. Esta clase de comportamiento recibir  est mulos por parte del controlador principal del juego cuando el jugador interact e con ellas. Este paso intermedio es necesario ya que hay momentos en los que el controlado principal debe evitar la interacci n del jugador con dichos elementos por encontrarse en mitad de un di logo. Con este est mulo, la clase de comportamiento gestiona las diferentes interacciones que tenga el usuario con s  mismo.

Finalmente, estas clases se encapsulan dentro de un *Prefab*, formando parte como componente de un objeto tridimensional, junto a otras componentes.

## 3.2. Detalles de la implementaci n

Existen algunas cuestiones cruciales y generales acerca de c mo se realiz  la implementaci n de diversas mec nicas que eran necesarias para que el juego funcionase.

Algunas de estas decisiones cruciales son, por ejemplo, la decisi n de *reRenderizar*<sup>1</sup> la escena junto con todos sus elementos cada vez que un *Flag*<sup>2</sup> o una Variable de entorno cambia. Esto puede tener efectos devastadores si nuestra escena tiene elementos que han sido modificados por el jugador en un momento determinado, pero en el caso del videojuego Checklist, no existen dichos elementos, por lo que, cada vez que se activa un flag, en lugar de notificar a cada uno de los elementos, se *renderiza* de nuevo la escena entera.

Esta decisi n se tom  debido a que, pese a que exist a la opci n de implementar el patr n Observador-Observable, y hacer que todos los elementos de la escena fueran observadores y fuesen notificados cuando el estado del juego cambie, es frecuente que ocurra que, un

---

<sup>1</sup>Renderizar es el proceso de generar una imagen, mediante el c culo de la iluminaci n, a partir de un modelo 3D. En este caso se refiere a regenerar la representaci n de la escena.

<sup>2</sup>Una Flag, bandera en Ingl s, es un elemento que se utiliza para establecer hitos dentro del juego, como haber hablado con un personaje, o haber fallado una respuesta.

elemento de la escena que en ese momento no tiene representación física, deba de reaccionar a este cambio en el estado del juego, y debido a que nunca recibirá la notificación, porque no existe, nunca aparecería.

No obstante esta decisión de volver a generar la escena, pese a que se mantiene, evolucionará en la siguiente iteración del proyecto, permitiendo al usuario modificar los elementos que se hallen dentro de la escena, guardando en un diccionario, referencias a los contextos dentro de las mismas. Por el momento no se han encontrado problemas de rendimiento con esta decisión, ya que las escenas, en general, suelen estar formadas por pocos elementos. Si se encontrasen dichos problemas, se desarrollaría un método más eficiente.

Otra de las decisiones cruciales de la implementación consistió en decidir si generar un *SequenceManager* que gestiona por completo las secuencias que se dan, como efectos, o diálogos; o por otra parte, hacer que las propias clases de *GraphConversation* y *Effect* fuesen capaces de ejecutarse por sí solas.

Después de comenzar con la implementación del *SequenceManager*, y construir una pila de secuencias, se vio que sería considerablemente complejo recordar datos contextuales acerca del estado de cada secuencia en caso de que fuese necesario parar la ejecución, y se decidió que era más sencillo que cada secuencia recordase por sí misma su posición.

En una sección posterior se detallan en profundidad los detalles de la implementación de las clases *Conversation* y *Effect*, dos clases que implementan la interfaz *Sequence*, y que permiten ser ejecutadas por si mismas.

### 3.2.1. Sobre la clase *Game*

La clase *Game* es el Controlador principal del juego, y por ello realiza todas las funciones anteriormente descritas.

Sus funcionalidades se podrían categorizar en 4 apartados:

- En primer lugar, y en la parte de más arriba del código se encuentra la gestión del *singleton* y del acceso a las variables de entorno. Es decir, se facilita el acceso a las

*Flags* del juego, a las Variables, a la obtención de *Macros*, Las especificaciones de los diálogos, etc. Esta parte es vital para el control del juego, y su estado.

- En segundo lugar, tras todas estas funciones, se encuentra el cargador de juego. Se ha implementado mediante el uso de un *Thread*<sup>3</sup> que permite visualizar el estado de la carga del juego mientras esta se realiza, y, como Unity no permite cargar recursos con *Resources.Load()* fuera del bucle principal del juego, se utiliza la librería del sistema *System.IO.File* para leer directamente los ficheros y generar texturas con ellos. Aquí entra en juego la clase *Texture2DHolder* de la que se realiza una explicación posteriormente.

Este cargador prepara el fichero XML utilizando la librería del sistema *System.Xml*, y lo secciona en pequeños bucles, uno por cada elemento del juego, es decir, uno por personajes, otro por objetos, otro por escenas, etc. Cuando termina la carga, se cambia el estado de la clase *Game* para comenzar con la visualización del juego.

- En tercer lugar, se encuentran determinadas funciones útiles que realizan tareas variadas como *RenderScene*, para pintar una escena o *Execute* para comenzar la ejecución de una secuencia, junto con la función *Update()* donde se realiza parte del control del juego, junto con la gestión del click del usuario, donde bloquearemos dicho click, notificaremos al elemento clicado o simplemente extraemos las acciones disponibles de dicho elemento, dependiendo del estado del juego.
- Por último, y al final del código, se encuentra otra parte de interacción, junto con el pintado de la interfaz. Para la interfaz se ha generado una *GUISkin*<sup>4</sup> propia que se modifica en función del tamaño de la pantalla para hacerse más grande o pequeña dependiendo de su proporción. Para el pintado de la interfaz se ha utilizado *GUILayout*,

---

<sup>3</sup>Un Thread, en programación, es un proceso que se ejecuta de forma paralela a la ejecución del programa, por lo que permite la realización de múltiples tareas simultáneamente

<sup>4</sup>Una *GUISkin* es un fichero de especificación de datos de Unity que contiene datos acerca de los elementos de la interfaz, como el tamaño de la fuente, colores de los fondos, o imágenes de los botones.

clase que facilita métodos para representar una interfaz, ya que se conocía su funcionamiento de haberla utilizado en proyectos anteriores, y era sencillo de implementar.

Esta GUI, además de adaptarse, se ha preparado para que se posicione correctamente utilizando *Camera.current.WorldToScreenPoint()*, es decir, que transforma los puntos del mundo tridimensional, a unas coordenadas de cámara, y mediante una serie de transformaciones, se colocan los cuadros en su lugar, evitando que estos cuadros de diálogo se salgan de la pantalla cuando un personaje habla desde uno de los extremos.

Esta explicación cierra, por encima, todos los elementos importantes de la implementación de la clase *Game*. Debido a que esta implementación no es la final, no se incluyen explicaciones más detalladas ni diagramas.

Esta clase es una de las clases que ha sufrido una mayor evolución en la segunda iteración, ya que, pese a que sigue controlando la interacción por parte del usuario, ya no realiza la lógica de dicha interacción en el bucle *Update()*, sino que, simplemente delega en los diferentes elementos interactuados para que sean ellos los que realicen la lógica de su interacción. Asimismo, la gestión de la interfaz ha sido delegada en la clase *GUIManager*, con un nuevo y rediseñado sistema de burbujas.

### 3.2.2. Sobre las Clases de Datos

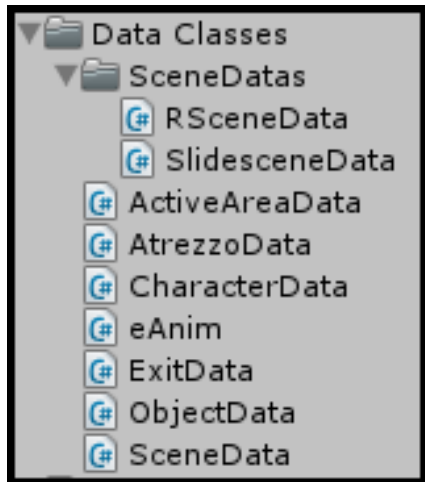
Existen multitud de clases de datos dentro del código del programa, aunque todas se parecen bastante entre sí, por lo que se explicarán la mayoría de ellas en conjunto.

Estas clases se auto-generan analizando un *XmlElement* que reciben en su constructor, y por lo general no aportan funcionalidad, salvo excepciones como la función *Check()* que implementan la mayor parte de recursos para verificar si se cumplen todas las condiciones para que dicho recurso pueda ser utilizado.

La mayor parte de clases de datos contienen:

- Una serie de Recursos: con soporte de videos y animaciones (implementadas con la clase *eAnim* y *eFrame*). Algunas clases implementan su propia clase para manejar sus





**Figura 3.2:** *Vista de carpeta de las clases de datos implementadas en la primera iteración.*

recursos con Condiciones, como `CharacterData` con su `CharacterResource`, o `SceneData` con su `SceneResource`, aunque muchas otras simplemente utilizan diccionarios de tipo `Dictionary<string,Texture2DHolder>Resources`.

- También tienen una serie de condiciones ante las que decidiran representarse o no, estas condiciones son gestionadas mediante la clase `Condition`.
- Algunas clases de datos, además de Recursos y Condiciones, también tienen Efectos que se realizan al interactuar con ellas, o un listado de Acciones que se muestra como un menú contextual al interactuar con ellas.

Particularmente, existen algunas anomalías en las clases de datos, como son las clases que implementan la interfaz `SceneData` la cual contiene, además de Recursos como todas las demás, gran variedad de objetos `ItemReference`, los cuales contienen una referencia a un elemento de la escena (ya sea un personaje, objeto o una salida) y la contextualización del mismo, es decir, su posición en ese momento, las condiciones que tienen que darse para que ese elemento aparezca o no, etc.

La creación de dicha interfaz fué necesaria debido a que existen multitud de tipos de escenas. En este caso se han implementado 2 tipos de escenas que eran necesarias para

el videojuego *Checklist*, las *RScene*, abreviando *Regular Scene*, que son escenas normales sobre las que jugar, donde se representan personajes, objetos, salidas, áreas activas; y las *Slidescene*, que son escenas con una sucesión de imágenes en su interior a modo de *Slides* como si de una presentación se tratase.

Finalmente comentar que, algunas clases de datos como los *ObjectData* o los *ActiveAreaData* tienen un elemento llamado *InfluenceArea* el cual no se ha implementado, y que permite que un objeto o un active área sea mucho más grande que su area de interacción, la cual está formada por una sucesión de puntos que generan al final un polígono. Esto permite que pueda haber un objeto con formas irregulares e interactuar con él sólo en las zonas de real contacto con la imagen. Por lo que todas nuestras áreas de influencia serán rectangulares.

Esta Área de Influencia, sin embargo, en la siguiente iteración del proyecto sí que ha sido implementada, mediante la utilización de librerías de triangulación de polígonos irregulares de cualquier tipo, y la transformación de dichos polígonos en Mallas tridimensionales. La explicación de dicha transformación será realizada en la siguiente iteración.

Finalmente, la clase *eAnim* tiene una peculiaridad, y es que en versiones antiguas de *eAdventure*, las animaciones se generaban a partir de secuencias de imágenes en la carpeta *animations*, carpeta utilizada para el almacenamiento de las imágenes que forman las animaciones.

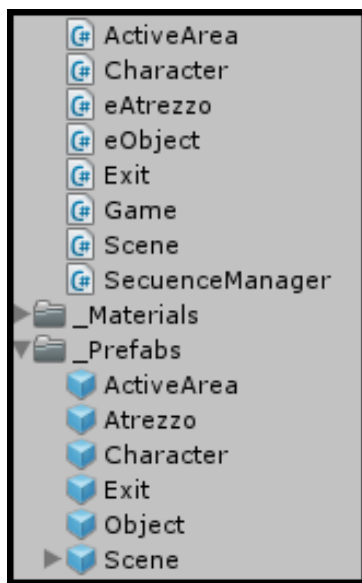
Es decir, existen animaciones formadas de la siguiente manera: “anim\_01.png”, “anim\_02.png” y estas están especificadas en el fichero de especificación del juego de *eAdventure* como “/animaciones/anim”. Esto supone dos problemas. El primero consiste en identificar cuál es el formato del fichero de imagen, por lo que existe una lista de tipos disponibles y se intenta encontrar un archivo que satisfaga tanto el nombre de la animación, como el formato correspondiente. Y en segundo lugar, no se sabe cuántos archivos hay, lo que hace que debas iterar hasta que no encuentres fichero.

Por otra parte, el nuevo formato de animaciones está definido en ficheros XML llamados

“*.ea*” que son mucho más sencillos de recorrer y contienen más información acerca de cada fotograma de la animación.

### 3.2.3. Sobre las Clases de Comportamiento

De manera similar al apartado anterior, se explicará el funcionamiento de la mayor parte de clases de comportamiento, puntualizando detalles acerca de algunas de dichas clases en concreto, como *Character* o *eObject*, que tienen comportamientos aislados como la gestión de fotogramas o la necesidad de cambiar cuando el usuario va a interactuar con ellos.



**Figura 3.3:** Vista de carpeta de las clases de comportamiento implementadas en la primera iteración, junto a sus *Prefabs*.

Las clases de comportamiento son todas herederas de *MonoBehaviour* por lo que se asignan a objetos dentro de la escena, y luego estos objetos se guardan en forma de *Prefabs*, cada uno a uno individualmente. Estos *Prefabs* suelen ser muy sencillos, en su mayoría constituidos únicamente por un *Quad*<sup>5</sup>, al cual se le cambia la textura con *this.GetComponent<Renderer>().material* y posteriormente, mediante la información de dicha textura, se aplican modificaciones escalando el *Quad*.

---

<sup>5</sup>Un *Quad* es una malla tridimensional compuesta por un plano, una de las mallas más simples.

Todas estas clases de comportamiento son la representación de algún elemento de la escena, y por ello tienen en su interior los siguientes elementos:

- En primer lugar una clase de datos asignada que será el “Guión” de nuestra clase para enrolarse. Es decir: una clase *Character* contendrá dentro una clase *CharacterData*, o una clase *eObject* (la letra e era necesaria porque la clase *Object* ya existe en el espacio de nombres, y proviene de juntar eAdventure con *Object*), contiene una clase *ObjectData*.
- En segundo lugar, aunque no todas la tienen, una *ItemReference*, que utilizan para contextualizarse. Un *character* no sería capaz de conocer su posición dentro de la escena si no tuviera un *ItemReference* para poder contextualizar, y tampoco sabría si ese es su momento de aparecer en la escena, o si no debe aparecer más.
- Por último contienen aquellas funciones que son necesarias para su correcto funcionamiento.

## Detalles de la clase *Character*

La clase *Character* contiene una serie de comportamientos necesarios y características específicas que hacen que sea diferente a las demás, pues tiene animación y ha de ir cambiando el fotograma que la representa con el paso del tiempo, por lo que en su función de comienzo de la ejecución, *Start()*, selecciona la animación que cumple todas las condiciones, y establece el primer fotograma, y en su función de actualización del estado, *Update()*, va cambiando su fotograma con la función *ChangeFrame()*.

Esta función *ChangeFrame()* accede al *renderer*<sup>6</sup> y cambia su textura además de actualizar determinadas variables de control, que establecen cuando deberá realizarse el próximo cambio de fotograma, o el fotograma que se está representando actualmente.

---

<sup>6</sup>El *renderer*, en Unity3D, es la componente de los objetos que tienen representación en la escena, que define cómo se representan en la misma, con características como los materiales que las conforman, las texturas, las mallas tridimensionales, y otros elementos.

## Detalles de la clase *eObject*, *Exit* y *ActiveArea*

Estas tres clases tienen la peculiaridad de que deben ser reactivas a interacción por parte del jugador, y mostrar algún cambio en su representación cada vez que el jugador pase el puntero del ratón por encima de ellas. Esto se implementó utilizando las funciones que facilita la clase *MonoBehaviour*, *OnMouseExit()* y que reciben el evento del puntero cuando este entra en su área de reacción y cuando sale de la misma.

Cada una de las tres clases que son reactivas a este evento realiza una acción diferente, donde, el *eObject* cambia su imagen y, por otra parte, *Exit* y *ActiveArea*, acceden a su material para cambiar su color, volviéndose de color rojo la representación de *Exit*, y de color verde la representación de una *ActiveArea*.

Adicionalmente, la clase *Exit* tiene una función *exit()* que se encarga de ejecutar todos los efectos que conllevan la salida de la escena, así como de mandar el renderizado de la nueva escena.

Esta clase *Exit*, en esta versión del intérprete, no funcionaba correctamente, pues una salida puede contener un *not-effect*, es decir, un efecto que se debe de ejecutar si se intenta salir de la escena cuando las condiciones no lo permiten. Esto implica que, la salida debe de gestionar su propia representación, y la Escena debe delegar esta funcionalidad en la salida.

Por otra parte, y como ya se mencionó en un apartado anterior, las *ActiveAreas* contienen una funcionalidad adicional en la última versión del intérprete, ya que estas se adaptan a un Área de Influencia, que modifica su contorno y forma, adaptándose a la forma de un polígono irregular definido por un listado de puntos en el fichero de especificación del juego.

## Detalles específicos de *Scene*

La clase *Scene* supone un reto más complejo de abordar, en comparación con todas las demás clases de comportamiento, pues su clase de datos puede ser de diferentes tipos dependiendo de qué clase implementa la interfaz. Es decir, una *Scene* puede ser a su vez una *VideoScene*, una *CutScene*, una *SlideScene*, o cualquier otro tipo de escena que exista.

Cada una de estas escenas tiene comportamientos individuales, pero para la gestión de los mismos, surgen una serie de problemas de implementación.

Cambiar las componentes de un Prefab en ejecución no es una práctica recomendada en Unity3D, además de que existen problemas a la hora de realizar una herencia de la clase `MonoBehaviour` en una subclase, y extender esta subclase en más clases aún, pues siempre se deben invocar desde la clase hija, los métodos implementados de la clase `MonoBehaviour` en la clase padre. Es decir, una clase hija debe ejecutar, por ejemplo el metodo *base.Start()*, antes de continuar con la ejecución de su código.

La implementación de la escena, por lo tanto, se realizó mediante una bifurcación en los comportamientos utilizando como elemento bifurcador el tipo de escena que la escena está almacenando para su representación. En esencia, se programan delimitadores con *Switch(sceneData.type)*, para ejecutar una labor determinada en función del tipo de escena que se desee.

Esto le permite a una escena normal, Renderizar todos los personajes de la escena, o, a una *SlideScene*, establecer su fondo y animarse cuando el jugador realice una interacción para cambiar de diapositiva.

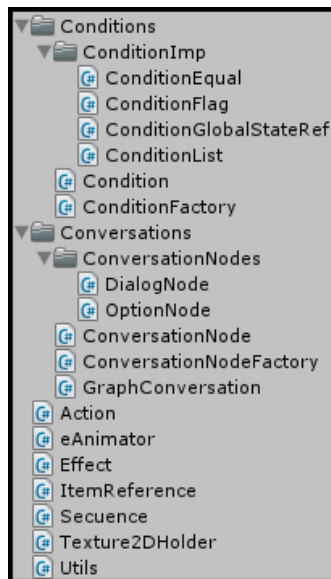
Asimismo, las escenas tienen una función *Interacted* que hace que las escenas en sí mismas puedan ser interactivas cuando no contienen ningún elemento como las *Slidescenes* que han de cambiar al hacer click, o como, por ejemplo las *VideoScenes* (que no se han implementado), que tienen una secuencia en su interior y debe ejecutarse al interactuar.

Esta función *Interacted* será refactorizada en la última versión, generando al interfaz *Interactable*, que permite ejecutar la función *Interacted*, y otras funciones, sobre los objetos que tengan que ser reactivos a interacción, para poder así delegar esta funcionalidad en las clases interactuadas, en lugar de extraer la funcionalidad y ejecutarla en el bucle principal de juego.

### 3.3. Sobre las Clases de Utilidad

Debido a la necesidad de generar determinadas clases que contengan funcionalidad que no debería estar ubicada en ninguna clase en concreto, la necesidad de que existan clases que no son comportamientos ni datos, o debido a la necesidad de refactorizar y rediseñar código que surgen las clases de utilidad. Clases que son utilizadas por multitud de clases en muchos contextos.

Dentro de estas clases de utilidad, encontramos aquellas que han sido extraídas de funcionalidad de eAdventure, como son las condiciones, las conversaciones, los efectos y las acciones. Todas estas clases de utilidad son clases híbridas que unifican la utilidad de una clase de datos con funcionalidad adicional como la capacidad de ejecutarse a si mismas y comunicarse con la clase controladora del juego para solicitar que realice tareas. Todas estas clases realizan tareas importantes para el desarrollo del juego, o facilitan el desarrollo de funcionalidad adicional.



**Figura 3.4:** Vista de carpeta de las clases de utilidad implementadas en la primera iteración.

La explicación de las clases de utilidad se realiza de forma separada debido a que las clases de utilidad contienen funcionalidad muy diferente entre ellas y no se puede unificar

la explicación de las mismas.

### 3.3.1. La clase Texture2DHolder

La clase Texture2DHolder surge debido a la necesidad de realizar carga de imágenes de manera transparente al usuario, solicitando únicamente al Texture2DHolder que realice la carga de una imagen, a partir de un directorio o unos bytes. Asimismo, esta ayuda a la carga y transformación de imágenes en hilos, ya que Unity3D no permite la carga mediante el método que facilita para carga de recursos, *Resources.Load()*, fuera del bucle principal de ejecución.

Para solventar el problema de la carga paralela de recursos, en lugar de utilizar *Resources.Load()*, se utiliza la librería de sistema, System.IO, utilizando carga de bytes directamente, almacenándolos temporalmente en un *array* de *bytes* dentro de si mismo. para obtener los datos de los ficheros directamente.

Esto causa un pequeño problema a la hora de generar un paquete, y es que, si se desea generar un juego con los recursos incluidos dentro del sistema de empaquetado de Unity, no se podrá acceder a estos recursos, ya que la librería System.IO busca los ficheros en el sistema de ficheros, y *Resources.Load()*, en su lugar, obtiene dichos recursos de los paquetes comprimidos de recursos generados. En la versión final del proyecto, ambos tipos de cargas están permitidos.

Asimismo, Unity no permite generar una *Texture2D*<sup>7</sup> si no es en tiempo de ejecución, por lo que, en el momento de la carga sólo se obtienen los bytes, y no es hasta el momento en el que la textura se utiliza por primera vez.

---

```
public Texture2D LoadTexture(bytes[] fileData)
{
    Texture2D tex = new Texture2D(2, 2, TextureFormat.BGRA32, false);
    tex.LoadImage(fileData);
    return tex;
}
```

---

<sup>7</sup>Una Texture2D es la clase que se utiliza para guardar texturas bidimensionales en Unity3D, como imágenes, fotografías, o sprites.



Tras esto la textura queda guardada y no se vuelve a generar, por lo que se reduce el tiempo de ejecución del programa, a coste de sacrificar memoria. En este caso, como las texturas de los videojuegos generados con eAdventure no ocupan más que el propio juego en sí mismas, se asume que, el coste en memoria de la aplicación no será muy elevado.

### 3.3.2. La interfaz *Sequence*: *Effect*, *Conversation* y *Condition*

)

La interfaz *Sequence* surge de la necesidad de generar una interfaz para interactuar con los efectos y las conversaciones, elementos que son capaces de ejecutarse y realizar tareas que pueden tener alto nivel de complejidad. Esta interfaz, en esencia, provee de un método *public bool execute()*, la cual ejecuta el contenido de sí mismas, y devuelve verdadero en caso de que, la secuencia necesite esperar, ya sea por la necesidad de interacción del usuario, o por que un elemento debe ejecutarse un tiempo determinado.

Pese a que la interfaz *Sequence* seguirá existiendo en la versión final del proyecto, se ampliará utilizando la interfaz *Interactable*, para poder solicitar al usuario que interactúe con algún elemento, ya sea una secuencia, o un elemento interactivo de la escena.

#### La clase *Effect*

La clase *Effect* es una clase que es híbrida entre Clase de Datos y Clase de Utilidad, pues recibe un *XmlElement* para generarse, y sin embargo aporta funcionalidad por sí misma. Lo que hace dicha clase es: Obtiene un listado de *EffectNode* y de *Condition* y, si existen condiciones, las asigna a cada nodo para que este únicamente se ejecute si cumple las condiciones que permiten su ejecución.

Esta clase contiene dentro de su fichero la clase *EffectNode* el cual, a su vez, contiene cada una de las descripciones de los nodos de una secuencia de efectos. Se implementó en esta iteración del proyecto con una bifurcación, utilizando un *Switch*, en lugar de hacer herencia debido a la cantidad de nodos diferentes, no obstante, en la versión final del proyecto, existen clases para tratar cada uno de los nodos de efecto que componen la secuencia de efectos.

Los *EffectNode* realizan tareas como activar una *Flag*, poner una variable, ejecutar una *Macro*, o ejecutar una condición, por ello tienen dos variables de control que definen su funcionamiento, y una función *execute()* que los pone en funcionamiento. Estas dos variables de control sirven para determinar si dicho nodo sólo se ejecuta una vez, o varias, y para controlar cuántas veces se ha ejecutado ya dicho nodo. Hasta que un nodo no determina que ha completado su ejecución, este no permite que la secuencia se siga ejecutando.

## Las clases *GraphConversation* y *Condition*

Estas clases son las más extensibles de todas las clases que se implementaron en la primera iteración del proyecto. Ambas tienen una factoría para generarse, ya sea *ConditionFactory* y *DialogNodeFactory*, y se generan automáticamente a partir de un *XmlElement*, que contiene la especificación extraída del archivo de especificación del juego.

Por su parte *GraphConversation* solicita a *DialogNodeFactory* un nuevo *DialogNode* por cada uno de sus hijos, y así se genera el diálogo. La “magia” de *DialogNodeFactory* y de *ConditionFactory* es que utilizan *System.Linq*<sup>8</sup> para examinar el espacio de nombres y obtener de ahí todas las clases que implementan la interfaz *DialogNode* o *Condition*. Esto se realiza de la siguiente manera:

---

```
types = System.AppDomain.CurrentDomain.GetAssemblies().SelectMany(s =>
    s.GetTypes()).Where(p => typeof(ConversationNode).IsAssignableFrom
    (p)).ToList();

types.Remove(typeof(ConversationNode));
```

---

Y una vez que se ha obtenido la lista de tipos que implementan dicha interfaz, se realiza una búsqueda secuencial, preguntando a cada clase si puede realizar la lectura de dicho *XmlElement*. En caso afirmativo, se le pide que se construya a partir de dicho documento, y la factoría devuelve dicho elemento. Las funciones *canParseType()* y *parse()* están dentro de las interfaces *DialogNode* y *Condition*. Esto se ve representado en el código que se presenta

---

<sup>8</sup>El espacio de nombres *System.Linq* proporciona clases e interfaces que admiten consultas que utilizan Language-integrated query (LINQ)

a continuación:

---

```
foreach (System.Type t in types)
{
    ConversationNode tmp = (ConversationNode) System.Activator.CreateInstance(t);

    if(tmp.canParseType(node.Name))
    {
        tmp.parse(node);
        return tmp;
    }
}
```

---

De esta forma, obtenemos un diseño sencillo y cómodo de extender, pues añadir nuevos tipos de nodo de conversación, o tipos de condiciones es una tarea tan sencilla como generar una nueva clase que implemente esas interfaces, y automáticamente, y sin la necesidad de incluir la existencia de dicha interfaz en la factoría, automáticamente se utiliza.

Si se desea entender o extender el conocimiento de la implementación de estas clases se recomienda mirar el código de estas clases, pues es bastante sencillo y da una visión mucho más clara de su funcionamiento.

### 3.3.3. La clase Action

Esta clase es tan simple como un efecto con representación. Está compuesta por un Effect que se ejecuta cuando se realiza la acción, una Condition que determina cuando esa acción está disponible, y una serie de recursos que se utilizan para su representación. La gestión de la visualización de las acciones se realiza en Game, en la función OnGUI().

## 3.4. Acerca de los elementos de Adaptación

El juego Checklist cuenta con un elemento adicional para el jugador, las Adaptation estas no son más que una serie de estados iniciales del juego para que, el juego pueda adaptarse al jugador, ya sea por motivos de accesibilidad o por motivos de nivel. Las Adaptation se han implementado, pero no existe ningún mecanismo que permite decidir cuál Adaptation

se selecciona en cada momento, sino que se selecciona una aleatoriamente al comienzo del juego, y tras esto se trabaja siempre con la misma.

# Capítulo 4

## Segunda Iteración

Como se explica en el la subsección 1.4, en la que se expone la metodología de desarrollo utilizada para el proyecto, este sufre de tres grandes iteraciones para su completitud. En esta sección se explican los detalles implementados en la segunda iteración, es decir, la generación de un mejor diseño de la aplicación, nuevas interfaces, refactorización de clases y código replicado, así como mejoras gráficas visuales dentro de la aplicación, incluyendo referencias a la sección 3, explicando los cambios en diseño y comportamiento.

### 4.1. Arquitectura del Proyecto

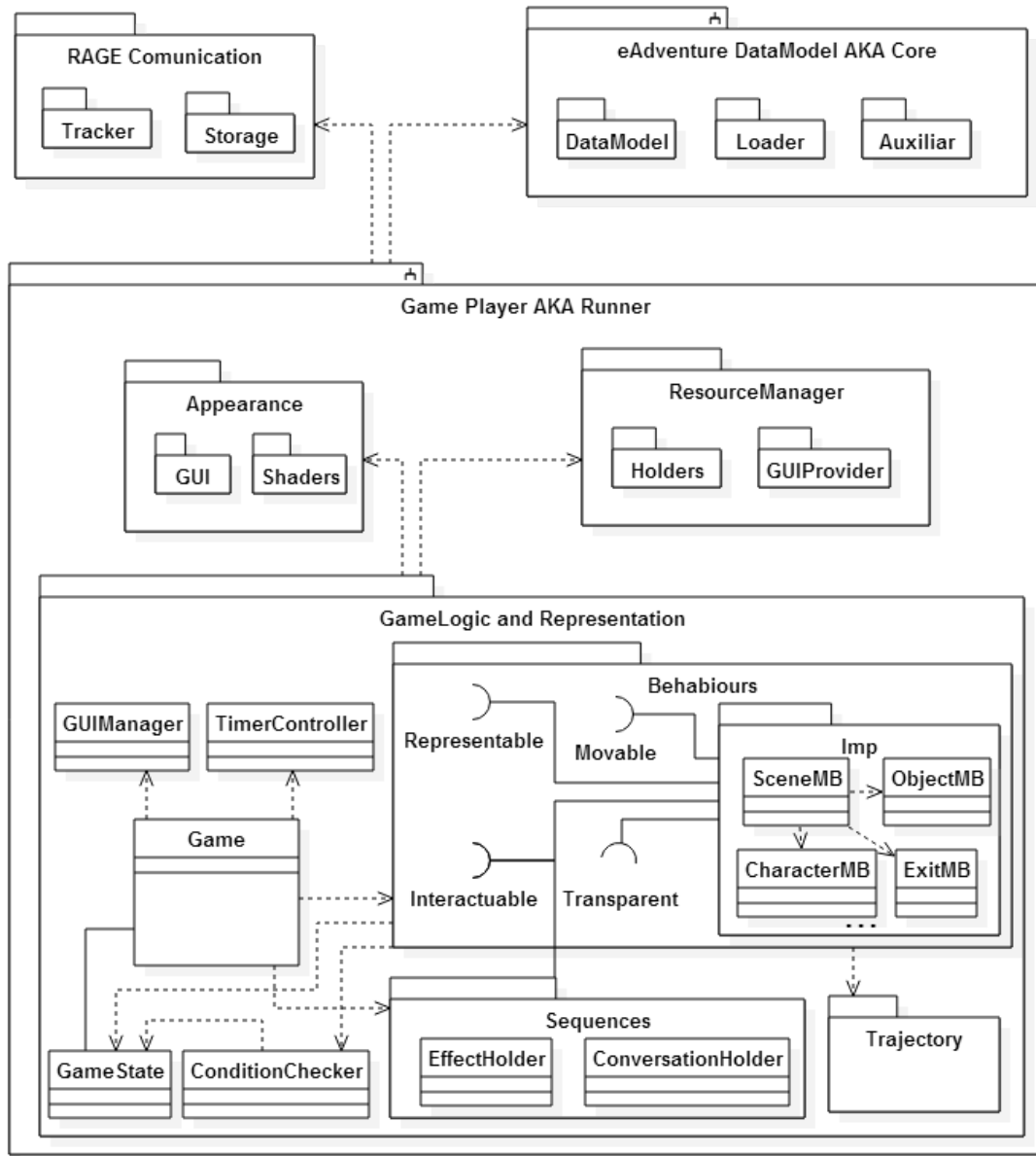
La arquitectura del proyecto ha cambiado radicalmente desde la primera iteración, en la que estaba centrada en cómo se debería tratar el documento de especificación de juego, incluido dentro del paquete del juego, en forma de XML para que, cada elemento fuese lo más dinámico posible y sostenible. A su vez, estaba centrada en la clase *Game*, y existían multitud de clases, llamadas clases de utilidad, que contenían funcionalidad que debía ser accesible a través de una clase controladora de nivel superior.

En esta arquitectura encontramos tres paquetes importantes que engloban la funcionalidad necesaria para la ejecución del proyecto. Están representados en la figura 4.1 Estos tres paquetes son:

- **Core:** *Core* contiene el núcleo de eAdventure portado a Unity. Este paquete está for-

mado, a su vez, por varios subpaquetes. El primero de ellos es *DataModel*, donde se encuentra todo el modelo de datos portado de eAdventure a Unity, siendo completamente fiel a la implementación realizada en Java, y soportando todas las funcionalidades que se soportaban en dicho editor. El segundo de ellos es *Loader*, donde está contenido el núcleo de lectura y carga del fichero XML de especificación del juego. Este *Loader*, recibe un XML y genera un objeto *AdventureData* que contiene todos los datos del juego. Por último, dentro de *Core* encontramos un paquete *Auxiliar* con clases útiles para este modelo de datos.

- **RAGETracker:** que contiene los elementos necesarios para generar un registro de actividad de juego del usuario y comunicarse con RAGE a través de Unity. RAGE se encarga de realizar tareas de evaluación y *Learning Analytics* mediante el análisis de las trazas producidas por el alumno mientras juega. Esto permite al profesor que esté utilizando un juego producido con uAdventure, que se beneficie de las ventajas de RAGE, pudiendo reforzar aquellos alumnos que estén teniendo un desarrollo insuficiente o anormal en el juego.
- **Runner:** que se encarga de la transformación desde la especificación una ruta donde hay un juego descomprimido, hasta la generación de un entorno gráfico interactivo que permite al usuario jugar al juego. Contiene 3 subpaquetes que se encargan de diferentes tareas. En primer lugar, el paquete *ResourceManager*, que se encarga de la carga transparente de recursos, ya sean imágenes, videos, botones, cursores, u otros elementos multimedia. En segundo lugar, el paquete *Appearance*, que se encarga de mejorar la visualización, mediante diferentes formas de mostrar burbujas de diálogo, o mediante el uso de *Shaders*. Por último, el paquete *GameLogic and Representation* que se encarga de ejecutar la lógica del juego. Este contiene en su interior una serie de gestores y controladores que son capaces de controlar la ejecución del juego, junto a las secuencias, trayectorias, así como una serie de comportamientos que tomarán lugar dentro de la escena.



**Figura 4.1:** *Arquitectura del sistema, a nivel de paquetes de la versión final del proyecto*

#### 4.1.1. El nucleo de Ejecución: Runner

El núcleo de ejecución, también conocido como *Runner*, está dividido en 3 subpaquetes, *Apearance*, encargado de mejorar la representación visual, *ResourceManager*, encargado de realizar la carga transparente de recursos, y *GameLogic*, el cual podría considerarse el núcleo en si mismo, de lógica del juego. Estos tres paquetes surgen de la generalización de elementos que se encontraban, o bien en la antigua clase *Game*, explicada en la sección 3.2.1, o bien en alguna de las clases de utilidad, explicadas en la sección 3.3.

De esta manera conseguimos que, la antigua clase controladora del juego *Game*, y las Clases de Utilidad, ahora no tenga que realizar tantas tareas como realizaba anteriormente. En su lugar surgen 5 clases. Estas clases se encargan de hacer transparente la gestión de diversas tareas. Algunas de ellas realizan la gestión de un elemento en concreto, otras controlan la ejecución de otros elementos, y, por último otras se encargan de gestionar el estado del juego.

Estas 5 clases son:

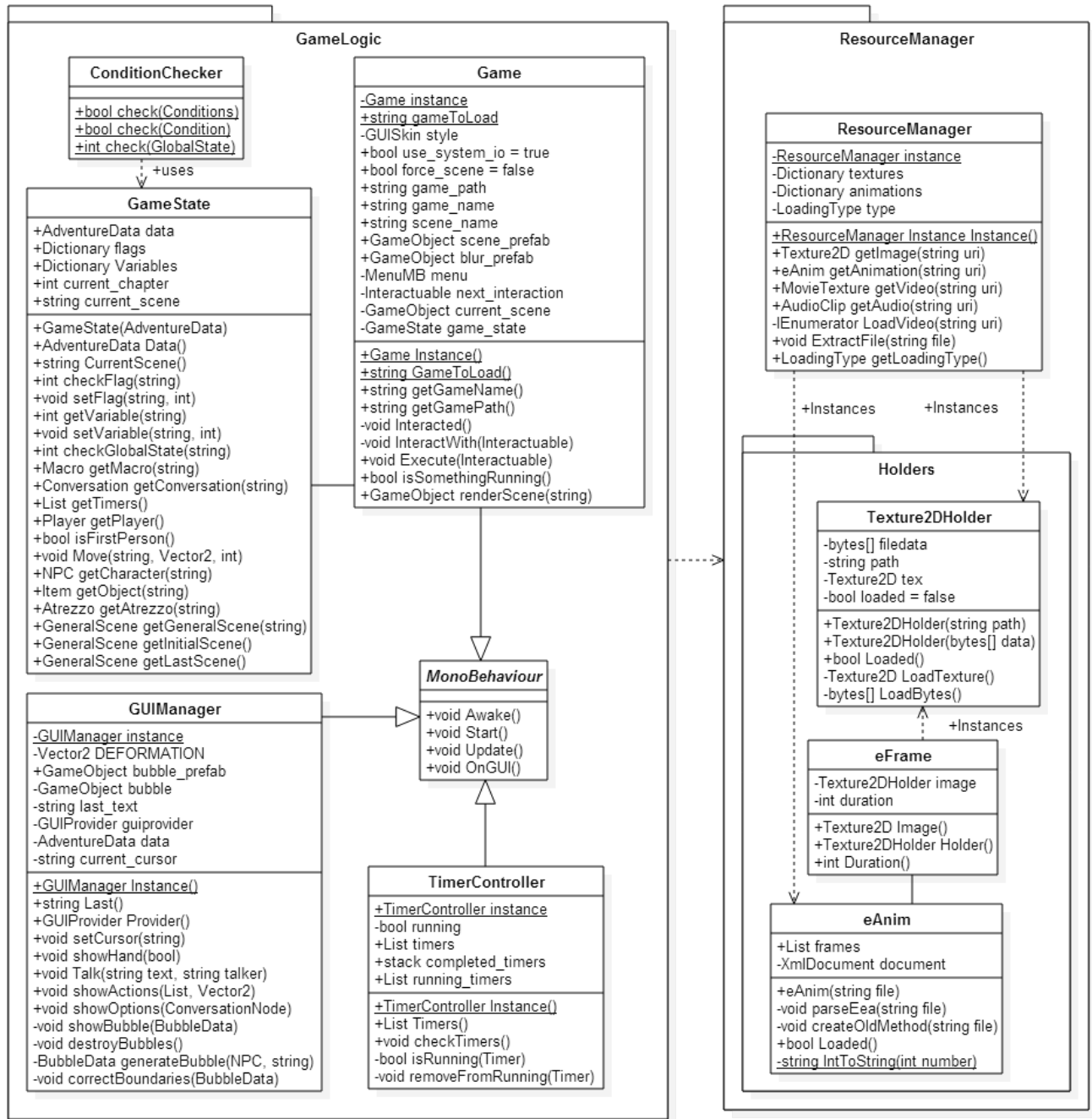
1. **Game:** La clase *Game* ha sido reducida para encargarse de tres tareas principales: En primer lugar, iniciar la carga y poner en funcionamiento el juego una vez esté cargado. En segundo lugar, controlar la interacción del usuario con los elementos del juego, ya sean elementos con representación, o elementos abstractos como secuencias. Y por último, ser la puerta de enlace que permite elegir qué escena se va a representar.
2. **GUIManager:** Este gestor se encarga de realizar la gestión de la interfaz que antes se realizaba en *Game*. Permite la emisión de burbujas de diálogo y el control sobre ellas, la posibilidad de mostrar una lista de opciones entre las que el usuario debe seleccionar una respuesta, y por último, la representación de acciones, como botones, y la gestión del menú contextual.
3. **GameState:** Aunque *GameState* no se considere un *Manager* en si mismo, porque, a diferencia de todos los demás, no es un Singleton, siempre sigue habiendo una única



Instancia, pues solo podemos tener un estado de juego, excepto cuando cargamos y guardamos partida. Maneja el estado del juego, que antes se realizaba en *Game*, y facilita funciones para acceder a objetos de la especificación del juego, como *Item* o *NPC*.

4. **TimerController:** Este gestor que se encarga de controlar los temporizadores que se utilizan en uAdventure para determinadas tareas, como, por ejemplo, hacer que un edificio se queme si no se ha evacuado a tiempo, es, tanto un *Singleton*, como un *Mono-Behaviour*, pues necesita de la función *Update()* para controlar que sus temporizadores no hayan saltado. Esta funcionalidad no estaba disponible en la anterior iteración.
5. **ResourceManager:** este gestor se encarga de facilitar un repositorio transparente a través del cual acceder a los recursos del juego. Añade funcionalidades adicionales que no estaban disponibles anteriormente, como la persistencia en memoria de recursos para agilizar los tiempos de carga, la carga de vídeos, así como la carga de archivos de sonido.

La representación de las clases que se explican en esta sección está disponible en la figura 4.2, junto a todas sus funciones. En este diagrama no están representados todos los elementos, pero si aquellos que han surgido de la evolución de las clases que se especificaron al comienzo de la sección. Este diagrama es bastante específico, pues contiene datos acerca de todas las funciones disponibles en cada uno de los elementos.



**Figura 4.2:** Diagrama de clases de los grandes gestores que controlan y proveen contenido para la ejecución del juego.

# Bibliografía

- [1] M. M. T. Loy. Observation of population inversion by optical adiabatic rapid passage. *Phys. Rev. Lett.*, 32:814–817, 1974.
- [2] W. D. Phillips, J. V. Prodan, and H. J. Metcalf. Laser cooling and electromagnetic trapping of neutral atoms. *J. Opt. Soc. Am. B*, 2:1751–1767, 1985.
- [3] J. Weiner, V. S. Bagnato, S. Zilio, and P. S. Julienne. Experiments and theory in cold and ultracold collisions. *Rev. Mod. Phys.*, 71:1–85, 1999.

# Apéndice A

## Title for This Appendix

## Apéndice B

### Title for This Appendix