UADVENTURE: DESARROLLO DEL INTÉRPRETE Y DE UN EMULADOR DE VIDEOJUEGOS DE EADVENTURE SOBRE UNITY3D

IVÁN JOSÉ PÉREZ COLADO

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

Trabajo Fin Máster en Ingeniería Informática

21 de Julio de 2016

Calificación: Sobresaliente 9.5

Director:

D. Baltasar Fernández Manjón

Autorización de difusión

Iván José Pérez Colado

21 de Julio de 2016

El abajo firmante, matriculado 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: "UADVENTURE: DESARROLLO DEL INTÉRPRETE Y DE UN EMULADOR DE VIDEOJUEGOS DE EADVENTURE SOBRE UNITY3D", realizado durante el curso académico 2015-2016 bajo la dirección de D. Baltasar Fernández Manjón, 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.

Fdo. Iván José Pérez Colado.

Resumen

Los videojuegos educativos, también conocidos como juegos serios, son una herramienta educacional muy poderosa, cuya utilización no está muy extendida en la educación. Estos Serious Games son costosos de producir, y son muy dependientes de los cambios tecnológicos, tanto en el Software como en el Hardware. Por ejemplo, multitud de Serious Games estaban producidos en Adobe Flash o Java, y hoy en día no pueden ser ejecutados en algunos de los dispositivos más nuevos. Uno de los pioneros de los videojuegos serios "Science Pirates: The Curse of Brownbeard", actualmente no está disponible porque no ha sido adaptado a los nuevos sistemas operativos. Por lo tanto, el ciclo de vida de los juegos serios debe ser simplificado para hacerlos una herramienta de confianza. En el equipo de desarrollo e-UCM se ha creado una herramienta de autoría de juegos serios basada en Java llamada eAdventure, así como multitud de juegos serios en colaboración con multitud de instituciones. Para lidiar con los problemas anteriormente identificados, y simplificar el proceso de creación y mantenimiento de juegos serios, y reutilizando la experiencia previa, se ha creado uAdventure. Este proyecto es un editor e intérprete construido sobre Unity3D, que permite la creación de videojuegos educativos sin requisitos de conocimientos de programación. Como uAdventure está construido sobre Unity3D, permite la exportación de videojuegos, de forma sencilla para múltiples plataformas, y los hace más resistentes a los cambios tecnológicos. A lo largo de esta memoria, se explica el proceso de generación del intérprete de videojuegos, así como la integración con el editor desarrollado por Piotr Marszal, en el que se realizan aportaciones, generando editores. Además, para realizar una labor de innovación, y dar soporte a los juegos cuyos desarrolladores no puedan invertir tiempo en transformar sus videojuegos al nuevo sistema de uAdventure, se ha desarrollado un emulador independiente capaz de importar y ejecutar juegos producidos con eAdventure en cualquier plataforma. Finalmente, para dar soporte y mejorar la parte de evaluación de los alumnos, se ha integrado RAGE en la infraestructura del proyecto, permitiendo el acceso a herramientas de Learning Analitics.

Palabras clave

eAdventure, Herramienta de autoría, Unity3D, Unity, Intérprete, Emulador, e-Learning, Videojuegos, Serious Games, Juegos Educativos, uAdventure, Ciclo de Vida, RAGE, Learning Analytics, xAPI.

Abstract

Educational games (aka serious games, SG) are a powerful educational content that are not extensively used in education yet. SG are costly to produce and they are very dependent of the technology changes in software or hardware. For example, many SG were produced in Adobe Flash or Java can not be run in some of the newer devices. One of the pioneer SG used in schools "Science Pirates: The Curse of Brownbeard" is currently not available because it is been adapted to new operating systems. Therefore we should simplify the full SG life cycle to make them a reliable educational content. Al the e-UCM research team we created a java based authoring tool called eAdventure (eA) and many SG in collaboration with many institutions. To deal with the previously identified problems and to simplify the creation and maintenance of SG reusing our previous experience and content we have created uAdventure (uA). uA is an SG editor built on top of Unity3D that allows for the creation of educational adventure games without requiring programming. uA has the same simplified graphical editor that eA and it is able to import previous games developed with eA. As uA is built on top of Unity3D it allows for a simple exportation of games for different platforms and make the created games more resilient to technological changes (as it is expected that Unity3D will cope with that complexity). Along this document, it's explained the development of the interpreter of eAdventure games, as well as the integration with the editor developed by Piotr Marszal, in which some contributions are made, developing new editors. Also, for a work of innovation, and to support the games whose developers can't invest the time to transform them to uAdventure's new system, it has been developed an standalone emulartor, able to import an run games produced with eAdventure on any platform or operating system. Finally, to support and improve the student assesstment part, RAGE has been integrated in the project infraestructure, allowing access to Learning Analitycs tools.

Keywords

eAdventure, Authoring Tool, Unity3D, Unity, Interpreter, Emulator, e-Learning, Videogames, Serious Games, uAdventure, Life cyrcle, RAGE, Learning Analytics, xAPI.

Agradecimientos

Quiero agradecer a mi director de proyecto, Baltasar Fernández Manjón, la oportunidad que me brindó en febrero para llevar a cabo este proyecto que comencé por cuenta propia, y que ha terminado siendo una experiencia fantástica que me ha abierto las puertas a un mundo que no podía imaginar.

Agradecer a Federico Peinado la comprensión y el apoyo que me ha prestado siempre. Ojalá pueda devolvértelo algún día.

Agradecer a todos mis compañeros del Aula 16 donde estoy disfrutando de algunos de los mejores días de mi vida. Sin vosotros habría terminado el proyecto mucho antes.

Agradecer a mi familia el cariño y apoyo incondicional que me han dado para llegar hasta aquí. Sin vosotros nada de esto habría sido posible.

Y por último, agradecer a Aylén todo lo que hace por mi. Gracias por cuidarme cuando creía que no podía dar más y hacerme creer que podía conseguirlo.

Para Aylén.

Nunca jamás podía haber imaginado lo increíble que mi vida de iba a volver desde el día en que te conocí.

Índice general

Índice de figuras

Capítulo 1

Introducción

Cada vez es más frecuente que dispositivos tecnológicos de uso cotidiano, como smartphones o tablets, remplacen 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. 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 el punto de partida de este proyecto y muestra la importancia de dar soporte al ciclo de vida de aquellos juegos que han dejado de ser mantenidos por sus desarrolladores ya que es conocimiento de gran valor para la comunidad educativa. Dicho juego se reconstruirá sobre Unity, motor de videojuegos muy popular, caracterizado por la amplia portabilidad

de los videojuegos creados con dicho motor, lo que no sólo facilita la adaptación a los nuevos dispositivos tecnológicos, sino que además, gracias a las frecuentes actualizaciones que recibe este motor, permitirá a largo plazo mantener actualizado el juego.

El planteamiento de la reconstrucción de dicho videojuego, lleva a la evolución de este proyecto en un sistema reutilizable, generando una herramienta capaz de interpretar el contenido de cualquier juego generado por eAdventure y permitir jugarlo. Por otra parte, le da al usuario la posibilidad de generar dicho juego en un paquete único ejecutable y compatible con múltiples plataformas y sistemas operativos.

A lo largo de este proyecto se genera una memoria que explica el proceso de cómo evolucionó 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 basada en clases, objetos, y en una línea de ejecución secuencial, a una arquitectura basada en clases-componentes, reactiva a eventos, y con un ciclo de vida de videojuego, en el que cada uno de sus elementos participa en su ejecución.

No obstante, y para el beneficio del mayor número de personas, las cuales no tienen por qué tener conocimientos de informática para poder disfrutar de juegos de eAdventure en cualquier plataforma, se produce un emulador de videojuegos de eAdventure que permite importar cualquier videojuego y jugarlo. Esta capacidad dará soporte al ciclo de vida de aquellos videojuegos cuyo ciclo de vida no pueda ser soportado por sus desarrolladores, contemplando así la mayor parte de casos que pueden darse con juegos de eAdventure.

Finalmente, una de las características menos desarrolladas de eAdventure es la capacidad de evaluar a los alumnos que jueguen a los juegos educativos. En eAdventure, este proceso se realiza mediante la generación de un registro a lo largo del juego, que está disponible al completar la sesión de juego. Estos informes deben mandarse al profesor al terminar para que este evalúe los resultados obtenidos y pueda decidir que medidas ha de tomar ante los

resultados obtenidos por los alumnos. Ante esto se plantea el uso de RAGE, un proyecto que, utilizando xAPI, genera gráficas y alertas que ayudan al profesor a identificar los alumnos que necesitan ayuda en tiempo real y utilizando una interfaz web para mostrar los datos. De esta manera se permite monitorizar a un mayor número de alumnos, así como de realizar enseñanza a distancia más fácilmente, pudiendo dar soporte a cursos online o moocs.

Todo este proceso se ve representado en la Figura 1.1, donde se parte de un juego de eAdventure, que es importado por uAdventure, y generado a múltiples plataformas, las cuales se comunican con RAGE a través de internet, y ayudan al profesor en la enseñanza.

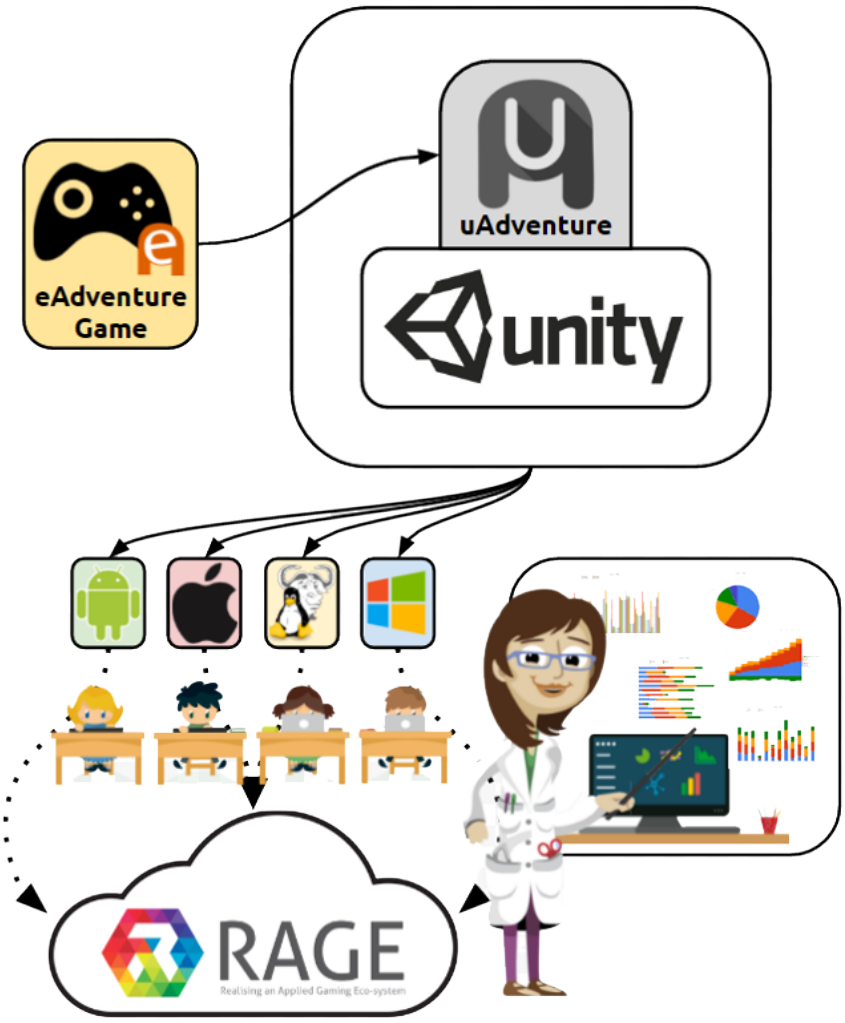

Figura 1.1: *Gráfico que presenta la introducción de uAdventure*

Capítulo 2

Introduction

It's getting more and more common that everyday technological devices, such as smartphones or tablets, replace personal computers, and not only in homes but also in schools, whith teachers who use that kind of devices as a learning tool. The eAdventure framework, among the games generated using this framework, run using the Java Virtual Machine, which is getting less support in these kind of devices. All those games generated with eAdventure are available for a less amount of people, and as they are mostly educational video games with valuable content for education, both at student level, and at professional level, with games developed for training, awareness and training of workers, it is important to provide support to all the games developed on eAdventure so they can be exported for different platforms like tablets or smartphones.

This is the case of the video game Checklist, created in and Adventure to train health proffessionals, emphasizing the importance of always performing Surgical Safety Checklist. This video game was developed by the eUCM research group and is only available for Windows, Mac OSX and Linux. Checklist is the start point of this project and shows the importance of supporting the life cycle of those games which are no longer mantained by their developers, because they contain valuable knowledge for education. This game will be rebuilt on Unity, a very popular video game engine characterized by their tools for exporting a video game to multiple platforms. This not only makes easier the adaptation to the newest devices, but also, thanks to the frequent updates that Unity gets those games will also be

updated for a long period.

The approach to the rebuilding of the game, leads the development of this project into a reusable system, creating a tool for reading the content of any game generated with eAdventure and allow to play it inside Unity. On the other hand, it gives the user the ability to generate the game in a standalone executable package compatible with multiple platforms and operating systems.

Throughout this project, a document will be generated that explains the process of how it evolved from the approach of a reconstruction of game and its many iterations of the design of architecture, and its subsequent integration with the project of Piotr Marzsal. Along this process, they appeared several problems, and in this document are described among to their resolution, facing a paradigm shift where eAdventure have a class-based architecture, with objects and a linear execution, and uAdventure must use a class-components based architecture, reactive to events, and with a running cycle of video game, in which each of its elements is involved in its execution.

However, and for the benefit of the most number of people, who doesn't necesary need to have computer skills to enjoy eAdventure games in every platform, a eAdventure video game emulator is produced that allows to import every game and let's the user play it. This ability supports those games whose developers can't support their life cycle, giving support to most of the eAdventure games.

Finally, one of the eAdventure least developed features is the ability to evaluate the students who play this serious games. On eAdventure this is done by generating a log throughout the game, which is available when the game session is completed. This log must be sended to the teacher so they can be corrected and evaluated. When they has been corrected the teacher can decide what actions to take with those results. In response, in this project is proposed the use of RAGE, a project that using xAPI, generate graphics and alerts that helps the teacher to identify the students who need help in realtime by using a web UI. That way teachers are allow to monitorize a highter number of students, as well as

supporting remote learning like online courses and moocs.

This proccess is shown on the Figure 1.1, where it starts with an eAdventure game, which is imported with uAdventure, and generated into multiple platforms, which also communicate with RAGE using the internet, and helps the teachers showing them learning analitycs.

Capítulo 3

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 [2], engloba aquellas aplicaciones y servicios que, tomando como base las TIC, se orientan a facilitar el proceso de enseñanza-aprendizaje. 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 [40], 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 [40].

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

Capítulo 4

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.

4.1. eAdventure

eAdventure es una herramienta de autoría de aventuras gráficas conversacionales, que permite aplicar un enfoque educativo a los videojuegos desarrollados con dicha herramienta, permitiendo generar un perfil de evaluación de los alumnos. Esta herramienta es libre y gratuita, y está desarrollada por el equipo e-UCM de la Facultad de Informática de la Universidad Complutense de Madrid.

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:

1. **Dificultad para equilibrar la parte divertida y la parte educativa del video-**

juego. 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 [10] 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 [27], The Day of The Tentacle [11] o, la franquicia española, Runaway [9]. 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 [16], explicado en el apartado 6.2.1, Wintermute Engine [33], explicado en el apartado 6.2.3, u Open SLUDGE [35], explicado en el apartado 6.2.2. Sin embargo, todo este tipo de herramientas, pese a ayudar mucho a producir videojuegos de este género, no aportan funcionalidades específicas para los videojuegos educativos. De esta manera, eAdventure, aporta un enfoque específico orientado a los videojuegos educativos.

2. **Los costes de producción**. El desarrollo de un videojuego es un proceso costoso, llegando a presupuestarse en varios millones de dólares. En la última década, han surgido varios proyectos de videojuegos educativos cuyo presupuesto supera los cientos de miles de euros [21]. 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.

3. **Problemas de distribución**. Finalmente, una vez que el videojuego está desarrollado ha de distribuirse para que las personas puedan beneficiarse del mismo. Es frecuente encontrar videojuegos que 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 tengan el efecto deseado. Para solucionar este problema, eAdventure decide utilizar la Máquina Virtual Java [18], la cual permite la ejecución de un software en múltiples plataformas siempre que estas tengan instaladas dicha máquina virtual.

   Sin embargo, pese a que Java ha gozado de mucha popularidad a lo largo de los años, 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 [32], la cual está siendo sustituida actualmente por ART [17], y que reemplaza completamente a la maquina virtual Java para la ejecución de programas desarrollados en dicho lenguaje.

Es por ello que el proyecto desarrollado que se explica en esta memoria 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.

4.2. El videojuego 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 de-

tectar 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 [20].

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 [12].

Este juego está desarrollado utilizando eAdventure, y por consiguiente es del género Aventura Gráfica, más concretamente una Aventura Gráfica en Primera Persona, en la que el jugador no maneja a un personaje dentro de un juego, sino que ocupa el lugar del protagonista. Asimismo, el jugador elije uno de los diferentes roles que se ofrecen y que participan dentro de un quirófano, pudiendo ser Anestesista, Cirujano y Enfermero. Tras esto se plantean tres situaciones diferentes en una operación: 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. Para añadir más complejidad al juego 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

4.3. 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 [37].

En Unity, cada objeto que toma partido dentro de la escena, está constituido por un GameObject y todas aquellas componentes asociadas, extendiendo a la clase MonoBehabiour, así como una componente Transform que le permite orientarse y deformarse en la escena, y una componente Renderer que le ayuda a representarse.

Capítulo 5

Metodología de desarrollo

Una parte crucial de todo desarrollo de un 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 Modelo Lineal Secuencial o un desarrollo utilizando el Modelo Lineal en Cascada, o metodologías basadas en la iteración, como el modelo en espiral de Boehm, o un desarrollo incremetal. Sin embargo, estas metodologías, en general, están pensadas para organizar a grandes equipos jerarquizados en procesos de desarrollo de software 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, mediante la generación de un estado del arte y la, 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 híbrida, que mezcla bases de las metodologías de software tracidionales, pero con influencias de los doce principios del manifiesto ágil [3], con 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:

1. **Estudio y primer prototipo**: en esta primera iteración se estudian y analizan los videojuegos de eAdventure con el objetivo de interpretar dichos paquetes e intentar representarlos, aprendiendo de los resultados obtenidos.

2. **Mejor diseño de la aplicación, nuevas interfaces, refactorización de clases**: mediante los resultados obtenidos en la primera iteración, se realiza un anlálisis y re-estructuración del proyecto. Por otra parte se añaden mejoras gráficas visuales dentro de la aplicación. Finalmente, en esta iteración se produce la integración del proyecto de Piotr Marszal, explicado en el capítulo 12, junto al intérprete de juegos, transformando ambos proyectos en una única herramienta capaz de generar juegos dentro de Unity. Como parte del proceso de integración, se adaptan algunas de las clases, reconstruyendo algunos de los elementos para que trabajen en múltiples condiciones.

3. **Generar un emulador capaz de importar juegos de eAdventure, así como nuevos editores**: este permite al usuario jugar a cualquier juego de eAdventure independientemente de la plataforma y el soporte., así como generar nuevos editores para uAdventure, como los editores de Efectos y Conversaciones, así como editores que permiten hacer mas sencilla la labor de evaluación y determinación del progreso en RAGE desde Unity.

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

Para alinear el proyecto con los objetivos, y con mayor frecuencia en la última iteración del desarrollo, se realizaban reuniones periódicas de revisión, generación de tareas para la próxima reunión. Dichas reuniones suceden a lo largo del proyecto 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 6

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.

6.1. 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 posicional 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 se distribuyen. Desde los primeros motores de videojuegos, que en un comienzo tomaron el nombre de los videojuegos que desarrollaban, como DOOM [26] o Quake [43], 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 hallan Unreal Engine [39], explicado en el apartado 6.1.1, Unity [38], explicado en el apartado 4.3, o CryEngine [15]. 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. Por ejemplo, Unreal Engine 4 se distribuye con una licencia que establece un mínimo de beneficios de 3000€para comenzar a solicitar un

5 % de ellos. Esta licencia permite desarrollar y comercializar un videojuego y únicamente cuando este comienza a dar beneficios y es rentable se paga por el uso del motor.

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,

6.1.1. 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 [39]. El motor ha ido evolucionando desarrollando videojuegos como Unreal Tournament [13], Turok [23], America's Army [30] -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 ejercito de los Estados Unidos-, Gears of War [24], la saga BioShock [25], o la saga Mass Effect [5].

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 6.1. Esto, no obstante, no implica que todo lo generado con Unreal Engine tenga una calidad gráfica excenelente; únicamente hace más sencillo este proceso.

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

Figura 6.1: *London Apartment, una demo técnica desarrollada con Unreal Engine 4.*

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 el desarrollo de uAdventure se presentan 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, los dos integrantes del equipo de desarrollo tienen experiencia en el uso de Unity, por lo que utilizando Unity se ahorra el complejo proceso de aprendizaje de un nuevo motor de videojuegos.

6.1.2. GameMaker: Studio

Este motor de videojuegos, creado por el profesor Mark Overmans, y posteriormente desarrollado por la empresa YoYo Games, con fecha de lanzamiento inicial el 15 de noviembre de 1999, es un motor de desarrollo de videojuegos, basado en el desarrollo rápido de aplicaciones [14]. Este motor es gratuito, aunque también se puede encontrar una versión comercial que amplia las funcionalidades del editor [14].

Este motor está especialmente diseñado para permitir al usuario desarrollar un juego

sin la necesidad de aprender un lenguaje de programación. No obstante, contiene su propio lenguaje de *scripting* interpretado llamado *Game Maker Language*.

Pese a que se pueden generar videojuegos tridimensionales, su diseño está enfocado a la generación de juegos en dos dimensiones. E incluye características como el manejo de recursos, entre los que se encuentran gráficos, sonidos, fondos, etc... los cuales se asignan a objetos, el manejo de Eventos, como la pulsación de una tecla, o una jerarquía de objetos sobre los que se actúa en el videojuego.

Como características destacadas de este motor, encontramos que permite al programador añadir elementos a la escena utilizando mecanismos de *Drag and Drop*. Así como la posibilidad de generar videojuegos en formato ".exe"para el sistema operativo Windows, así como la posibilidad de generar un archivo ".apk", instalable en Android.

6.2. 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 [28] para videojuegos del género *Role Playing Game* (RPG), GameGuru [7] para videojuegos del género *First Person Shooter* (FPS) o Quintus [19] 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 de 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.

6.2.1. Adventure Game Studio

Dentro de esto de editores, encontramos a el más conocido de todos ellos, Adventure Game Studio [16], 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 [29].

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 6.2 se puede observar la vista principal del editor, con una escena en la parte del centro-izquierda. En la parte superior derecha se ve la lista de escenas disponibles,

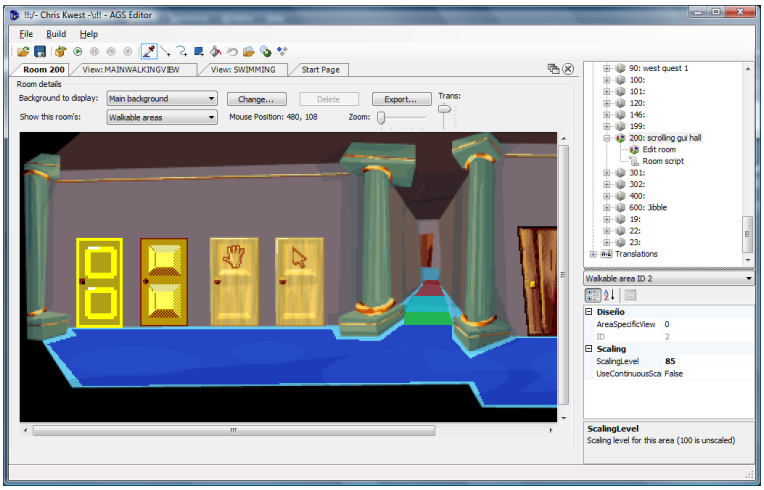

Figura 6.2: *Una escena en Adventure Game Studio*

y cuando estas se abren, se añaden en forma de pestañas al editor.

6.2.2. 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 interactuables; 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.

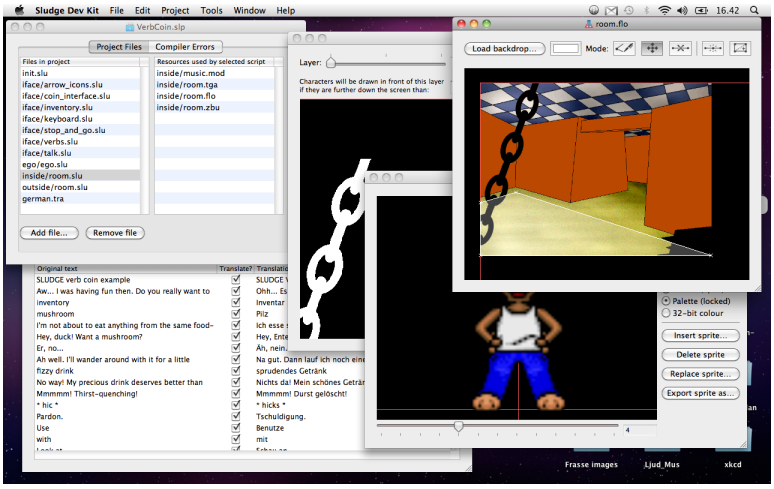

Figura 6.3: *Multitud de ventanas de Open SLUDGE ejecutándose en OSX*

En la Figura 6.3 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 fóndo, un diálogo que está siendo traducido del Inglés al Alemán.

6.2.3. WinterMute Engine

Por último encontramos WinterMute Engine [33]. 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 posilibidad 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.

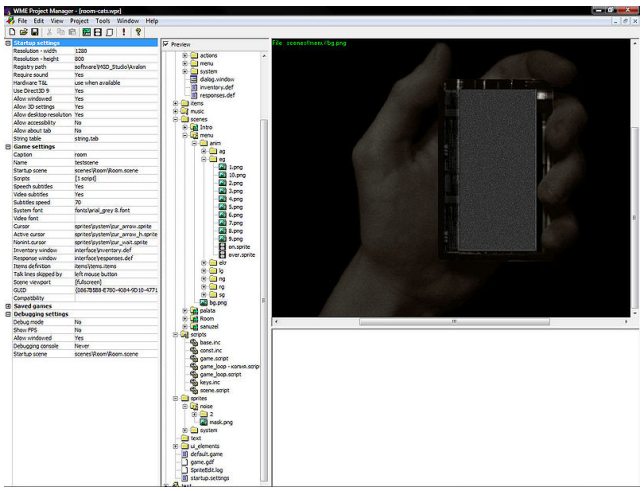

Figura 6.4: *Gestor de proyectos de Wintermute Engine*

En la Figura 6.4 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 coluna de la derecha es una previsualización del archivo seleccionado.

6.3. 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[1], 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 [41].

[1]Disponible en el repositorio de GitHub: https://github.com/e-ucm/eadventure-legacy-android

Figura 6.5: *El videojuego Fire Protocol mostrando la habilidad de hacer Zoom con una lupa e identificar elementos.*

El proyecto eAdventure Android añadía características que facilitaban el uso de juegos 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 6.5. 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 Figura 6.6.

Figura 6.6: *El videojuego Chocolate Factory mostrando el menú contextual.*

Esta funcionalidad de emulador de eAdventure en Android es la que se consigue obtener 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 Maquina 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 [41], 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 llamo 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 6.7. 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" en 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.

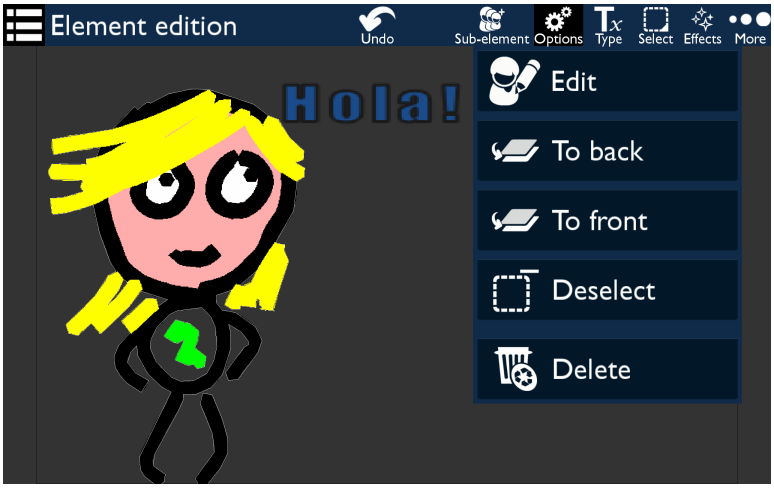

Figura 6.7: *Primera versión de Mokap, llamada eAdventure Editor en Android.*

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 6.8 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,

Figura 6.8: *Última versión de Mokap, mostrando una escena compleja compuesta por múltiples elementos compuestos.*

mediante una secuencia de imágenes. Sin embargo, el robot de la derecha utiliza Spine [34], animación basada en esqueleto. Las animaciones generadas con Spine son mucho más ligeras, 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, Mokap, 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.

Capítulo 7

Objetivos

Una vez completada la introducción a las necesidades del proyecto, presentadas en el capítulo anterior, mediante la abstracción de dicha introducción, y de esas necesidades, se genera un listado de objetivos que completar para satisfacer las necesidades del proyecto. Este listado presentará aquellos puntos que deberán abordarse durante el desarrollo del mismo para completar con éxito el proyecto.

Los objetivos identificados son los siguientes:

1. Hacer funcionar el videojuego Checklist en Unity3D: En primer lugar se plantea el objetivo con el que nació este proyecto. No se especifica la manera a través de la cual se debe hacer funcionar dicho juego, únicamente se especifica que debe funcionar, incluyendo aquellas caracterísiticas que, sin afectar demasiado a la estética, permitan su jugabilidad.

2. Hacer funcionar el mayor número de juegos desarrollados en eAdventure por CATEDU [8]: El segundo objetivo consiste en, generalizar el objetivo uno para que sea aplicable a los videojuegos desarollados por el Centro Aragonés de Tecnologías para la Educación. La importancia de este objetivo reside en que, los videojuegos desarrollados por dicho centro fueron desarrollados por personal cualificado con ayuda y soporte de profesionales especializados en los sectores para los que se desarrollaron dichos juegos.

3. Hacer funcionar el mayor número de juegos posibles de eAdventure en Unity3D: El

tercer objetivo planteado consiste en poder generalizar el proceso establecido en el primer y segundo objetivos para que sea aplicable a cualquier juego de eAdventure. Esto implica implementar la mayoría de características disponibles en eAdventure, como son el tratamiento de jugador en primera persona o el soporte a trayectorias, las cuales, son características complejas de implementar.

4. Generar un diseño y arquitectura de aplicación de calidad: Para alcanzar este objetivo, se deberá elegir y aplicar una metodología de desarrollo que permita <. En estas iteraciones se desarrollarán protopitos, los cuales serán refactorizados y rediseñados, generando diagramas de clases de la arquitectura del proyecto que se desea implementar, y que represente las clases desarrolladas.

5. Integrar de este intérprete de juegos con el proyecto de Piotr Marszal, el cual consiste en la reconstrucción del editor de juegos dentro de Unity. Este proyecto será la parte encargada de ejecutar los juegos que sean creados, desarrollados, o abiertos con dicho editor. Dentro de este objetivo se encuentra implícita la labor de realizar tareas de unificación de clases, como el modelo de datos o las herramientas para cargar el mismo.

6. Implementar una serie de facilidades que mejoren la interacción en el ámbito táctil. Este objetivo se produce por la habilidad del motor de videojuegos Unity para generar videojuegos múltiplataforma, entre las que se encuentran las plataformas táctiles como son Android e iOS.

7. Dar la capacidad a los no desarrolladores, de poder jugar a videojuegos generados con eAdventure en cualquier plataforma, y sin necesidad de tener Unity, así como de beneficiarse de las características de uAdventure.

8. Mejorar la capacidad de evaluar existente en eAdventure, y permitir, de alguna manera, la posibilidad de evaluar a distancia, además de facilitar al profesor incrementar, sin temor a poder dejar de dar soporte individual, la cantidad de alumnos a los que enseñar utilizando videojuegos educativos.

Con estos objetivos planteados, el desarrollo del proyecto se enfocará para poder satisfacerlos en mayor o menos medida. Los resultados obtenidos tras el proyecto se presentan en las conclusiones de la memoria, presentadas en el capítulo 13.

Capítulo 8

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[1], 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 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 se completa un ciclo de juego. 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 la 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 por si mismo producido por eAdventure. Dicho paquete es un archivo ".jar". Mediante la extracción de dicho paquete, no sólo se obtienen los recursos gráficos del juego.

[1]Enlace al repositorio: https://sourceforge.net/projects/e-adventure/files/games/checklist/

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 lineas

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 mayor complejidad técnica, como es la interpretación de un archivo de texto, reducirá costes a la larga gracias a la reutilización de dicha funcionalidad.

- Pese a que existen herramientas para la localización de programas mediante el análisis del código en busca de fragmentos de texto para traducir [42], 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 fun-

damentos 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 [1]. 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 [31]; 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.

Tras haber analizado todos los factores presentados, se tomó la decisión de realizar un intérprete de juegos de eAdventure, capaz de analizar los elementos contenidos en los paquetes autocontenidos de eAdventure. Con esta investigación realizada, y las decisiones más básicas tomadas se comienza con el diseño e implementación del proyecto.

Capítulo 9

Primera Iteración: Prototipo inicial

9.1. 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 sí mismo.

En esencia, 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 9.1 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 él. 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 *MonoBehabiour* 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

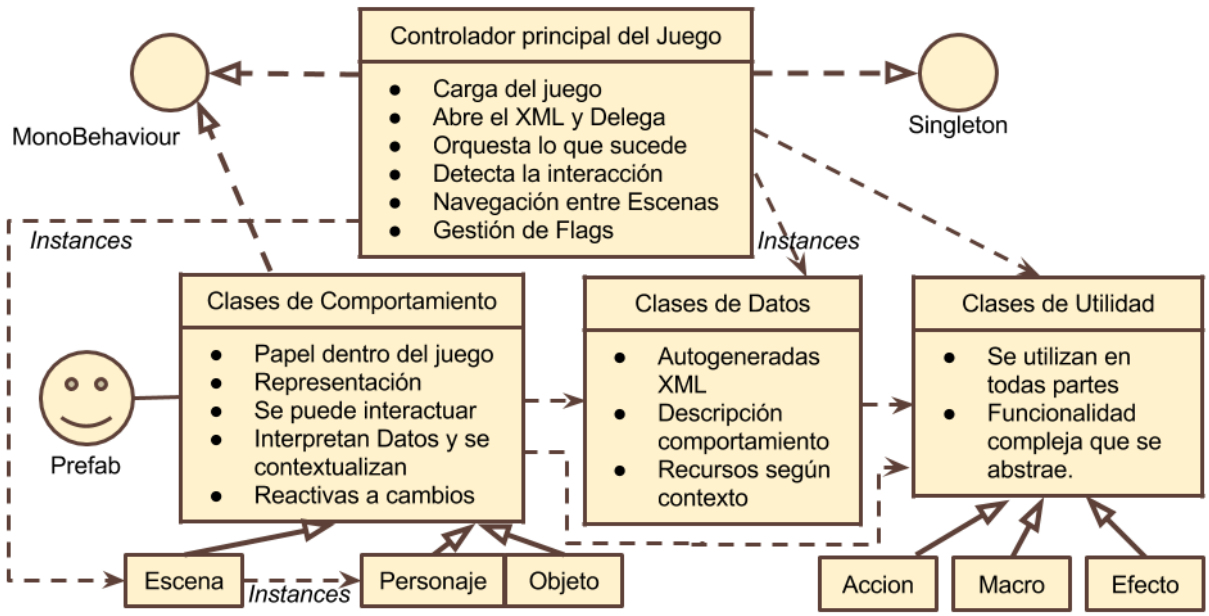

Figura 9.1: *Diagrama de clases simplificado de la arquitectura de clases de la primera iteración del proyecto.*

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 9.1 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.

9.1.1. 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.

9.1.2. 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, siento meros contenedores de datos.

9.1.3. 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 atrezo y todos aquellos elementos que tienen una representación virtual en el videojuego.

Cada una de estas clases heredará de la clase de Unity *MonoBehabiour*, que indica, como se explica en el apartado 9.1, 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 recibirá estímulos por parte del controlador principal del juego cuando el jugador interactue con ellas. Este paso intermedio es necesario ya que hay momentos en los que el controlado principal debe evitar la interacción del jugador con dichos elementos por encontrarse en mitad de un diálogo. Con este estímulo, la clase de comportamiento gestiona las diferentes interacciones que tenga el usuario con si mismo.

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

9.2. Detalles de la implementación

Existen algunas cuestiones cruciales y generales acerca de cómo se realizó la implementación de diversas mecánicas que eran necesarias para que el juego funcionase.

Algunas de estas decisiones cruciales son, por ejemplo, la decisión de *reRenderizar*[1] la escena junto con todos sus elementos cada vez que un *Flag*[2] 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 decisión se tomó debido a que, pese a que existía la opción de implementar el patrón 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

[1]Renderizar es el proceso de generar una imagen, mediante el cálculo de la iluminación, a partir de un modelo 3D. En este caso se refiere a regenerar la representación de la escena.

[2]Una *Flag*, bandera en Inglés, 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 *SecuenceManager* 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 *SecuenceManager*, y construir una pila de secuencias, se vió 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 un apartado posterior se detallan en profundidad los detalles de la implementación de las clases *Conversation* y *Effect*, dos clases que implementan la interfaz *Secuence*, y que permiten ser ejecutadas por si mismas.

9.2.1. 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*[3] 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*[4] 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*,

[3] 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

[4] 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.

9.2.2. 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

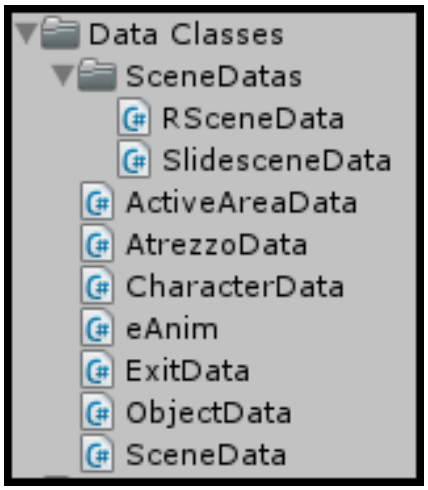

Figura 9.2: *Vista de carpeta de las clases de datos implementadas en la primera iteración.*

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 representarese 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 dos 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 área activa 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 cuyos nombre se presentan de la siguiente manera en secuencia: “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

".eaa" que son mucho más sencillos de recorrer y contienen más información acerca de cada fotograma de la animación.

9.2.3. 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 9.3: *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 *MonoBehabiour* por lo que se asignan a objetos dentro de la escena, y luego estos objetos se guardan en forma de *Prefabs*, cada una a uno individualmente. Estos *Prefabs* suelen ser muy sencillos, en su mayoría constituidos únicamente por un *Quad*[5], al cual se le cambia la textura con la propiedad del material *this.GetComponent<Renderer>().material.mainTexture*, y posteriormente, mediante la información de dicha textura, se aplican modificaciones escalando el Quad.

[5]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 funcion *ChangeFrame()*.

Esta función *ChangeFrame()* accede al *renderer*[6] 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.

[6]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*: *OnMouseEnter()* y *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* es la base que construye las diferentes "pantallas" que se mostraran al jugador a lo largo de la aventura. 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 *MonoBehabiour* 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 *Interactuable*, 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.

9.3. 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 extraidas 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.

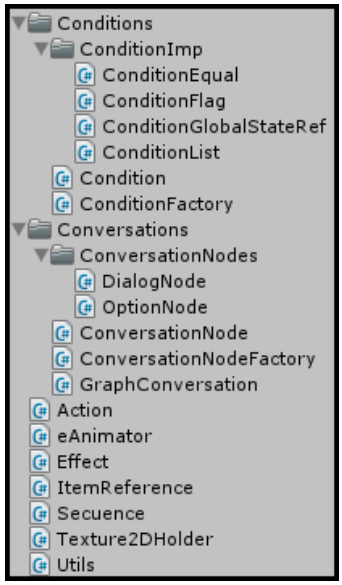

Figura 9.4: *Vista de carpeta de las clases de utilidad implementadas en la primera iteración.*

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.

9.3.1. 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 Unity 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*[7] 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;
}
```

[7]Una Texture2D es la clase que se utiliza para guardar texturas bidimensionales en Unity3D, como imágenes, fotografias, 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.

9.3.2. 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 si 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 *Interactuable*, 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. El aspecto más llamativo de *DialogNodeFactory* y de *ConditionFactory* es que utilizan *System.Linq*[8] 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 inferfaces *DialogNode* y *Condition*. Esto se ve representado en el código que se presenta a continuación:

[8]El espacio de nombres System.Linq proporciona clases e interfaces que admiten consultas que utilizan Language-integrated query (LINQ)

```
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.

9.3.3. 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 funcion *OnGUI()*.

Capítulo 10

Segunda Iteración: Versión final e Integración

Como se explica en el el apartado 5, en la que se expone la metodología de desarrollo utilizada para el proyecto, este sufre de tres grandes iteraciones a lo largo de su desarrollo. En este apartado 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 al capítulo 9, explicando los cambios en diseño y comportamiento.

10.1. 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.

Este cambio en la arquitectura se produce tras realizar un análisis de los resultados obtenidos en la primera iteración, debido al objetivo de generar un diseño de calidad, y la necesidad de preparar el proyecto para la posterior integración con la parte de edición de

uAdventure desarrollada por Piotr Marszal.

En esta arquitectura encontramos tres paquetes importantes que engloban la funcionalidad necesaria para la ejecución del proyecto. Están representados en la Figura 10.1 Estos tres paquetes son:

- **Core**: *Core* contiene el núcleo de eAdventure portado a Unity. Este paquete está formado, 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 nucleo 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 de una ruta donde hay un juego descomprimido, hasta la generación de un entorno gráfico interactivo que permite al usuario jugar al juego. Contiene tres 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.

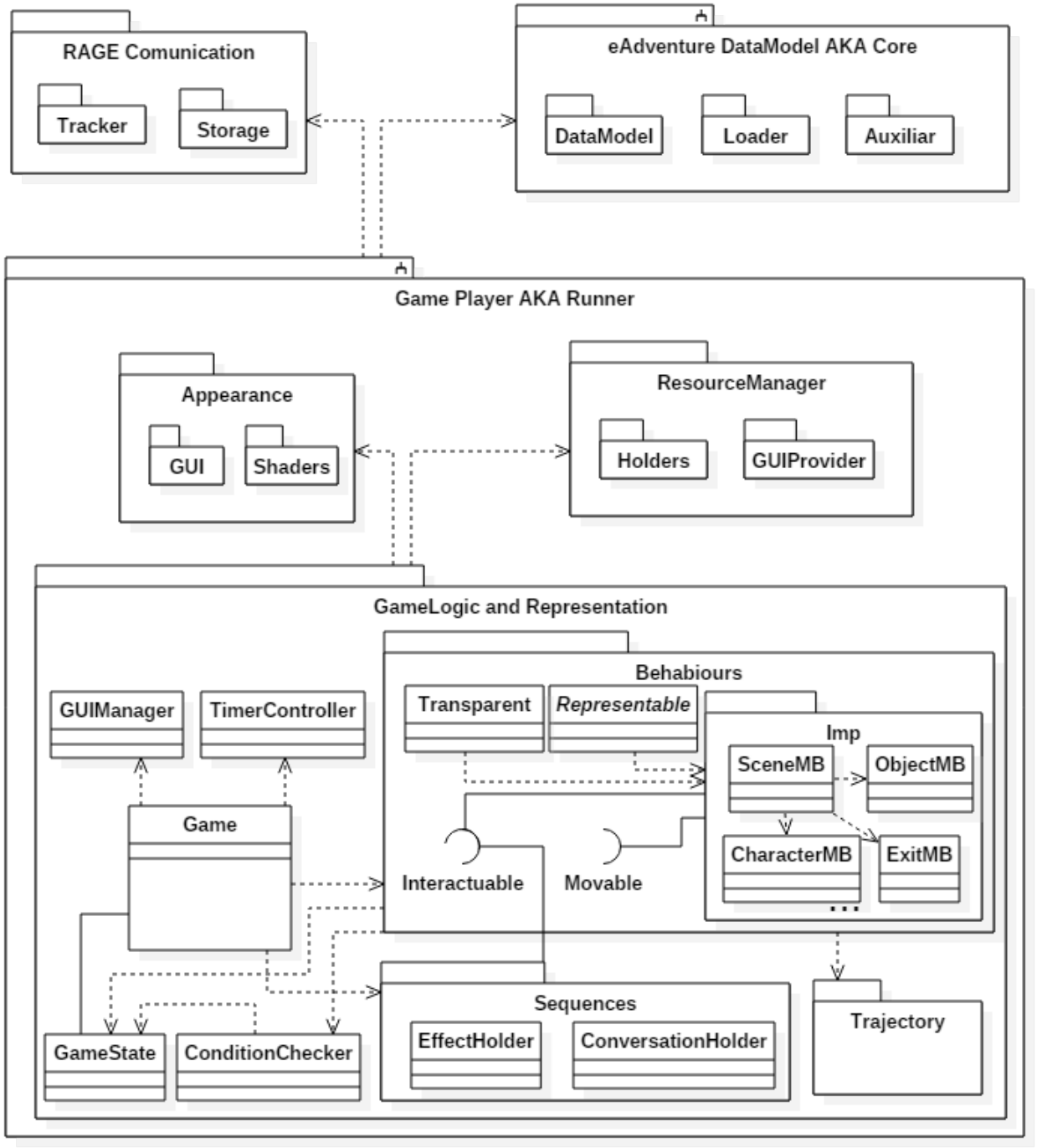

Figura 10.1: *Arquitectura del sistema, a nivel de paquetes de la versión final del proyecto*

10.2. El núcleo de Ejecución: Runner

El núcleo de ejecución, también conocido como *Runner*, está dividido en tres 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 el apartado 9.2.1, o bien en alguna de las clases de utilidad, explicadas en el apartado 9.3.

De esta manera conseguimos que la antigua clase controladora del juego, *Game*, y las Clases de Utilidad, ahora no tengan, que realizar tantas tareas como realizaban 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 *MonoBehaviour*, 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 este apartado está disponible en la Figura 10.2, 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 del apartado. Este diagrama es bastante específico, pues contiene datos acerca de todas las funciones disponibles en cada uno de los elementos.

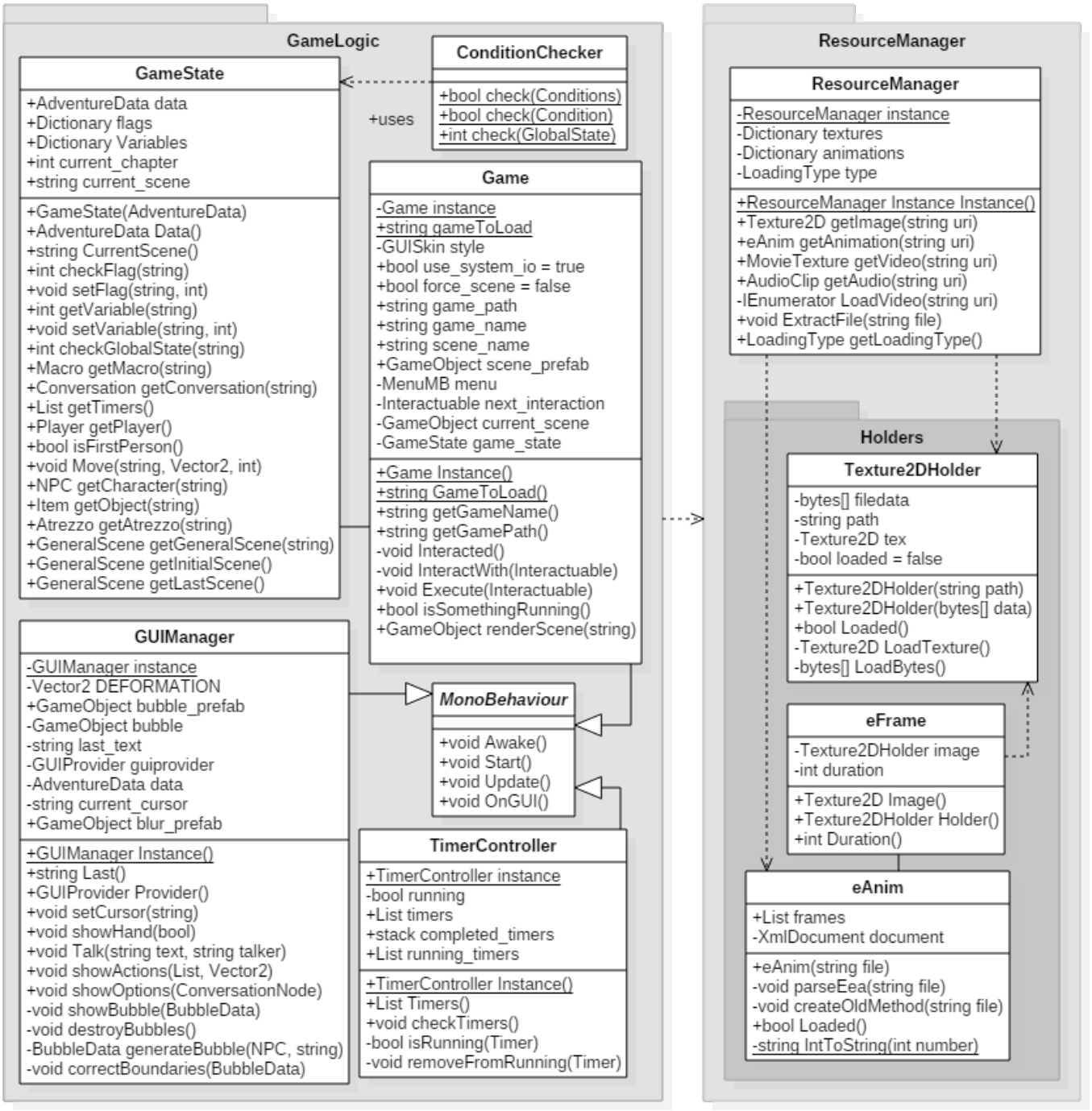

Figura 10.2: *Diagrama de clases de los grandes gestores que controlan y proveen contenido para la ejecución del juego.*

10.2.1. El estado del juego: GameState

Antes de explicar la clase *Game*, es importante explicar la clase que realiza de nexo de unión entre eAdventure y uAdventure, pues esta tiene dos tareas principales: controlar el estado del juego, sus variables, las *Flags* que definen hitos; y la tarea de facilitar el acceso a elementos del juego como los personajes que hay en el capítulo que está siendo jugado en ese momento, como la escena inicial o final de dicho capítulo.

En esencia *GameState* provee un acceso transparente a la especificación del juego en ese momento. Si le pides un personaje, te va a dar la representación de ese personaje en el capítulo que estés jugando. Es importante la relación de asociación direccional que se ve en la figura 10.3 entre *Game* y *GameState*, pues la clase controladora del juego necesita un estado del juego para funcionar, y sin ella, el juego no puede ejecutarse.

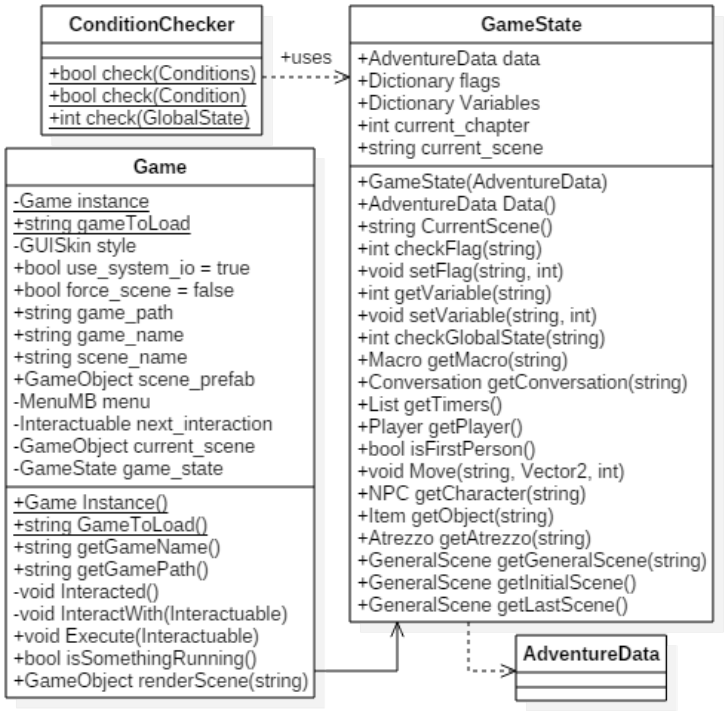

Figura 10.3: *Diagrama de clases de GameState, junto a Game, ConditionChecker y AdventureData*

El validador de Condiciones: ConditionChecker

Adicionalmente, se muestra en la figura 10.3 a una pequeña clase estática llamada *ConditionChecker*, que se encarga de facilitar la validación de condiciones y estados globales de eAdventure, estas condiciones pueden estar compuestas por multitud de comprobaciones de *Flags* o Variables dentro del *GameState*, por lo que este validador facilita y simplifica mucho el acceso a *GameState*.

10.2.2. El Controlador del Juego: Game

La clase *Game*, ha evolucionado de la clase explicada en el apartado 9.2.1. Como se ha mencionado multitud de veces a lo largo del apartado 10.1, esta clase *Game* ha evolucionado para delegar comportamientos en los 5 grandes gestores. Sin embargo, la funcionalidad que *Game* aporta en el ciclo de vida y ejecución del juego sigue siendo de vital importancia.

Esta clase *Game* ahora se encarga de 3 tareas: iniciar la carga y poner en funcionamiento el juego una vez esté cargado; controlar la interacción del usuario con los elementos del juego, así como con efectos y conversaciones; y ser la puerta de enlace que permite elegir qué escena se va a representar. No obstante estas tres tareas son vitales, pues en esencia, un juego sólo se puede jugar si se trata esta funcionalidad.

Como la clase *Game* es la puerta de enlace que transforma de una clase *AventureData* a algo jugable, junto a la explicación de *Game*, como subapartados, se presentarán los comportamientos encargados de enrolarse como los diferentes elementos que forman las escenas de eAdventure.

Uno de los cambios más notorios en todo el núcleo de representación es la desaparición de las Clases de Datos explicadas en el apartado 9.2.2. Esto es debido a la incorporación del modelo de datos original portado directamente de eAdventure.

En la figura 10.4 se muestra un diagrama de clases que explica las relaciones de las clases que participan en esta representación del juego. Aunque *SceneMB* tenga una relación de dependencia con *Game*, la escena únicamente necesita acceder a *GameState*, para obtener las

especificaciones de los elementos que va a instanciar. *SceneMB* a su vez mantiene relación de dependencia con *PlayerMB*, *CharacterMB*, *ObjectMB*, etc... debido a que *SceneMB* instancia todas estas clases como hijas suyas.

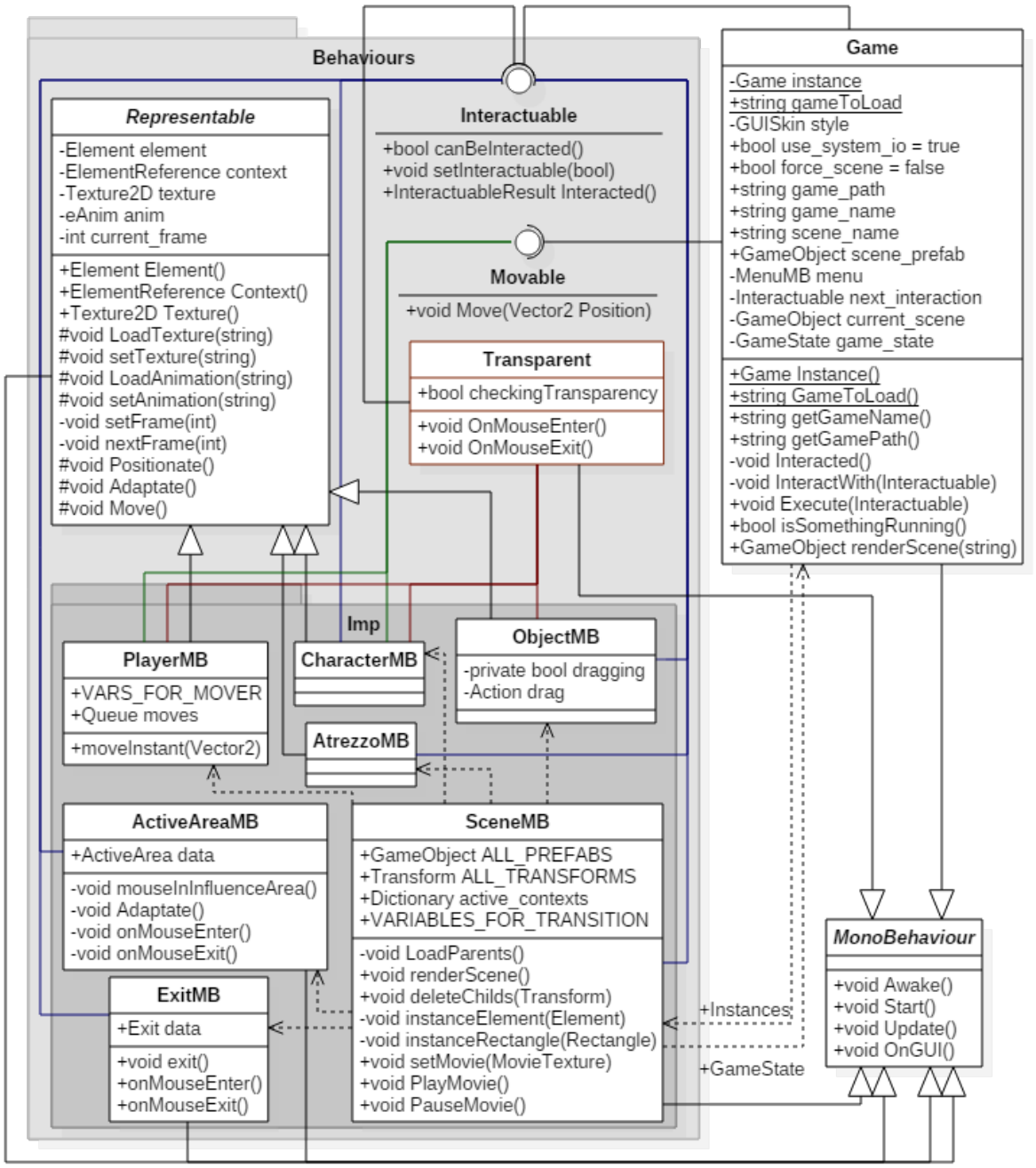

Figura 10.4: *Diagrama de clases de Game, con todo el nucleo de representación*

Representable, Interactuable, Movable y Transparent

Este grupo está compuesto por dos clases y dos interfaces, y permiten a la implementación de los comportamientos facilitar la definición de lo que pueden y no pueden hacer. Las dos interfaces son *Interactuable* y *Movable*. Tras esto encontramos *Representable* como clase abstracta que ayuda a la representación de todos los elementos que utilizan recursos, facilita la relación con el *ResourceManager*, y, al extender la clase *MonoBehaviour*, permite a todos los elementos que la extienden ser componentes a su vez. Por ultimo tenemos la clase *Transparent* que, facilita la interacción con elementos que, por motivos de representación en su imagen, no ocupan toda la imagen, teniendo transparencia.

En primer lugar, hablando de *Interactuable*, es la interfaz que permite a *Game* la interacción entre el usuario y los elementos de la escena. *Game*, cuando detecta una pulsación de ratón, lanza un rayo desde dicha posición, en dirección en la que mira la cámara, obtiene una lista de objetos, y, si implementan la interfaz *Interactuable*, les notifica que se ha realizado una interacción con ellos. Estos tienen tres opciones que responder: que no van a hacer nada con la interacción, que hacen algo con ella, o que hacen algo y además esperan más por parte del usuario. En el momento en que *Game* detecta alguna que hace algo, deja de notificar a los demás.

En segundo lugar, hablando de *Movable*, es una interfaz sencilla que deben implementar todos aquellos elementos que deseen moverse. Y que únicamente tiene un método *Move()*

En tercer lugar, hablando de *Representable*, encontramos la clase abstracta más compleja, pues permite a los elementos anteriormente citados representarse. Por si misma no realiza ninguna tarea, necesita que la clase que la extienda le indique lo que necesita. Por ejemplo, un objeto de Atrezzo, en su función *Start()* realiza lo siguiente:

```
void Start()
{
   base.Start ();
   base.setTexture(Atrezzo.RESOURCE_TYPE_IMAGE);
   base.Positionate ();
}
```

Además de esto, *Representable*, se encarga de transformar esa línea de especificación de recurso *Atrezzo.RESOURCE_ TYPE_ IMAGE* en una *Texture2D* mediante el uso del *ResourceManager*. Asimismo, esta clase *Representable* tiene también facilidades para el uso de animaciones y su actualización en fotogramas.

Finalmente, hablando de la clase *Transparent* es una componente que se añade a los objetos de la escena, y tiene una relación de dependencia con ellos, ya que, sin esta clase, nunca se habilitarían dichos objetos para permitir la interacción con ellos. En *Transparent* se analizan los píxeles de la textura activa en *Representable*, y, si el píxel sobre el cual se tiene el ratón no es transparente, es decir, su componente *Alpha* de color es mayor que 0, se accede a *Interactuable* y se establece como que se puede realizar interacción con dicho elemento.

Estos cuatro elementos de los que se componen los objetos de la escena están representados en la Figura 10.5, con sus relaciones.

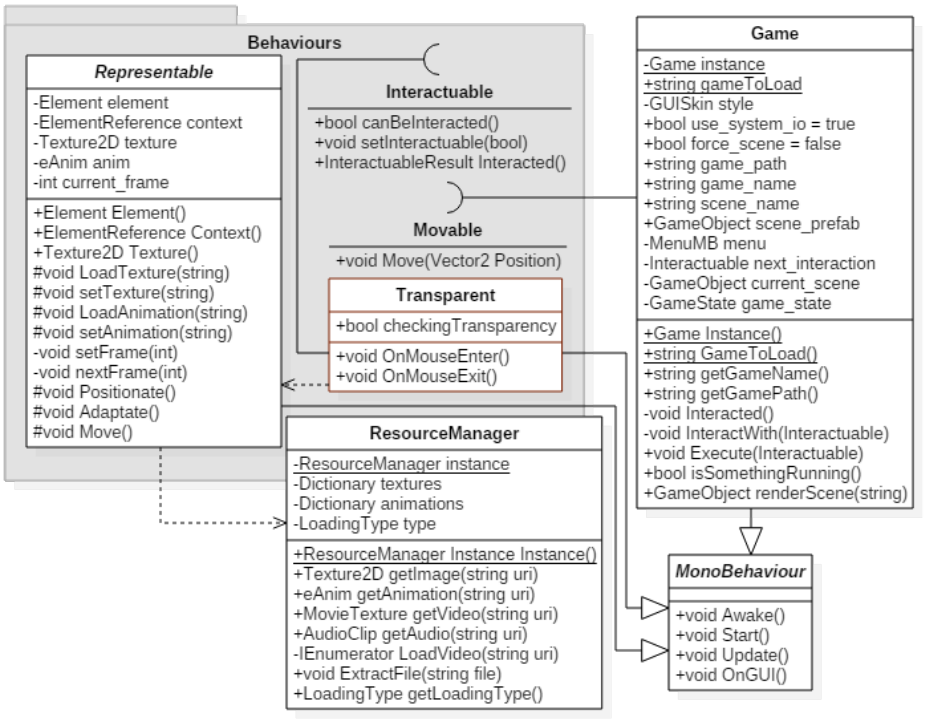

Figura 10.5: *Diagrama de clases de Representable, Interactuable, Movable y Transparent*

10.2.3. La escena y sus elementos

La escena de eAdventure, además de poder ser de multitud de tipos diferentes, está llena de elementos, algunos de ellos que están "vivos", otros que permiten interacción, y otros que únicamente son elementos que mejoran la representación visual. En conjunto, todos estos elementos permiten representar lo que se desee dentro de un escenario. Los elementos que se presentan en este apartado evolucionan de los explicados en el apartado 9.2.3.

En este apartado se remarcan las diferencias entre las clases de comportamiento presentadas en el apartado 9.2.3 y las actuales, especificando cómo han evolucionado estas. En esencia, su funcionalidad se ha mantenido, o delegado en otras clases, así como han ganado funcionalidad extraída de la clase *Game*. Esto es debido a que siguen siendo los mismos elementos que conforman la escena.

Otro detalle es, para diferenciar más fácilmente entre las clases del modelo de datos y los comportamientos, se les han añadido las siglas MB de *MonoBehaviour*, que identifican que son comportamientos y no clases de datos.

Detalles de la escena: SceneMB

La escena ha sido refactorizada y ampliada para incluir más funcionalidad y facilitar el uso de otra. Detalles de esto se puede encontrar en la figura 10.4.

En primer lugar, ahora la escena da soporte a vídeos, aunque estos vídeos han de estar en formato OGV[1]. Se ha realizado una investigación para determinar si es posible realizar una conversión del formato de los vídeos y su explicación se encuentra en el apartado dedicado a la tercera iteración del proyecto.

En segundo lugar, la escena ha simplificado la instanciación de sus elementos, e implementa dos métodos para dicha tarea, los cuales son:

```
private void instanceElement<T>(ElementReference context) where T : Element
private void instanceRectangle<T>(Rectangle context) where T : Rectangle
```

[1]El formato OGV es el contenedor de archivo de vídeo que utiliza el códec libre de video Theora.

Estas funciones, pese a ser funciones tipadas, tienen la peculiaridad de delimitar el tipo de clases que admiten. Esto se consigue utilizando *where T : Interface*.

En tercer lugar, como ahora el juego tiene soporte a juegos en tercera persona, en los que el jugador forma parte de la escena, esta ahora puede contener trayectorias para que el jugador se pueda mover sobre ellas. El cálculo de estas trayectorias fue una de las tareas más complejas de implementar, y se explica en el apartado 10.2.3.

Por último, ahora la escena gestiona la interacción que se realice sobre ella, aunque es diferente dependiendo de qué tipo de escena se esté mostrando:

- **Scene**: Si la escena es una escena normal, y además el juego es en tercera persona, se busca el punto que sea más cercano a la trayectoria, se calcula una ruta para que el jugador llegue a ese punto, y se le indica al jugador que inicie el movimiento dada dicha ruta.

- **SlideScene**: Si la escena contiene diapositivas, pasaremos a la siguiente diapositiva. Si por el contrario ya no quedan, se continuará a la escena que la siga, ya sea la anterior o una nueva.

- **VideoScene**: Si se recibe una interacción, se para el vídeo, y se continúa cargando la siguiente escena igual que en una *SlideScene*.

Los objetos del juego: ObjectMB

Con respecto a lo explicado en el apartado 9.2.3 acerca de la antigua clase *eObject*, esta ha sufrido una evolución. Se ha simplificado mucho gracias al uso de *Representable*, *Interactuable* y *Transparent*. Ahora únicamente reacciona si la componente *Transparent* se lo permite, y gracias a *Representable* y al *ResourceManager* es mucho más sencillo cambiar entre la textura normal y la textura activa, que se debe mostrar cuando el ratón pasa por encima el objeto.

Por otra parte, los objetos implementan una nueva funcionalidad, que les permite ser arrastrados en vez de generar un menú contextual cuando se interactúa con ellos. La espe-

cificación de cuándo esto debe suceder se encuentra en la propia especificación del *Item* que se está representando. Se remarca lo descrito en el apartado 10.2.2 acerca de la desaparición de las clases de datos descritas en el apartado 9.2.2, y la sustitución de las mismas por el modelo de datos de eAdventure.

Las deformables ActiveAreasMB

Como se ha mencionado en apartados anteriores, en la versión final, las *ActiveAreas* adquieren la capacidad de deformarse. Esto se consigue gracias a la utilización del Área de Influencia facilitado en la especificación del *ActiveArea*.

En la función *adaptate()*, una *ActiveArea* se encarga de leer esta *InfluenceArea*, y transformar estos puntos, que se encuentran en coordenadas relativas a la escena, a puntos colocados alrededor del centro de ellos. Tras esto se utiliza la libreria *LibTessDotNet* que genera una lista de triángulos a partir de esos puntos.

Una vez se obtienen los puntos y la lista de triángulos, se genera una nueva malla tridimensional estableciendo los vértices en dicha malla, y los triángulos en forma de referencias a los índices de la lista de vértices. Una vez generada dicha malla, se intercambia la *SharedMesh*[2] por la nueva malla, consiguiendo como resultado un objeto tridimensional con la forma deseada.

Finalmente, este elemento de la escena, como muchos otros elementos interactivos, tienen una componente *AutoGlower*, la cual se explica en el apartado 10.2.6, que hace que brillen durante un instante cada intervalo regular de tiempo.

Esto se ve representado en la Figura 10.6. Las figuras blancas son las áreas de influencia que dan forma a dichas areas activas. Estas capturas han sido obtenidas del juego utilizado para formar acerca de los primeros auxilios, desarrollado por CATEDU [8].

[2]La SharedMesh es una componente de los objetos tridimensionales de Unity3D que contiene la malla tridimensional que lo representa.

Figura 10.6: *Representación de las ActiveAreas en el videojuego FirstAid*

Las clases vivas: PlayerMB y CharacterMB

De las clases mostradas anteriormente, existen dos clases que implementan la interfaz *Movable*, pues necesitan moverse por la escena. Estas son *CharacterMB* y *PlayerMB*.

En esencia estos dos elementos de la escena son iguales, salvo por la peculiaridad de que *CharacterMB* permite interactúar con el, y realizar acciones que requieran la interacción con ella. Como dicha interacción es trivial, únicamente se explica la parte relacionada con la gestión del movimiento y animaciones.

En primer lugar, cuando en una de estas clases es invocada la función *Move()*, se genera una cola de puntos a los que se debe ir. Tras esto se establece el primer nodo de la cola como nodo destino, y, dependiendo de la dirección hacia la que se deba desplazar para alcanzar dicho nodo, se muestra una animación u otra. Cuando el movimiento se completa y se alcanza el nodo destino, el siguiente nodo de la cola pasa a ser el nodo objetivo, y se realiza el mismo proceso hasta completar la cola de nodos.

En segundo lugar, cuando una de estas es llamada a hablar, si esta tiene animaciones para el diálogo, se activan mientras la burbuja de diálogo siga activa.

Las trayectorias: TrayectoryHandler

La gestión de las trayectorias dentro de la escena es una de las labores más complejas de realizar, pues, en general, los algoritmos de búsqueda de rutas suelen ser algoritmos complejos, y la interacción con una trayectoria, que es algo simbólico, una lista de puntos en el espacio, unidos por líneas, es una tarea compleja.

Por ello, para la gestión de trayectorias, se han implementado tres clases necesarias para funcionalidades concretas:

1. **TrajectoryHandler**: esta clase es la encargada de proveer el manejo de una *Trajectory* de eAdventure. Se genera a partir de una de estas, y facilita funciones capaces de generar rutas entre puntos que se encuentren en dichas trayectorias.

2. **LineHandler**: esta clase se encarga de la gestión de una Línea de trayectoria. Está compuesta por dos nodos y la línea que los une. Tiene funciones para determinar que líneas son vecinas de esta línea, cuales son los puntos de contacto a partir de otro punto, o si saber si un punto está contenido en dicha línea.

3. **MathFunction**: esta clase, dados dos puntos de una recta, mediante el uso de funciones matemáticas, es capaz de generar un valor Y para una X dada, o un valor X para una Y dada, así como es capaz de generar puntos de contacto a partir de otro punto.

El tratamiento de trayectorias comienza con la generación de un *TrajectoryHandler*. Este analiza dicha trayectoria, y para cada lado de esta genera un *LineHandler*. Una vez que todos han sido generados, se llama a la función *UpdateNeighbors()* que recorre esta lista de líneas y, si dos líneas contienen extremos comunes, se determina que estas son vecinas.

Las líneas, cuando son generadas, utilizan sus dos puntos para generar una *MathFunction* que les ayudará a generar y a determinar puntos que estén en su línea.

Lo complejo del tratamiento de trayectorias llega con el cálculo de rutas. Para ello se ha implementado una variante del algoritmo de Dijkstra [4] para hallar caminos mínimos en

grafos no direccionales con coste. La diferencia es que, debido a que en la ruta el nodo inicial y el nodo final no forman parte de la trayectoria en si misma, sino que son nodos nuevos que se generan en tiempo de ejecución, y únicamente se identifica en qué líneas de trayectoria están contenidos dichos puntos. Esto se realiza de esta manera porque la Trayectoria es una clase extraída del videojuego, existente en el modelo de datos de eAdventure importado en C# y no se debe modificar, pues su modificación cambiaría la especificación del juego y causaría errores e incoherencias a la larga.

Tras esto se identifica que líneas contienen el punto de origen y el punto destino, y con esto se realiza el algoritmo de marcado de Dijkstra pero únicamente utilizando las líneas, sin los nodos. Y finalmente se ejecuta la función *reach()* que prepara el lanzamiento de la función recursiva de mismo nombre encargada de recorrer el grafo siguiendo la especificación del algoritmo de Djikstra.

No obstante, el funcionamiento de este algoritmo no es tan bueno como se desearía, pues le falta incluir la distancia que hay entre el punto de origen o destino, y el nodo inicial elegido de la línea de origen y destino. Asimismo, se pueden dar resultados no óptimos. Como trabajo futuro se plantea implementar el algoritmo de Djikstra con nodos, e incluir los nodos origen y destino en la trayectoria.

Las clases que participan en el manejo de trayectorias y en la generación de rutas están representadas en la Figura 10.7, donde se muestra un diagrama de clases con las dependencias y relaciones de las mismas.

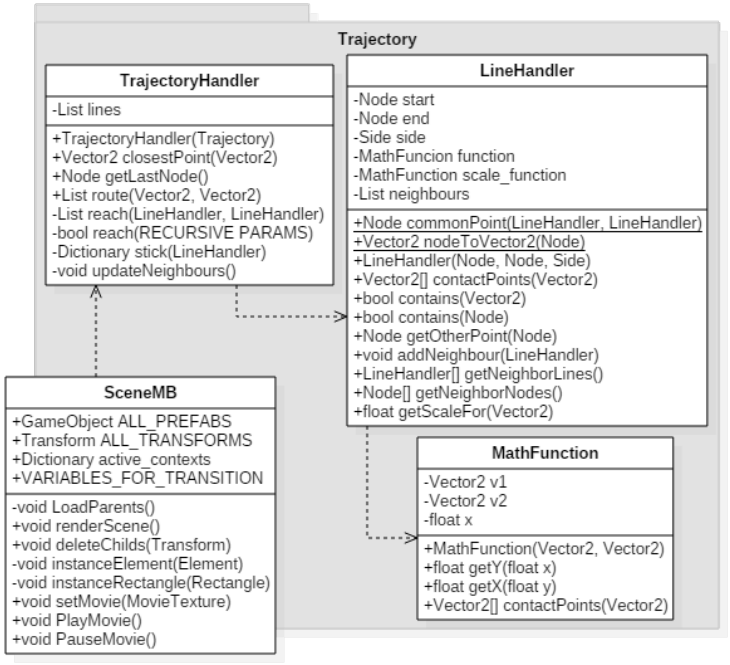

Figura 10.7: *Diagrama de clases de las trayectorias, junto a LineHandler y MathFunction*

Por ello, se plantea como trabajo futuro una nueva versión de este algoritmo, utilizando nuevas estructuras que encapsulen y repliquen la funcionalidad de las trayectorias de eAdventure

10.2.4. Las secuencias: Effect y GraphConversation

La implementación de las secuencias, visible en la figura 10.8, tanto de efectos, como de conversación cambió mucho desde la primera iteración hasta la segunda iteración. En esencia, se han eliminado las factorías de efectos, pues estas ya no necesitan ser autogeneradas a partir del fichero de especificación en XML, pues el paquete *Loader* del modelo de eAdventure ya se encarga de realizar dicha lectura.

Sin embargo, siguen manteniendo la esencia, y no se ha implementado ningún *SequenceManager* que se encargue de ejecutar las secuencias. En su lugar, siguen siendo autoejecutables, siendo más sencillo de recordar estados si la escena se para en la mitad de su ejecución y a su vez contiene llamadas a secuencias dentro de si mismas.

Mas allá de esto, el esfuerzo se realizó añadiendo la mayor parte de efectos disponibles en eAdventure, y únicamente dejando unos pocos por implementar, y por otra parte, haciendo funcionar los *Holders*, con las clases del modelo de datos de eAdventure.

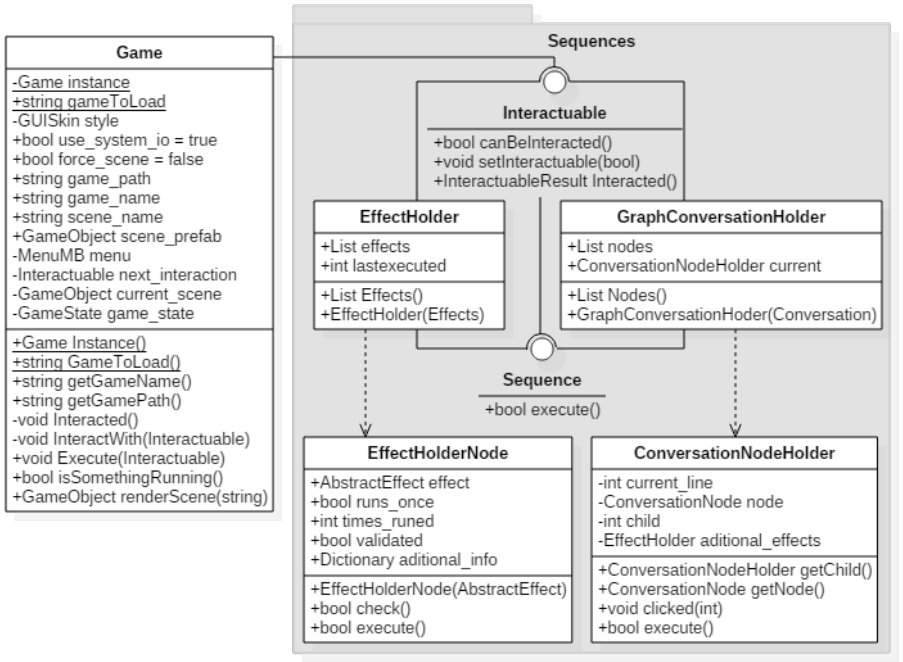

Figura 10.8: *Diagrama de clases de las Secuencias, mostrando tanto EffectHolder como GraphConversationHolder*

10.2.5. El controlador de los temporizadores: TimerController

El controlador de los temporizadores es una clase nueva que no existía en la anterior iteración. Esta clase comportamiento se añade como componente a la cámara y se encarga, mientras está contando, de controlar los *Timers* que el juego necesita para su ejecución.

Para manejar los temporizadores se ha implementado un enumerado llamado *TimerType* que establece el tipo de temporizador que se está tratando, y una clase llamada *TimerState* que únicamente sirve para almacenar datos acerca de los temporizadores que se encuentran en ejecución.

Este controlador tiene además la opción de parar su ejecución para que, si el juego se encuentra parado esperando a que el jugador interactúe en una secuencia, no salten temporizadores en mitad de dicha secuencia.

Para gestionar estos temporizadores se tienen tres listas: una lista con todos los temporizadores de dicho capítulo, una lista con los temporizadores que están ejecutándose, y por último una cola con los que ya se han completado.

Se aprovecha de la función *Update()* de *MonoBehaviour* para controlar el paso del tiempo en los mismos, así como detectar qué temporizadores cumplen las condiciones para ejecutarse, y cuáles de los que se encuentran en ejecución ya no deberían estar ejecutándose.

La Figura 10.9 muestra el diagrama de clases de lo descrito anteriormente. En este diagrama se ve como *Game* establece los *Timers* que deben ser controlados al inicio de cada capítulo, y *TimerController* únicamente los gestiona. Cuando alguno de estos *Timers* se completa, *TimerController* le dice a *Game* que ejecute los efectos de dicho temporizador.

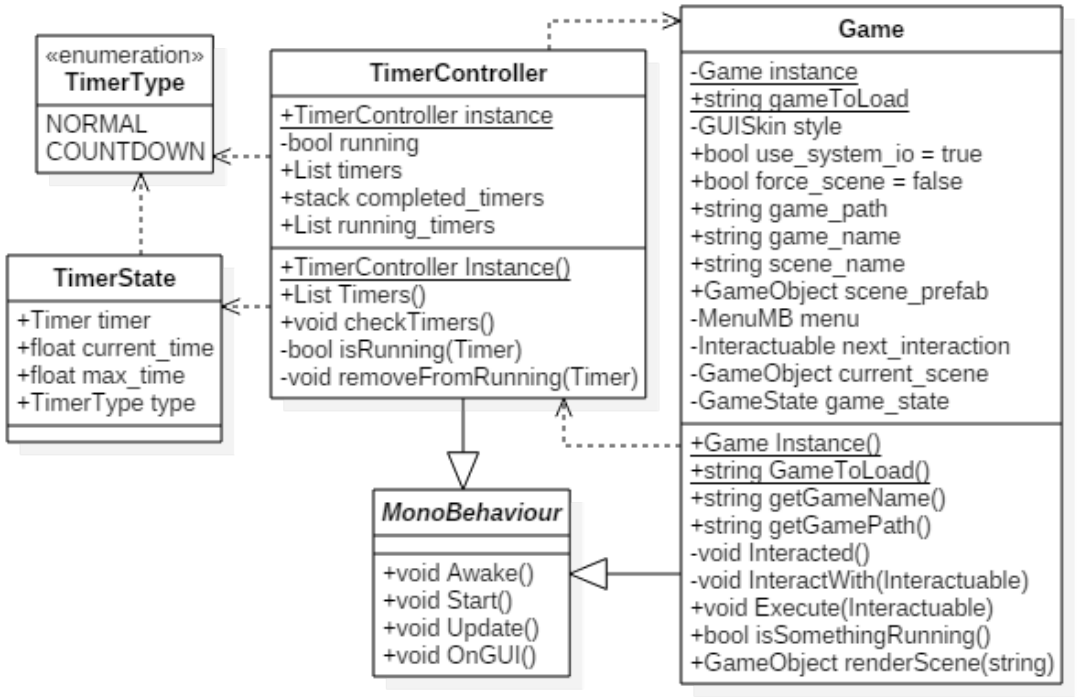

Figura 10.9: *Diagrama de clases de TimerController, incluyendo las clases y enumerados para su funcionamiento.*

10.2.6. El gestor de interfaz: GUIManager

Como se explica en el apartado 10.2, existe una clase gestora que se encarga de manejar la interfaz en su totalidad de forma transparente. Facilita una serie de funciones que permiten, de forma sencilla, mostrar burbujas de diálogo, cambiar el cursor, mostrar un menú contextual de acciones, o un menú de opciones.

Este gestor surge de la reducción de contenido de *Game*, que antes realizaba estas tareas, y fue relevada de su función debido a que dicha clase comenzaba a tener una dimensión demasiado elevada. La Figura 10.10 muestra el diagrama de clases de la clase *GUIManager* junto al paquete *Apearance*, el paquete *GUIProvider* que contiene todo lo necesario para acceder al *ResourceManager* de forma automática y transparente, así como la carga de recursos por defecto y la gestión de nombres a constantes y viceversa, y el paquete *Menu*, que contiene una serie de clases para la representación y animación del Menú.

Este gestor hereda de *MonoBehaviour*, por lo que es asignado como componente de un elemento dentro de la escena. En este caso se asigna a un objeto *Canvas*, que permite la representación de elementos de interfaz.

El ciclo de vida de *GUIManager* comienza cuando despierta la escena. En este momento se establece como instancia a si mismo. Tras esto, cuando *Game* ya ha cargado el juego, y se ejecuta la función *Start()*, este provee al *GUIProvider* de los datos del juego en forma de un *AdventureData*. Este se prepara y carga los recursos que deba cargar, y tras esto, el *GUIManager* se queda en espera hasta que el juego necesite mostrar al usuario algo.

Una vez que el *GUIManager* se encuentra en espera, facilita una serie de funciones útiles para representar cosas al usuario, estas son:

- **Talk(string line, string talker)**: Esta función permite al sistema que los personajes hablen. Esto se ha implementado mediante el uso de burbujas.
- **showActions(List<Actions>, Vector2 position)**: Esta función pone en ejecución al menú contextual, y hace que se muestre con las acciones especificadas en una posición.
- **showOptions(ConversationNode)**: Esta función permite mostrar un listado de opciones de texto que el usuario debe elegir. Adicionalmente, el fondo se emborrona y se muestra el texto de la última pregunta.

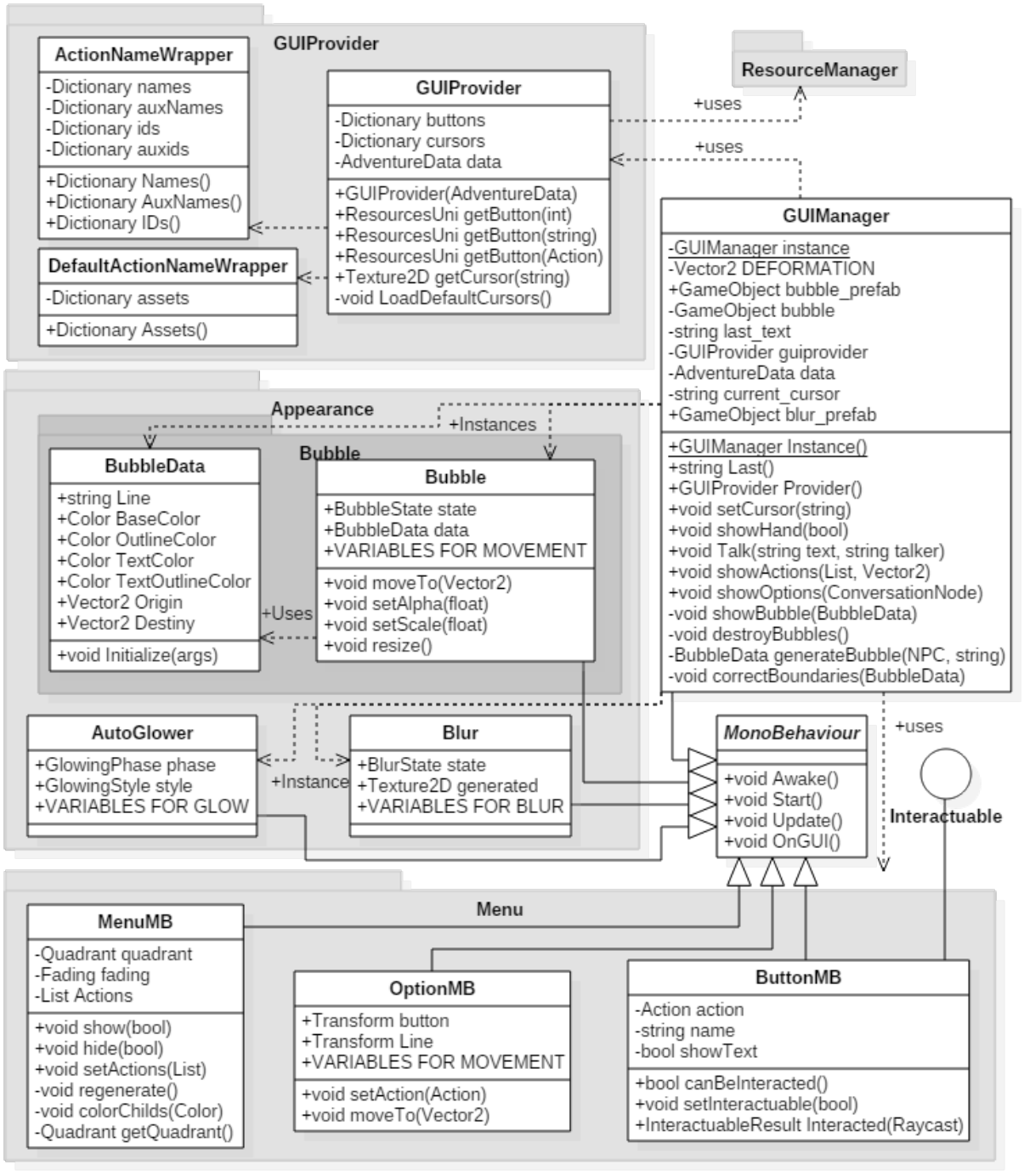

Figura 10.10: *Diagrama de clases de los grandes gestores que controlan y proveen contenido para la ejecución del juego.*

Proveedor de recursos de GUI: GUIProvider

La interfaz es una capa de la vista que necesita representación, y por ello, necesita acceder a recursos para poder representarse. Sin embargo, nuestro gestor de *GUI* no usa directamente al *ResourceManager*, sino que utiliza una clase intermedia para acceder a los recursos, esta clase es *GUIProvider*, y se encarga de, facilitar el acceso a texturas que tengan una funcionalidad concreta. Es decir, por ejemplo, obtener la textura que representa a una acción, o el cursor que se debe mostrar para un determinado elemento. *GUIProvider* se encarga de proveer los recursos apropiados.

Además de esto, eAdventure sigue una jerarquía a la hora de definir recursos para la interfaz, esta es la siguiente:

1. Se cargan los recursos por defecto. Estos recursos están repartidos en dos directorios, el primero es la carpeta "/gui/hud/contextual/" donde se encuentran los botones que representan los distintos tipos de acciones y que se obtienen utilizando la clase *DefaultActionNameWrapper*. El segundo es la carpeta "/gui/cursors/" donde se encuentran los cursores del juego, y que se obtienen utilizando la clase *ActionNameWrapper*. Estas dos clases *Wrapper* se encargan de transformar de Acción, o identificador de Acción a nombre, y viceversa.

2. Se cargan los recursos especificados en la base del archivo "descriptor.xml". Tras la lectura de dicho archivo en *Loader*, estos se encuentran en el objeto de datos *AdventureData* que gestiona *GameState*. Estos también se asignan utilizando los *Wrappers* anteriormente especificados, y, sustituyen a los botones y cursores cargados en el apartado anterior con los que se hayan en esta especificación.

3. Por último puede ocurrir una situación anómala, y es que un elemento tenga una acción personalizada en su interior. En este caso, y excepcionalmente, es el propio botón del menú el que se encarga de solicitar al *ResourceManager* dichos recursos.

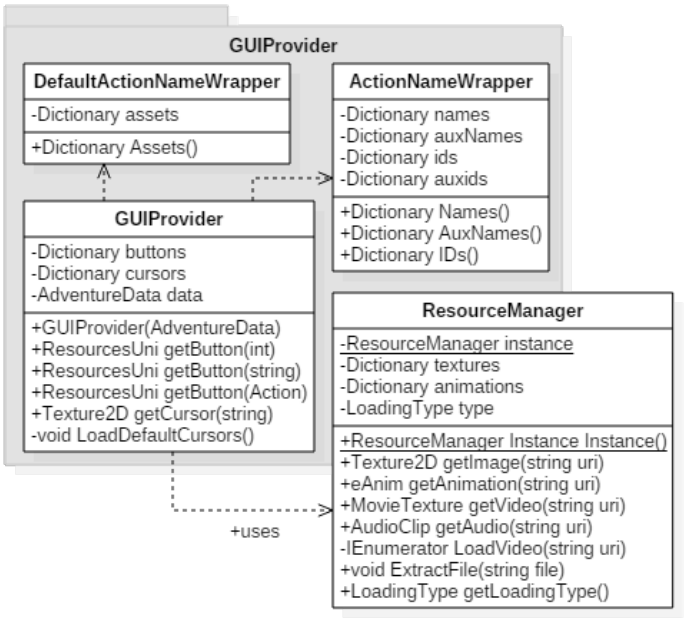

Figura 10.11: *Diagrama de clases de GUIProvider, junto con ResourceManager.*

Las clases que participan en este proceso están representadas en la Figura 10.11. En esta figura, aunque *ResourceManager* esté sobrepuesto al paquete *GUIProvider*, esto es debido a reducir el tamaño del diagrama, realmente este no forma parte de dicho paquete.

Las burbujas de diálogo: Bubble

La representación de diálogos en un juego de aventuras es una tarea necesaria, pues los personajes necesitan hablar para compartir información entre ellos. La manera de representar estos diálogos en uAdventure es la misma que se decidió utilizar en eAdventure: Las burbujas de diálogo, o bocadillos de diálogo. Sin embargo, estas burbujas han evolucionado en su representación, realizando una pequeña animación a la hora de ser invocadas, y a la hora de desaparecer.

El proceso de mostrar una burbuja de diálogo comienza en *GUIManager* y es el siguiente: En primer lugar, se identifica si el que habla es el jugador o un personaje, y si es el jugador, se distingue si se trata de un juego en primera persona o un juego en el que el jugador tenga

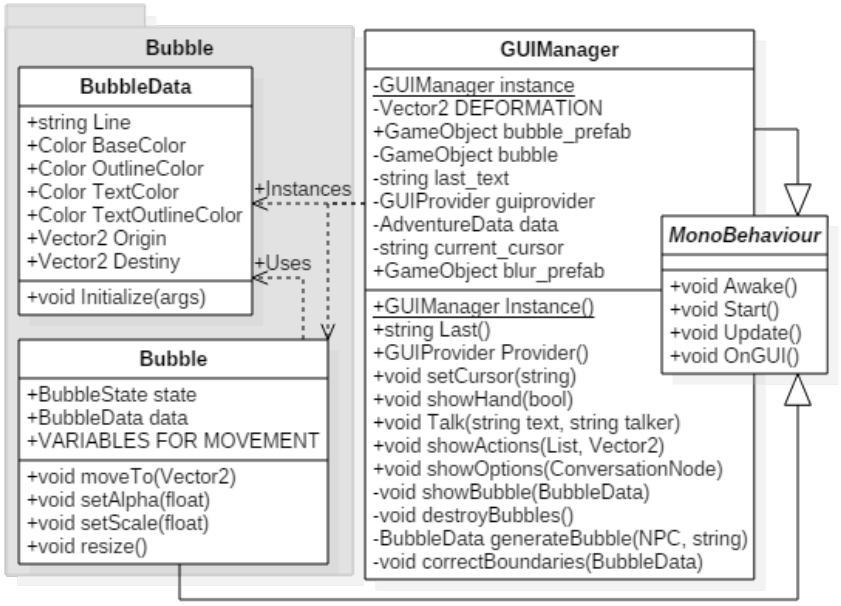

Figura 10.12: *Diagrama de clases de Bubble, junto con GUIManager y MonoBehaviour.*

representación en el juego. Una vez se ha determinado quién habla, se obtienen los datos del hablante y se genera una *BubbleData* con dichos datos, decorándola y estableciendo su trayectoria. Este *BubbleData* se pasa a través de una serie de deformaciones que adaptan dichas trayectorias a la pantalla para que no se salga de la misma, y para que se coloque correctamente independientemente de la resolución de la misma. Tras esto se instancia una nueva burbuja en la escena y se delega en ella.

La burbuja, en su función *Start()*, establece su texto en la representación, se prepara para moverse, y se prepara para empezar a mostrarse, pues al principio es transparente. Poco a poco esta se va moviendo y haciéndose visible en su función *FixedUpdate()*, y cuando termina, se queda quieta. Cuando es necesario que la burbuja desaparezca, se ejecuta la función *destroy()* que la hace desaparecer poco a poco, haciéndose más pequeña y transparente.

Elementos de mejora visual: Appearance

Existen dos clases que, para mejorar la representación visual del juego, se han desarrollado en el proyecto. Cómo se explicó en el apartado 6.3 del estado del arte, y visible en la Figura 6.5, el proyecto eAdventure Android utilizaba mecanismos específicos para ayudar al usuario a encontrar elementos en un entorno táctil. En este proyecto, para ayudar al usuario a encontrar estos elementos, se realiza un brillo de los mismos.

La clase componente *AutoGlower* se encarga de realizar este brillo sobre los objetos que la contengan. Tiene varios modos, como simplemente hacer un flash, o aparecerse y desvanecerse y hacer un flash. Este *AutoGlower* utiliza un *Shader* que genera dicho flash y que es parametrizable para establecer tanto la posición del brillo, su color y su anchura. Este efecto es visible en las Figuras 10.14 y 10.15;

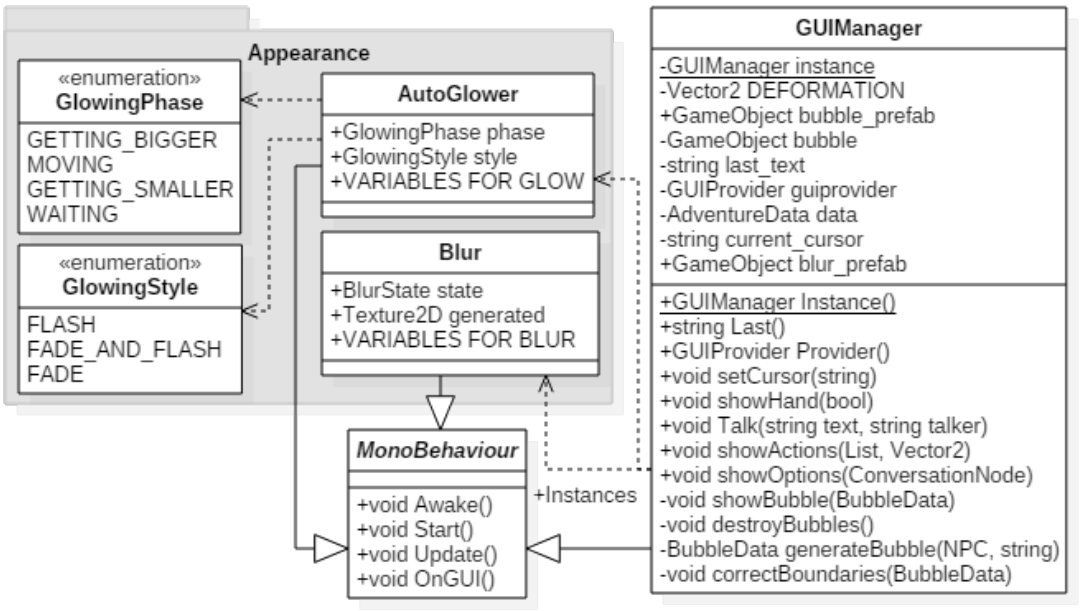

Figura 10.13: *Diagrama de clases de Apearance, sin incluir Bubble.*

Por último, tenemos a la clase *Blur*, que, al igual que la anterior utiliza un *Shader* para conseguir su efecto visual. En este caso, *Blur* lo que hace es volver borroso lo que hay detrás de ella. Es utilizada para generar un cuadrado borroso sobre el cual presentar las distintas

opciones disponibles en la lista de opciones de una conversación. Este efecto se puede ver en la Figura 10.16.

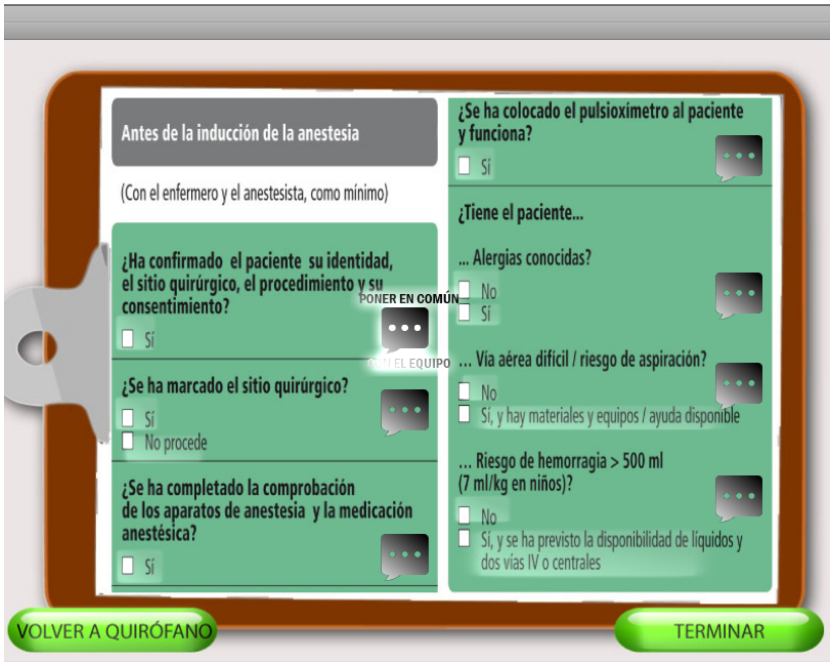

Figura 10.14: *Brillo de los elementos causado por la componente de Appearance AutoGlower.*

El menú contextual de acciones: Menu

El menú contextual es uno de los elementos que existían antes y que ha sido recreado, añadiendo animación al mismo para que su representación visual sea más sencilla. La utilidad del menú contextual es mostrar acciones en forma de botones para que el usuario pueda pulsarlos y ejecutar las acciones que están detrás de los mismos.

Este menú contextual se genera utilizando una de las funciones de *GUIManager*, la función *showActions(list<Action>)* que, inicializa el menú y utiliza su función *regenerate()*, que lo que hace es, en función de los parámetros que se le hayan establecido, genera nuevos botones, y los coloca para que se animen correctamente. Tras esto la clase *OptionMB* se encarga de ir colocando tanto la línea que une el centro del menú con el botón, como el botón en si mismo. Asimismo, esta clase establece la acción en *ButtonMB*, clase que implementa la

Figura 10.15: *Progresión de imágenes que muestra el brillo de los elementos.*

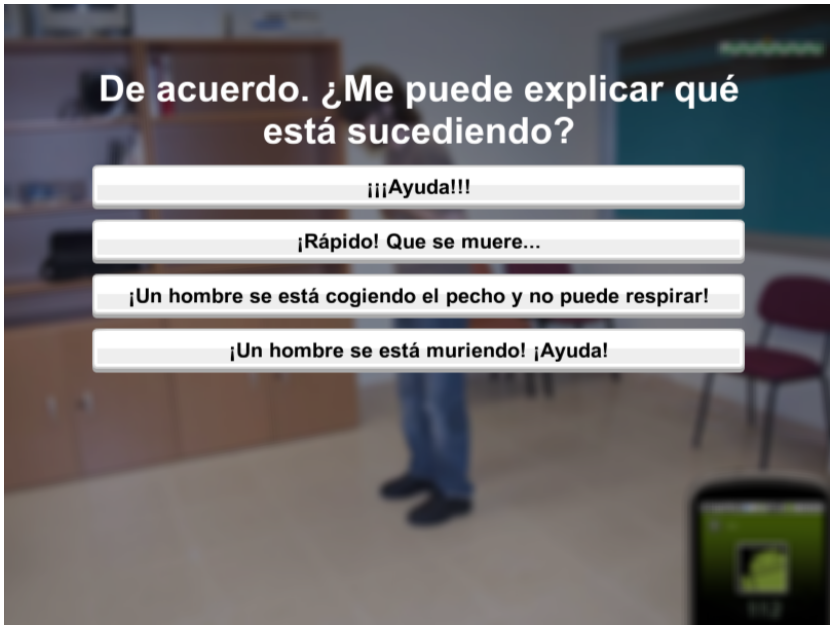

Figura 10.16: *Captura de una lista de opciones entre las que el jugador puede elegir. Se muestra el fondo borroso causado por Blur.*

interfaz *Interactuable*, y que permite al usuario interactuar con ella. Es la clase *ButtonMB* la que se encarga de modificar su representación en función de la imagen que le provea el

GUIManager. Visible en la Figura 10.17, donde participan *MonoBehaviour* e *Interactuable*.

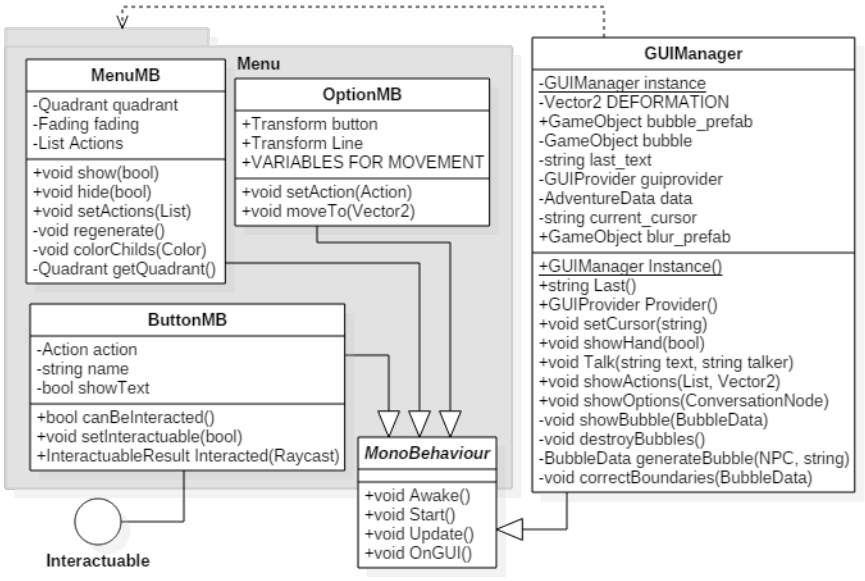

Figura 10.17: *Diagrama de clases de Menu, incluyendo las relaciones que tienen con MonoBehaviour e Interactuable.*

Finalmente, la Figura 10.18 muestra el menú generado en una situación en la que se pueden realizar cinco acciones sobre un elemento de la escena.

10.2.7. El gestor de recursos: ResourceManager

Como se ha comentado varias veces a lo largo de este apartado, para facilitar el acceso a los recursos desde cualquier parte del proyecto, existe un *Singleton* llamado *ResourceManager* que provee de elementos multimedia al que lo solicite.

Este *ResourceManager* es capaz de cargar múltiples tipos de recursos, y para ello utiliza métodos diferentes. Asimismo, existen dos formas de cargar recursos, y las clases que se encargan de acceder al sistema de archivos para cargar los recursos se adaptan a estos tipos.

Estos tipos de carga, o *LoadingType*, son: utilizando el sistema de ficheros del sistema operativo, mediante la librería *System.IO*, teniendo la ventaja de que, una versión ya generada de uAdventure puede cargar cualquier juego que se establezca en un directorio determinado;

Figura 10.18: *Imagen de la representación visual del menú.*

y por otra parte, utilizando *Resources.Load()*, mediante el sistema de ficheros de Unity, que tiene la ventaja de que estos archivos, una vez que se produce una *Release* del proyecto, están incluidos en un paquete comprimido por Unity, y permiten instalar uAdventure junto al juego en cualquier plataforma sin tener que preocuparse de incluir el juego.

Con respecto a los tipos de ficheros que se pueden cargar con *ResourceManager*, encontramos los siguientes tipos:

- *Imágenes*: Para la carga de imágenes existe una clase llamada *Texture2DHolder*. Dependiendo del tipo de carga que utilice *ResourceManager*, o bien se carga la textura directamente con *Resources.Load()*, o bien se leen los *bytes* de dicho fichero, para más adelante transformarlos en una *Texture2D*.

- *Animaciones*: Las animaciones que se cargan son las de eAdventure, y están compuestas por una serie de imágenes, esto se hace en la clase *eAnim*, la cual no ha cambiado mucho en su funcionamiento desde lo explicado en el apartado 9.2.2. Para añadir soporte a *Resources.Load()* ha sido necesario implementar un método que renombra todos los archivos ".eaa" a ".xml" pues Unity no reconoce dichos archivos como ficheros

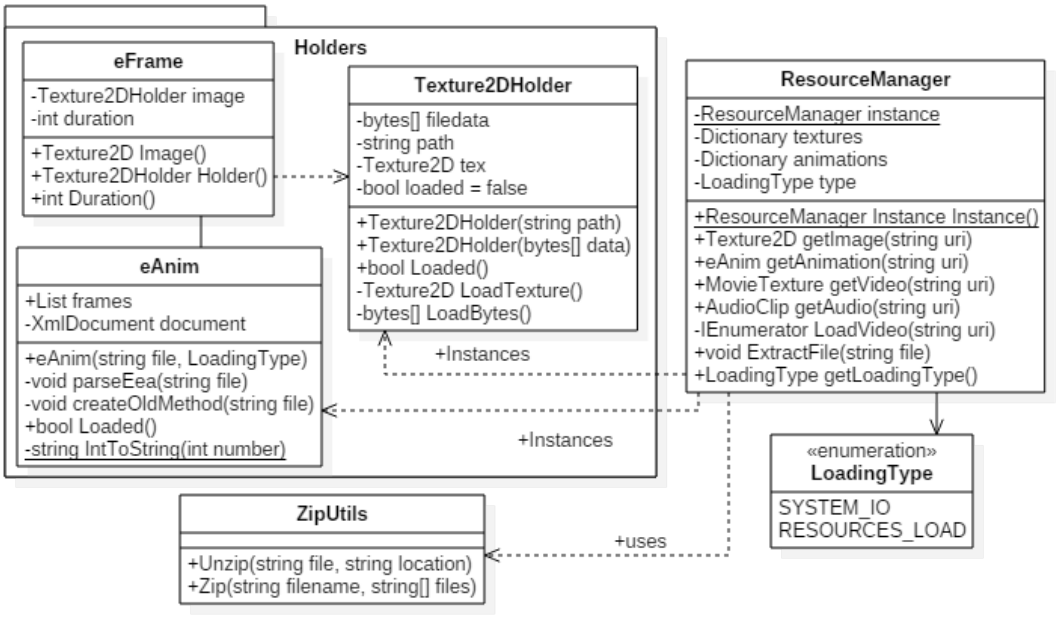

Figura 10.19: *Diagrama de clases de ResourceManager junto a los cargadores de recursos.*

de texto y no permite cargarlos.

- *Vídeos*: La carga de vídeos se realiza almacenándolos en una *MovieTexture*. Para la carga de estos desde el sistema de ficheros se utiliza la clase *www*, que permite hacer solicitudes POST y GET, ya sea en local, al sistema de ficheros, o a un servidor. Esta carga requiere la utilización de corrutinas, pues sin ellas el juego se quedaría completamente congelado hasta que el recurso estuviera cargado. Finalmente mencionar que los vídeos únicamente pueden estar en formato ".ogv" bajo el codec OGG Theora.

- *Audio*: De manera similar a los vídeos, se utiliza la clase *www* para su carga, aunque en esta ocasión son almacenados en *AudioClip*.

- *Archivos Zip*: Estos ficheros técnicamente no son cargados en tiempo de ejecución, sino que son extraídos bajo demanda del emulador para importar juegos. Estos juegos deben estar en formato ".jar", y únicamente se extrae de ellos la parte necesaria. Más detalles acerca de este proceso están disponibles en el apartado 11.1.

Acerca de los vídeos, y de la importación de los juegos que se realiza en la tercera iteración del proyecto, se realizó una investigación acerca de la posibilidad de transformar dichos vídeos en el proceso de importación del juego. Los resultados no fueron satisfactorios, pero abrieron varias puertas para abordar este problema.

En primer lugar, se probaron multitud de librerías que facilitaban la conversión de vídeos a sistemas .NET [22]. Sin embargo, ya que Unity utiliza una versión reducida del mismo, dichas librerías fallaban en tiempo de ejecución y no completaban la conversión del vídeo. Por otra parte, algunas personas que se han enfrentado al mismo problema, han generado *Wrappers* que utilizan un ejecutable del proyecto de software libre FFMPEG [6], que provee de una serie de utilidades para transformar vídeos fácilmente.

Se intentaron incorporar dichos *Wrappers* en el proyecto, pero ninguno de ellos funcionó. Finalmente se decidió que la alternativa de implementar un *Wrapper* muy sencillo, con únicamente la funcionalidad de transformar al formato OGV, sería la opción mas viable. en windows, esto se haría de la siguiente manera:

```
string path = " -i \"" + video + "\" -codec:v libtheora -qscale:v 7 -codec:a
    libvorbis -qscale:a 5 \"" + video.Remove(video.Length-4, 4) + ".ogv\"";
Process foo = new Process();
foo.StartInfo.FileName = System.IO.Directory.GetCurrentDirectory() +
    "/ffmpeg/ffmpeg.exe";
foo.StartInfo.Arguments = path;
```

También existe una manera de lanzar clases Java directamente desde Unity, pudiendo realizar este proceso también en Android, de la siguiente manera:

```
using (AndroidJavaClass ext = new AndroidJavaClass ("com.external.class"))
{
   ext.CallStatic ("whatever", Parameter1, Parameter2);
}
```

Por lo que, como trabajo futuro del proyecto, queda pendiente implementar dicho conversor de vídeos multiplataforma, que utilice diferentes librerías, ya sea *FFMPEG* u otra librería para dicha conversión.

10.3. El modelo de Datos: Core

En el proceso de integración del proyecto de Piotr Marszal 12 que consiste en reconstruir el editor de eAdventure dentro de Unity, se decidió unificar el modelo de datos. En vista de que, para una implementación al completo de todos los elementos del editor, el modelo de datos planteado en el apartado 9.2.2, con las clases de datos que eran capaz de generarse a si mismas, era insuficiente para satisfacer las necesidades del proyecto, se decidió invertir mucho tiempo en portar al completo el modelo de datos de las clases Java a Unity.

Esta tediosa y repetitiva labor fue realizada casi en su totalidad por Piotr, quien copiaba el código Java en C# y se encargaba de sustituir las librerías problemáticas por librerías análogas de .NET. Dentro de mi aportación, se realizaron tareas de corrección de errores y de pruebas para comprobar que todo funcionaba correctamente.

De esta misma manera, Piotr importó el *Loader* original de eAdventure, y no fue hasta que se comenzó a utilizar en este proyecto que no se identificó que dicho cargador era inviable de utilizar pues tardaba entre medio y un minuto en leer el documento de especificación del juego; además de que lo hacía con errores que, pese a intentos de arreglarlos, debido a que la implementación de dicho *Loader* fue realizada por uno de los desarrolladores de eAdventure, la complejidad del funcionamiento del mismo se escapaba a nuestra comprensión.

Como estos problemas con el *Loader* continuaban, se decidió crear una nueva implementación del cargador del modelo de datos utilizando la librería *System.XML* que se utilizó para las Clases de datos de la primera iteración, y que ya se conocía que era fácil de utilizar, pues permite utilizar sintaxis de XPath[3], y acceder al contenido del documento XML de forma sencilla.

Esta segunda implementación la comenzó Piotr, recodificando la mayor parte de *sub-parsers* del cargador. Sin embargo, él no pudo probar las nuevas clases que estaba implementando, por lo que no pudo darse cuenta de que muchas de ellas, debido a que la base

[3]XPath (XML Path Language) es un lenguaje declarativo que permite construir expresiones regulares que recorren, procesan y seleccionan elementos de un documento XML

que aplicaba para la transformación no era del todo correcta, no funcionaban. Los grandes *subparsers* como el cargador de escenas, efectos, conversaciones, areas activas, etc... tuvieron que ser nuevamente implementados desde cero en este proyecto. Finalmente, dentro de este proyecto se realizó la tarea de implementar los cargadores principales de *AdventureData* y *Chapter*, haciendo estos funcionales al sistema de cargado de ficheros de *System.IO*, así como el de *Resources.Load()*.

Una vez construido el nuevo *Loader*, los tiempos fueron reducidos en un 90 %, pasando de tardar 45 segundos de tiempo de carga en el videojuego *Checklist*, a alrededor de 4 segundos. No obstante esta medida no puede realizarse con precisión debido a que en dichos tiempos de carga se incluye el tiempo que utiliza Unity para inicializarse y ponerse en funcionamiento.

Para completar la integración fue necesario desacoplar partes del sistema de eAdventure entre si mismas, pues el modelo de datos de dicho sistema está extremadamente interconectado, y cada vez que se intenta reutilizar una de sus partes, es necesario traer junto a esa parte, un gran número de partes de dicho modelo.

Con dicho desacoplamiento, también fueron eliminados del modelo de datos determinados elementos utilizados para iniciar la carga del juego. Por ello fue necesaria una investigación en profundidad hasta hallar una manera segura de trabajar con el paquete *Loader*.

Finalmente, cuando se hubo unificado el modelo de datos, se realizó una unión de ambos proyectos en un único proyecto y repositorio. El proyecto de Piotr se ubica casi en su totalidad dentro del espacio de nombres *Editor*, por lo que, para la unificación de los proyectos, fue necesario extraer de dicho espacio de nombres el modelo de datos, junto al *Loader*, otras clases auxiliares, y más elementos. A esta parte que se extrajo de *Editor* es a la que se conoce como *Core*, pues representa al núcleo de eAdventure.

10.4. La comunicación con RAGE: RAGETracker

Pese a que uno de los objetivos principales de este proyecto es el de garantizar y simplificar el ciclo de vida de los juegos serios producidos con eAdventure, así como extender su vida útil lo máximo posible, otro de los planteamientos con los que surge este proyecto es el de poder mejorar la parte de evaluación de eAdventure mediante el uso de RAGE.

Una de las partes que, por tener un grado de dificultad muy elevado de incluir en Unity, debido a que las librerías que se utilizaban no dan soporte a .NET, y se debe construir una nueva infraestructura para soportarlo, es la parte de *Assessment*. Estas partes, que además no tenían toda la funcionalidad deseada, van a ser sustituidas por *Assessment* con RAGE.

RAGE, Realising an Applied Gaming Eco-system [36], es un proyecto que apunta a desarrollar, transformar y enriquecer tecnologías específicas para la industria de los videojuegos serios, en forma de paquetes autocontenidos para juegos que ayudan a los estudios de videojuegos en el desarrollo de juegos serios, haciéndolo más sencillo, rápido y eficiente económicamente.

En RAGE hay una parte específica dedicada a la evaluación de videojuegos, y esta parte provee una API para los profesores que deseen crear perfiles de evaluación y progreso, así como otros elementos de utilidad que se deseen monitorizar, como el numero de veces que una pregunta ha sido respondida de forma incorrecta, o los días que los estudiantes han sido más activos jugando a un videojuego.

En eAdventure, este proceso de evaluación se realiza mediante la especificación de una serie de frases de evaluación, en lenguaje natural, que, cuando se den una serie de condiciones, serán incluidas en un registro de evaluación. Este registro se muestra al jugador al terminar el juego, y este debe enviar dicho registro al profesor, o debe ser revisado por el profesor al terminar la clase.

Este proceso tiene multitud de problemas, entre los cuales podemos encontrar: casos como que el profesor nunca llegue a ver dicho registro de evaluación, ya sea porque el alumno decida no enviar dicho registro al profesor, o porque el alumno lo cierre sin querer;

problemas a la hora de evaluar porque no sea suficientemente específico en el desarrollo del juego; o el gran problema de que, para poder realizar la evaluación, es el propio desarrollador el que tiene que incluir elementos de evaluación dentro del juego. Sin embargo, todos estos problemas son asumibles, porque se entiende que el profesor va a tener tiempo para dedicar a cada alumno, hablar con él en caso de que tenga dudas acerca del desarrollo del juego, y que los juegos desarrollados, en general, son de corta duración, y pueden ser completados de inicio a fin en una sesión.

Pero, ¿qué ocurre si pasamos de tener el ámbito de una clase, a una cantidad de alumnos que sobrepasa el límite del profesor? ¿Podría leer los registros de todos los alumnos y evaluar uno por uno? La respuesta es no, o al menos no de forma apropiada. Por ello, es necesario que este proceso de evaluación sea rediseñado y reconstruido utilizando las ventajas que RAGE provee a los desarrolladores de juegos. Delegando este proceso de evaluación, y únicamente monitorizando las desviaciones que sufren los alumnos, se podrá conseguir que el proceso de evaluación de un número de alumnos elevado, mediante el uso de un juego generado por uAdventure, se realice de forma sencilla y cómoda.

Además de las ventajas que RAGE proveerá a uAdventure, el segundo se encargará de facilitar la generación de estos perfiles de evaluación desde el propio editor de uAdventure. Un ejemplo de este proceso automatizado sería, la generación del progreso del usuario dentro del juego a través del árbol de escenas que comunica la escena inicial con la escena final. También se podrían marcar variables importantes para el desarrollo del juego, para su monitorización, y se generarían gráficas de su evolución, así como poder generar alertas si una de dichas variables se saliese de un rango apropiado. Finalmente, se intentaría preservar el antiguo sistema de alertas de eAdventure, utilizando el sistema de alertas de RAGE.

Recalcar también que RAGE no permite únicamente facilitar la evaluación al profesorado, sino que también ayuda a investigadores y desarrolladores, permitiendo generar estadísticas como las preguntas más falladas, y la opción más elegida dentro de ellas; generación de mapas de calor dentro de una escena, para saber las zonas donde más interactúa el usuario,

o saber las escenas donde el jugador pierde más el tiempo, o le resultan más complejas de superar.

En cuanto a la funcionalidad que *RAGETracker* aporta en si mismo, únicamente facilita el proceso de comunicación con RAGE. Implementa unas clases que realizan la conexión con el API, autorizan al usuario mediante un *Token* de identificación, y relacionan el juego al que se está jugando, con un juego dado de alta en RAGE. La implementación de dicho *Tracker* está disponible en https://github.com/e-ucm/unity-tracker.

Capítulo 11

Tercera Iteración: El Emulador y los editores

Como se explica en el apartado 5, en donde se expone la metodología de desarrollo utilizada para el proyecto, este pasa a lo largo de tres grandes iteraciones para completarse. En este apartado se explican los detalles implementados en la tercera iteración, es decir, la generación de un emulador de juegos e eAdventure, utilizando el núcleo de ejecución *Runner* explicado en el apartado 10.2, y el modelo de datos de eAdventure explicado en 10.3; además de la generación de editores para el editor de uAdventure desarrollado por Piotr Marszal.

11.1. El Emulador de Juegos

Uno de los principales problemas de todo software consiste en que el presupuesto que se invierte en el desarrollo de dicho software no debe invertirse por completo en su desarrollo, pues la vida útil del mismo durará varios años más, en los que debe haber un mantenimiento y soporte para dicho software. El software producido por eAdventure, en muchos casos, hace años que dejó de tener soporte, pues no sólo los desarrolladores abandonaron los proyectos, sino que eAdventure no sufre ninguna actualización desde el 29 de Octubre de 2012, y todos los errores que han surgido, y los problemas de soporte que tiene en diferentes dispositivos han limitado la vida útil del software producido con eAdventure, y su ciclo de vida.

uAdventure, construido sobre Unity3D permite abrir proyectos de eAdventure, y video-

juegos ya generados, dando la posibilidad de modificarlos y probarlos sobre el nuevo núcleo de ejecución, ampliando y simplificando el ciclo de vida del videojuego. Tras haber abierto dicho juego, mediante las herramientas de compilación de Unity se puede generar un paquete ejecutable con el juego en su interior.

No obstante, muchos proyectos han sido completamente abandonados, y es posible que nadie decida transformar un juego o que, por desconocimiento de su existencia, no se transforme a uAdventure. Por todo ello, este *framework* puede ser compilado en sí mismo como un emulador de juegos de eAdventure, permitiendo al usuario explorar su sistema de ficheros, e importar de forma sencilla aquellos juegos que desee jugar.

Este emulador evoluciona de una prueba incluida en la clase controladora del juego en el prototipo generado en la primera iteración. Esto consistía en permitir al usuario seleccionar un juego de una lista de juegos disponibles en lugar de lanzar directamente un juego especificado en la clase controladora. En este primer prototipo la interfaz era muy sencilla, sin opciones, y no permitía técnicamente importar juegos, sino que debían de importarse descomprimiendo los juegos en una carpeta determinada del sistema de ficheros. Además, incluía un diálogo de carga que mostraba el progreso de la carga del juego.

Esta primera versión del emulador se muestra en la Figura 11.1.

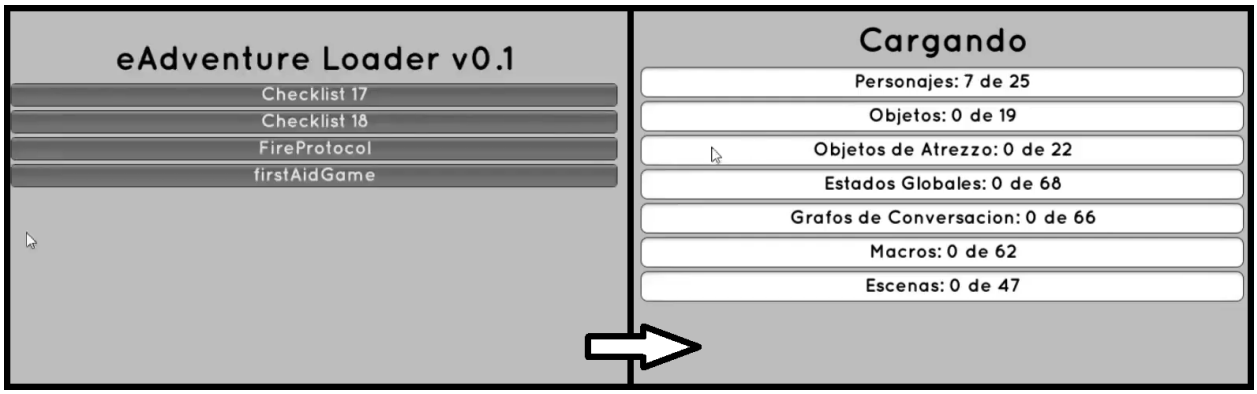

Figura 11.1: *La parte de la izquierda muestra el menú principal del primer prototipo del emulador. Una vez seleccionado un juego, se mostraba un diálogo de carga como el que se ve en la parte derecha.*

No obstante, y dado que esta primera versión del emulador era insuficiente, además de

que el diálogo de carga de ficheros no permitía ser ejecutado si se utiliza el sistema de carga de recursos de Unity, *Resources.Load()*, este emulador evolucionó en cinco vistas:

- **Menú Principal**: El menú es la puerta de entrada al emulador. Cuando se pone en funcionamiento muestra en el centro un catálogo de juegos instalados, si los hay. Pulsando sobre cualquiera de estos juegos comienza la ejecución del mismo. Finalmente, en la parte inferior de la vista hay tres botones que permiten acceder a más vistas del emulador. Cada juego presentado en el menú principal tiene un icono que lo representa. La obtención de dicho icono se hace explorando los archivos incluidos dentro del directorio "/gui/" en busca de un fichero llamado "standalone_game_icon.png". Si dicho fichero no existe, se selecciona el icono de la versión de eAdventure con la que fue generado dicho juego, incluido también en el mismo directorio.

Figura 11.2: *Vista del menú principal del emulador de uAdvneture.*

- **Explorador de Archivos**: Dado que Unity no provee, en tiempo de ejecución, de una manera de seleccionar un fichero del sistema de archivos, se ha implementado desde cero un explorador de archivos. Este permite navegar por carpetas, las cuales se ven en un color amarillo, y seleccionar juegos para ser añadidos. Estos juegos se representan con el icono de un mando de videojuegos, y con un color azul. Tras seleccionar un juego se puede pulsar el botón *Add Game*, que lanza la vista que se presenta a continuación.

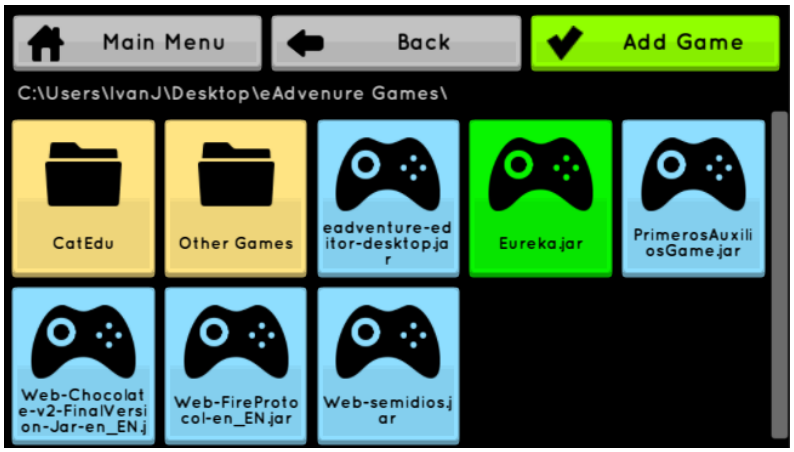

Figura 11.3: *Vista del explorador de archivos del emulador de uAdventure.*

- *Importador de juegos*: Pese a que esta sea una de las vistas más sencillas, pues únicamente muestra un texto y una pequeña animación de tres puntos moviéndose, tras esta animación se encuentra el proceso de importado del juego. En el que se descomprimen las partes necesarias, y se borran algunos elementos tras su descompresión. Como se explicó en el apartado 10.19, la transformación de los vídeos se realizaría en esta escena.

Figura 11.4: *Vista del importador de juegos del emulador de uAdventure.*

- *Configuración*: La ventana de configuración provee al usuario de la capacidad de poder modificar el funcionamiento del intérprete, pudiendo configurar tres apartados:

Gráficos, Sonido y Otros. Dentro del apartado gráfico se permite la posibilidad de deshabilitar los *Shaders* explicados en el apartado 10.2.6, así como deshabilitar los vídeos, o las animaciones. Dentro del apartado de sonido se permite configurar el uso de audio, así como su volumen y si se desea, forzar el uso de *Text To Speech*, texto hablado. Finalmente, en el apartado de otras configuraciones, se permite modificar la velocidad del texto en las burbujas, así como la posibilidad de dehabilitar RAGE, o la persistencia de ficheros multimedia, para reducir el uso de memoria de la aplicación.

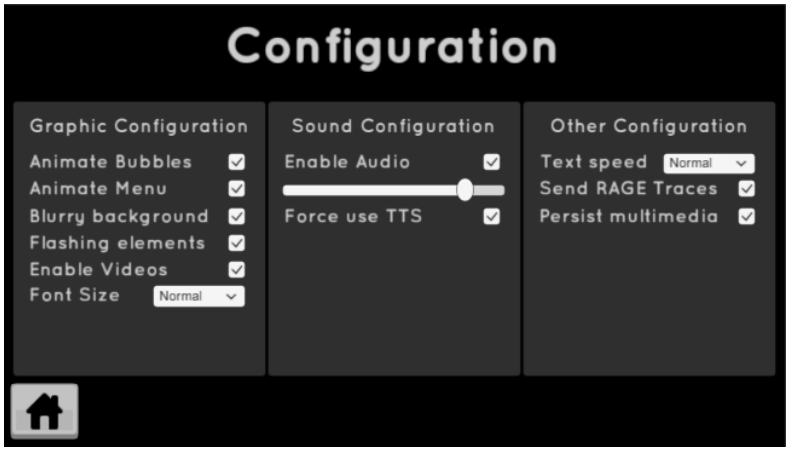

Figura 11.5: *Vista de configuración del emulador de uAdventure.*

- *Créditos*: La vista de créditos contiene el listado de las personas que han participado, dirigido o realizado aportaciones tanto a uAdventure como a eAdventure.

11.2. Los Editores de Secuencias

Como parte del proceso de integración de los proyectos, y para realizar un aporte comparable al esfuerzo que Piotr Marszal había realizado al importar el modelo de datos de eAdventure, en el desarrollo de este proyecto se generaron dos editores basados en ventanas para la edición de las secuencias.

Dada la experiencia de generación de editores de este estilo obtenida durante el desarrollo del TFG IsoUnity, que consistía en la generar un editor de videoaventuras en tercera persona

Figura 11.6: *Vista de los créditos presentados en el emulador de uAdventure.*

con estilo isométrico, en el que se incluyen editores que presentan similitudes con dichas necesidades, el desarrollo de dichos editores será más sencillo.

Las clases de eAdventure que requieren editores son: los efectos, que pese a ser lineales, se plantea la posibilidad de incluir bifurcaciones que los transformen en un grafo en un futuro; Las conversaciones, que son grafos de diálogo con nodos de conversación que incluyen diálogo o opciones de respuesta; y el editor de condiciones, que al ser tan pequeño se explicará como parte del editor de efectos.

Para el desarrollo de cada uno de estos editores se implementarán tres clases cruciales:

- La ventana principal del editor, que es capaz de gestionar la secuencia en si misma, compuesta por un listado de nodos, y de mantener en orden las ventanas que representan los editores individuales de cada uno de los nodos.

- Una interfaz que permita implementar editores para los nodos, con métodos para pintarse utilizando la *GUI* de editor de Unity, y métodos que permiten identificar si un editor sirve para representar un tipo de nodo, y clonarse a si mismo.

- Finalmente, una factoría que explore el espacio de nombres utilizando *Linq*, en busca de editores que implementen la interfaz para generar editores de nodo. Dicha factoría también tiene que proveer un listado de los editores disponibles, así de facilidades para

instanciar un editor para un nodo concreto. Existen detalles acerca de la implementación de factorías utilizando esta técnica en el apartado 9.3.2

Con estas tres clases, ya se pueden generar editores individuales para los nodos, consiguiendo un editor de secuencias muy extensible, preparado para añadir nuevos editores de forma sencilla y sin necesidad de modificar ninguna de las clases. Generando una nueva clase que implemente la interfaz del editor, este ya la identificaría y añadiría a su listado de editores disponibles.

11.2.1. El Editor de Efectos

Entre los editores de secuencias implementados, el más sencillo, pero a su vez más largo de implementar es el editor de efectos, pues estos son una secuencia lineal, no son ni árboles, ni grafos, por lo que la implementación de un editor con ventanas lineal se plantea sencilla. Sin embargo, la cantidad de editores de efectos que se deben implementar es muy elevada, pues existen multitud de efectos que implementan la interfaz *AbstractEffect*.

Para la implementación de esta ventana de editor, se extiende la clase abstracta *EditorWindow* que provee una serie de métodos que, de implementarse, permiten la generación de una nueva ventana para el editor. En primer lugar, una función *Init()* que inicializa la ventana para un efecto dado. Una funcion *nodeWindow(int id)* que se encarga de gestionar cada ventana individualmente. Unas funciones *CreateWindows()* y *CreateWindow(AbstracEffect effect)*, que son las encargadas de generar las ventanas para los efectos. Una funcion *curveFromTo()* que genera una línea entre dos ventanas, y finalmente, la función *OnGUI()* que ejecuta todo lo anterior para generar la ventana.

En la Figura 11.7 se ve una secuencia de efectos sencilla, compuesta por 5 efectos en los que se establece valor para unas variables, el jugador habla, y finalmente, se lanza una conversación completa. El menú desplegable muestra todos los editores de efectos implementados. En la parte de arriba de la Figura se observa un botón muy ancho con el texto "New Effect", que sirve para añadir un nuevo nodo a la lista de efectos. En esta ventana,

las subventanas que permiten la edición de efectos se pueden mover, por lo que permiten la organización de los mismos de la forma más conveniente.

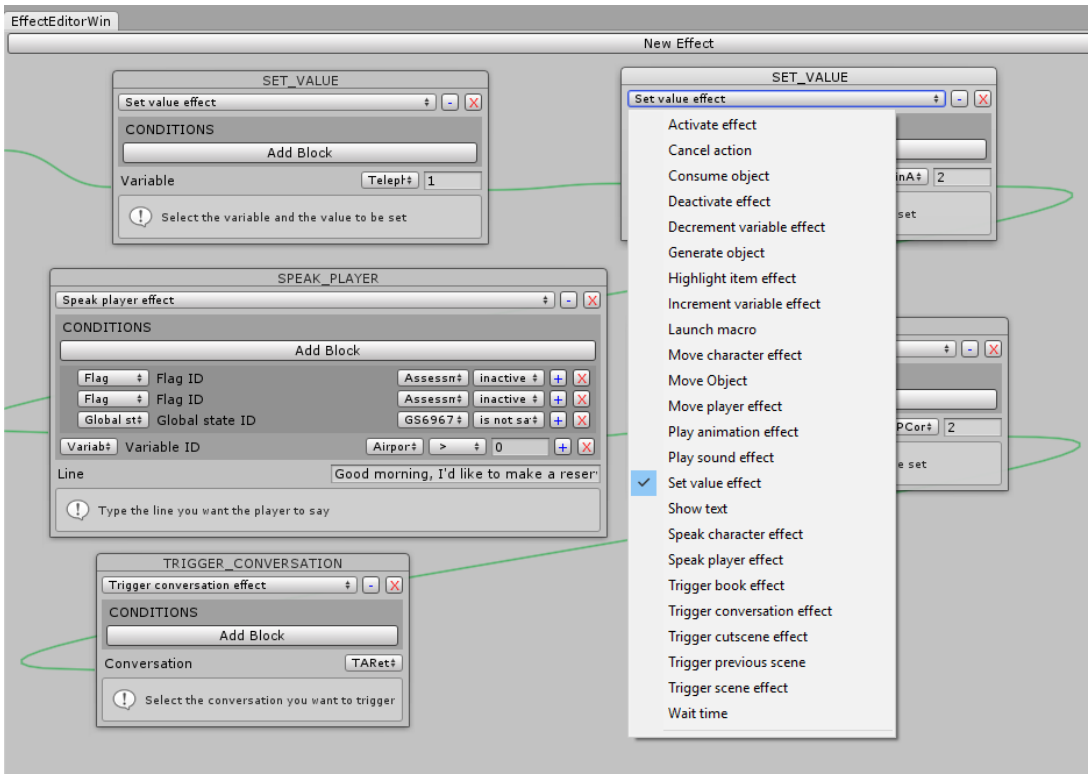

Figura 11.7: *Ventana principal del editor de efectos.*

En la figura 11.8 se muestra un nodo de efecto que muestra una gran cantidad de condiciones para su ejecución. Este nodo puede reducirse en tamaño para facilitar la edición mediante el botón con el guión de color azul, transformándose en la ventana derecha.

11.2.2. El Editor de Conversaciones

Pese a que en primera instancia parece que el editor de conversaciones es muy similar al editor de efectos, y aunque visualmente lo sean, las conversaciones son grafos, y la modificación de un editor que trabaja con listas a un editor capaz de trabajar con grafos direccionales no es una tarea trivial.

Las modificaciones con respecto al editor de efectos están relacionadas con la gestión de ciclos en los grafos, y la posibilidad de generar hijos para nodos concretos, así como permitir

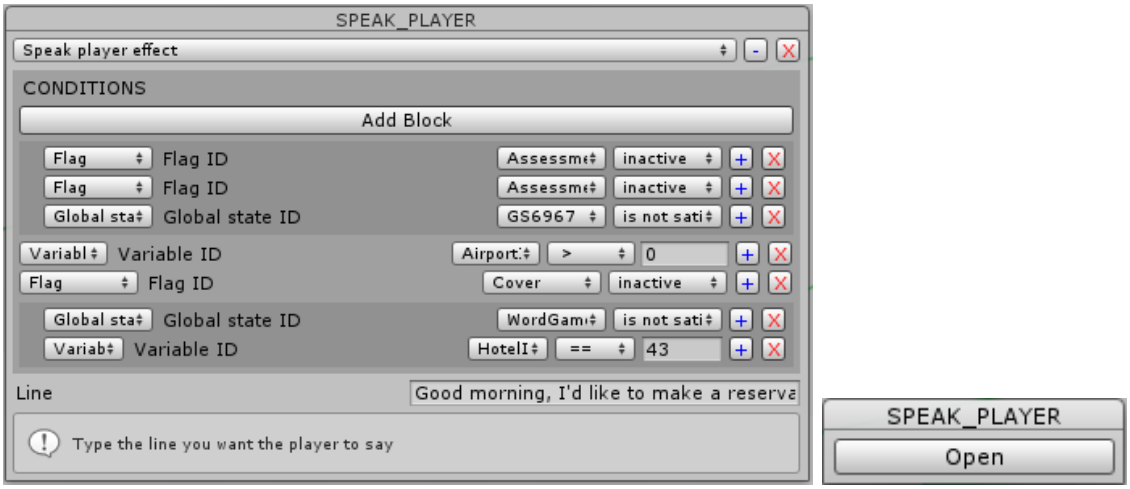

Figura 11.8: *Editor de nodo de efecto con condiciones, vista normal y reducida.*

al usuario establecer el hijo de un nodo directamente, mostrando en la interfaz cual es la rama de nodos que, si no tiene ningún otro nodo padre, se va a perder, y cual es la nueva rama que se va a establecer como hija de dicho nodo.

Como en el desarrollo de este editor no era tan necesario generar un gran número de editores, pues las conversaciones únicamente están compuestas por editores de diálogos y de opciones, dichos editores cuentan con un acabado mucho más conseguido, reutilizando muchas de las imágenes de la interfaz de eAdventure. Esto se muestra en la Figura 11.9, en la que, además, se muestra lo descrito en el párrafo anterior. La línea azul será el nuevo hijo, y la roja el hijo descartado.

Como detalle adicional, se ha generado una clase llamada *NodePosicioner* que, estableciendo un número de nodos, y una altura de la ventana, posiciona automáticamente los nodos en su lugar correspondiente automáticamente, de manera que ver las secuencias es más cómodo si se abren y no existen datos acerca de su posición. Esto se muestra en la figura 11.10.

Finalmente, y para mostrar las posibilidades del editor, tenemos la Figura 11.11, en la que se muestra una secuencia muy grande. El icono del candando nos permite la edición de condiciones para mostrar dicha línea, ya sea de opción o de diálogo. La flecha negra permite

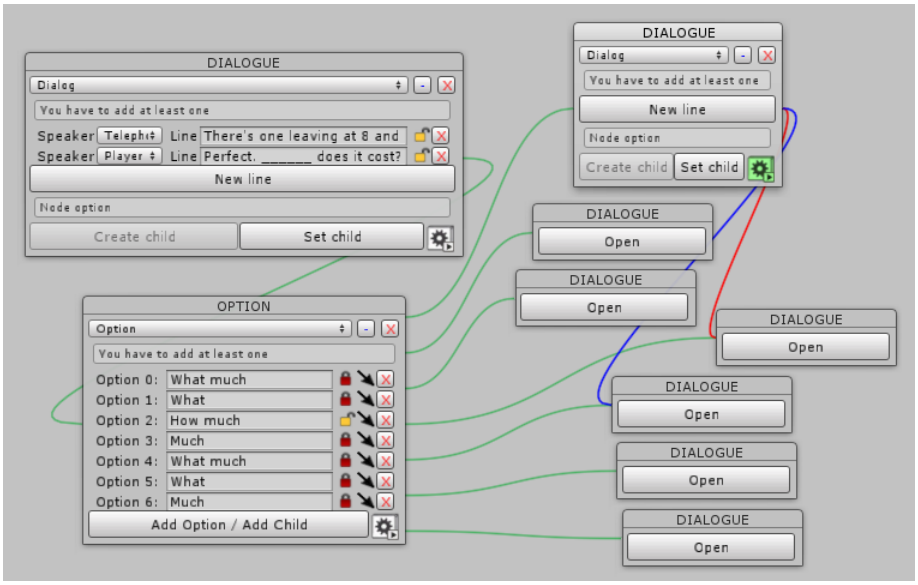

Figura 11.9: *Pequeño diálogo mostrado en el editor de diálogos.*

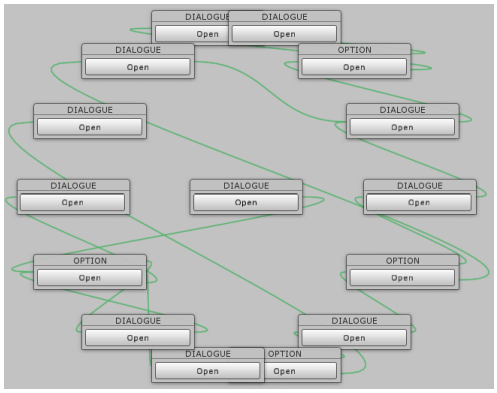

Figura 11.10: *Posicionamiento automático de nodos en el editor de diálogos.*

establecer el nodo hijo para dicha opción directamente. Y por último, el botón de la esquina inferior derecha de cada diálogo permite acceso a los efectos que se ejecutan en dicho nodo. Estas características también se pueden observar en la Figura 11.9

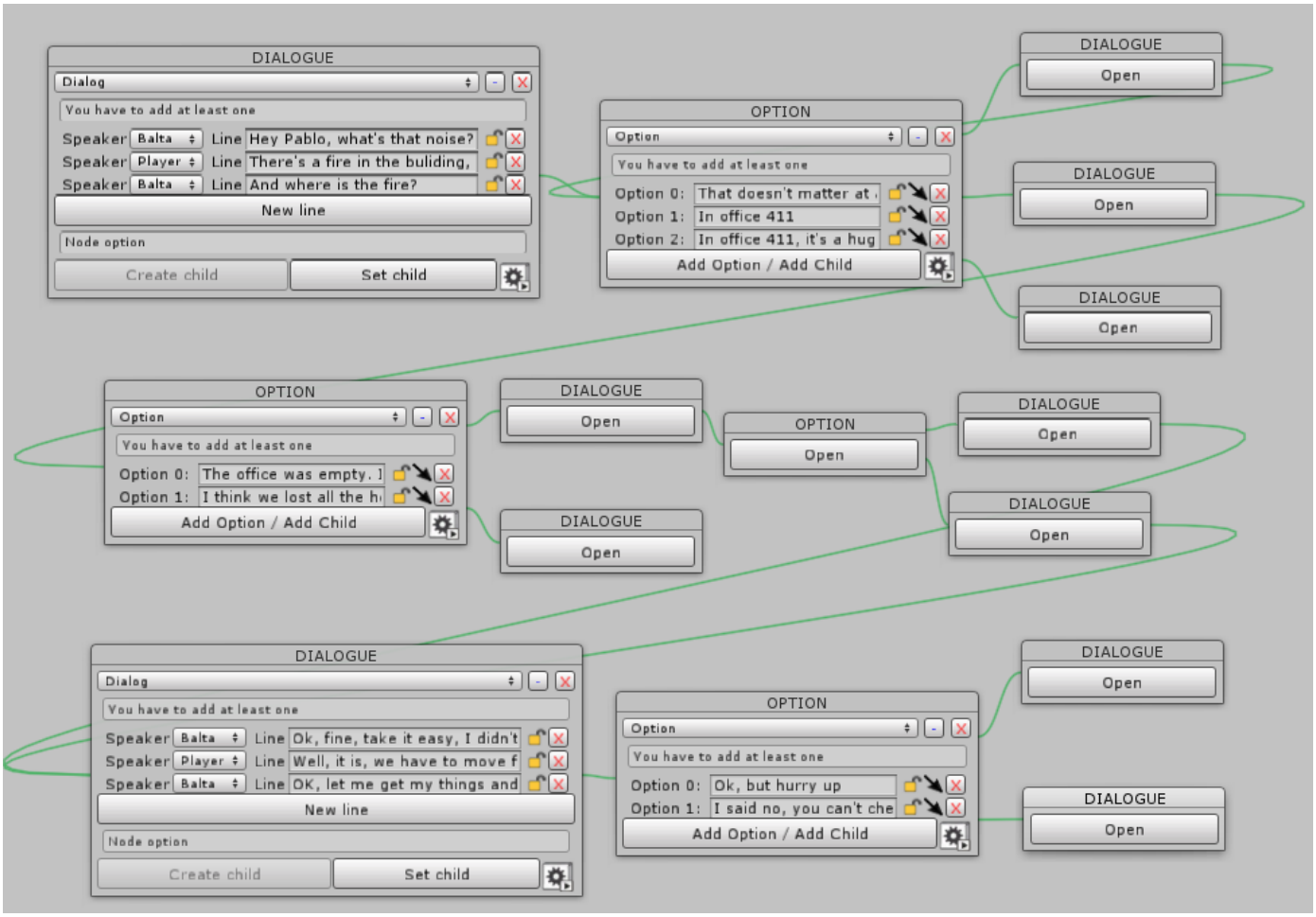

Figura 11.11: *Gran diálogo mostrado en el editor de diálogos.*

11.3. El Editor de RAGE

Como se ha mencionado varias veces a lo largo del documento, especialmente en el apartado 10.4, donde se explica la necesidad de sustituir las funcionalidades de evaluación de alumnos por un sistema más elaborado que el que encontramos en eAdventure; el sistema de evaluación que se utiliza en uAdventure está fuera del mismo, en la nube, y se llama RAGE.

Esta herramienta provee, mediante una interfaz web, una serie de páginas con información importante y gráficas acerca del progreso y evaluación de los alumnos de forma en grupo e individual, permitiendo al profesor enfocar los esfuerzos en aquellos alumnos que

hayan quedado rezagados o que demuestren problemas en su aprendizaje.

Pese a que esta interfaz no provee de funcionalidad específica más allá de la que provee la propia interfaz web de RAGE, las clases y funciones implementadas serán utilizadas como trabajo futuro para guardar las configuraciones de evaluación y progreso de videojuegos en RAGE. El propósito de esta ventana se conservará, permitiendo, si se desea, modificar los algoritmos generados por uAdventure para el cálculo de estos elementos.

La Figura 11.12 muestra las clases que participan en el editor de RAGE. En estas clases se encuentra *RageWindow*, que provee de interfaz y de controlador para la interacción con RAGE, la clase *GameConfiguration*, que es una clase contenedora de los datos de la configuración de un juego en RAGE, con la capacidad de transformarse en JSON, igual que *Warning*. Por otra parte, se encuentra ThreadedNet, una clase que permite realizar peticiones POST y GET de forma paralela utilizando hilos de procesamiento.

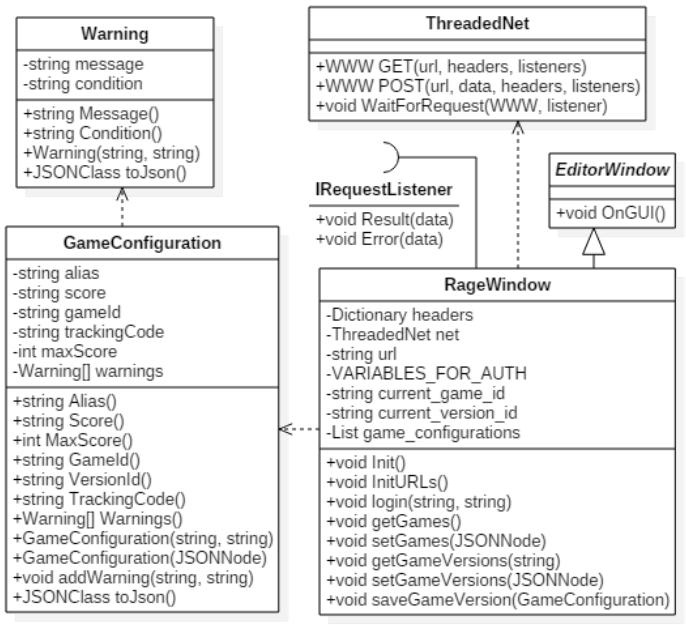

Figura 11.12: *Diagrama de clases de las clases que participan en el editor de RAGE.*

Capítulo 12

El editor de uAdventure

A lo largo de este documento se referencia en múltiples ocasioes el proyecto de Piotr Marszal. Este proyecto consiste en la reconstrucción del editor de eAdventure dentro de Unity, utilizando las extensiones de editor que Unity permite.

Como uno de los objetivos de este proyecto es dar soporte al ciclo de vida de los juegos de eAdventure, para conseguir que los desarrolladores que crearon juegos en eAdventure no se sientan demasiado fuera de lugar al trabajar con uAdventure, este proyecto ha mantenido una similitud exacta a la hora de recontruir los diferentes menús de la interfaz de eAdventure.

Este tipo de herramientas que extienden la interfaz de Unity son complejas de desarrollar, pues necesitan que el desarrollador trabaje con un enfoque diferente, pues los elementos y clases que participan en el proceso de edición son diferentes de los que participan en la ejecución del juego. Este cambio de mentalidad, junto con los problemas de serialización de elementos hacen que este tipo de proyectos sean complejos de implementar.

Este proyecto ha dado soporte a todas las características de eAdventure, con el aporte adicional de nuevos editores explicados en el apartado 11.2, en los que la interfaz se ha innovado para mejorar la capacidad de organizar los diferentes nodos que componen las secuencias de elementos.

Las Figuras 12.1 y 12.2 presentan una vista de la misma interfaz, siendo la primera la de eAdventure original y la segunda la del editor reconstruido en Unity.

Dentro de la lista de trabajo futuro planteado, se expone la posibilidad de mejorar la

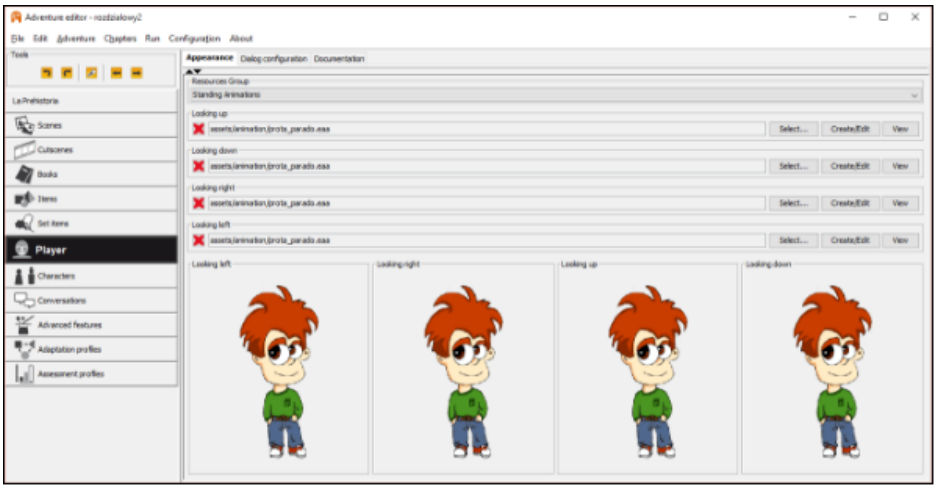

Figura 12.1: *Captura del editor original de eAdventure.*

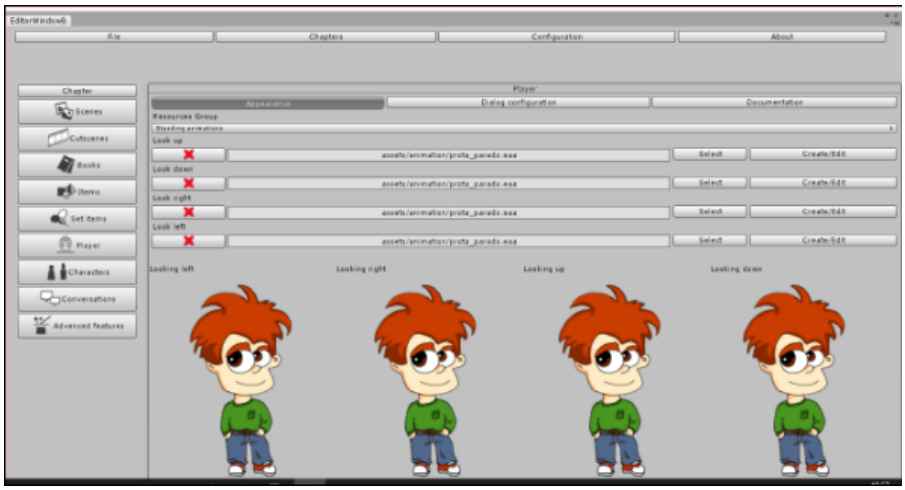

Figura 12.2: *Captura del editor reconstruido sobre Unity, uAdventure.*

interfaz para crear un editor híbrido que una las interfaces de Unity y uAdventure, preparando un perfil de visualización del editor que abstraiga menús y herramientas que puedan distraer a un desarrollador poco experimentando, y dando facilidades para encontrar lo que busque.

Capítulo 13

Conclusiones

Tras haber completado el proyecto, incluyendo el diseño, la implementación y la escritura de esta memoria, se extraen determinadas conclusiones que se deben abordar en este *post mortem* del proyecto. Primero se realizará una reflexión personal acerca de los resultados del proyecto y luego se presentarán conclusiones categorizadas dentro de los diferentes objetivos que se plantearon, estableciendo una relación entre objetivos planteados y resultados obtenidos.

Bajo mi impresión personal, estoy bastante satisfecho con el resultado obtenido, pues, en mayor o menor medida he completado la lista de objetivos que me planteé, aprendiendo gran cantidad de nuevos conceptos. Mediante la experimentación he aprendido que, al menos en mi caso, trabajo mejor realizando un pequeño diseño, en papel, de cómo me imagino que deberían funcionar las cosas, y desarrollando un prototipo que realice lo implementado, y me permita ver si el resultado obtenido es el esperado, o se necesita realizar un rediseño de la aplicación.

Una de las conclusiones que extraigo del proyecto es que es muy satisfactorio trabajar con un estudiante extranjero, de una cultura diferente, como es Piotr Marszal. La experiencia de trabajar con una persona que se toma su trabajo tan en serio, al que le puedes pedir resultados y responde ante tus necesidades, es una experiencia muy buena, pues te permite crecer como persona al conocer otra forma de trabajar, y hace que te impliques más en el desarrollo del proyecto, desarrollando un código de mayor calidad.

El listado de objetivos, planteado en el capítulo 7, de forma simplificada, es el siguiente:

1. Hacer funcionar el videojuego Checklist en Unity3D.
2. Hacer funcionar el mayor número de juegos desarrollados por CATEDU.
3. Hacer funcionar cualquier juego de eAdventure en Unity3D.
4. Generar un diseño y arquitectura de aplicación de calidad.
5. Integrar este intérprete con el proyecto de Piotr Marszal.
6. Implementar una serie de facilidades que mejoren la interacción en el ámbito táctil.
7. Dar la capacidad a los no desarrolladores, de poder jugar a videojuegos generados con eAdventure en cualquier plataforma, y sin necesidad de tener Unity, así como de beneficiarse de las características de uAdventure.
8. Mejorar la capacidad de evaluar existente en eAdventure.

Acerca del objetivo 1: Este objetivo fue uno de los primeros en completarse, pues tras la construcción del prototipo inicial, el videojuego Checklist, descrito en el apartado 4.2, ya era completamente jugable. Pese a que el resultado obtenido en dicho momento del desarrollo no era, a nivel de arquitectura de proyecto y de representación visual del mismo, un resultado tan bueno como para poder producir dicho juego como juego final, según el proyecto avanzó, se implementaron una serie de mejoras que consiguieron que el juego Checklist pudiese generarse y ser un videojuego de calidad.

Para satisfacer este objetivo fue necesaria la implementación del prototipo inicial del proyecto al completo, presentado en el apartado 9. Realizando una investigación acerca de los paquetes ".jar.[ej]ecutables de los juegos, explicada en el apartado 8. Tras esta investigación, se desarrolló una primera arquitectura del proyecto, y se implementaron las clases necesarias para poder leer el fichero de especificación del juego, y almacenar los datos que

se presentaban en dicho fichero. A estas clases se las conocería como Clases de Datos. Tras esto se generaron clases capaces de interpretar dichas Clases de Datos y representarse en el videojuego, conocidas como Clases de Comportamiento. Finalmente, y debido a la necesidad de funcionalidad adicional, se implementaron clases que daban soporte a efectos y a conversaciones, así como una clase controladora del juego, llamada *Game*, capaz de gestionar la interacción y el estado del juego.

Acerca del objetivo 2: Los juegos de CATEDU son una gran cantidad de juegos, entre los cuales se encuentran el juego de Primeros Auxilios, o el juego de Viaje a Londres. Para dar soporte a este objetivo fue necesaria la realización de la segunda iteración del proyecto, aunque no en su totalidad. En esta segunda iteración se añadieron nuevas características no soportadas en la primera, como la gestión de Temporizadores, explicada en el apartado 10.2.5, la capacidad de mover objetos a lo largo de la pantalla, explicada en el apartado 10.2.3, la posibilidad de que hubiera juegos en tercera persona, explicadas en el apartado 10.2.3, y la capacidad de que, en estos, el jugador pudiera moverse, explicados en el apartado 10.2.3.

Finalmente, para soportar totalmente estos videojuegos fue necesario dar soporte a contenido multimedia audiovisual, como son los videos y el audio. El soporte a estos elementos está explicado en los apartados 10.2.7 y 10.2.3.

Acerca del objetivo 3: Dado que conseguir que literalmente cualquier juego de eAdventure funcione en uAdventure es una tarea compleja, pues requiere de implementar todas y cada una de las características de eAdventure, por el momento y muy lamentablemente, es posible que algunos de los juegos que se intenten hacer funcionar, puedan producir algún fallo por la falta de implementación de alguna de las características de eAdventure. Sin embargo, y para intentar garantizar esta característica, se han implementado prácticamente todos los efectos para las macros, efectos y acciones, explicado en el apartado 9.3.2.

Acerca del objetivo 4: Para satisfacer este objetivo ha sido necesaria la realización de multitud de iteraciones sobre el código del intérprete de uAdventure, mediante la refac-

torización y el continuo diseño de elementos de la aplicación se ha conseguido alcanzar un diseño que se apoya sobre los pilares de la Ingeniería del Software, y senta sus bases para ser lo más extensible posible, replicando la menor cantidad de código posible, basándose en patrones como el patrón *Singleton*, o *Factory*, implementando Gestores y Controladores para la gestión y control de tareas, etc... Toda esta labor de diseño además, está acompañada de diagramas de clase que presentan la arquitectura, diseño de clases e implementación a lo largo de la segunda iteración del proyecto, presentada a lo largo de todo el capítulo 10 de esta memoria.

Acerca del objetivo 5: Para la integración de los proyectos ha sido necesario generar un modelo de datos común para las dos aplicaciones, para que ambos proyectos se construyeran utilizando la misma base. Sin embargo, no sólo el modelo de datos se ha desarrollado, sino que también se desarrolló en conjunto un lector de ficheros de especificación de juegos de eAdventure. Todo esto está explicado en el apartado 10.3. Por otra parte, en el proceso de integración, en este proyecto se generaron dos editores para el editor de uAdventure desarrollado por Piotr. Dichos editores están explicados en el apartado 11.2.

Acerca del objetivo 6: Para mejorar la interacción en el ámbito táctil se da soporte en el proyecto al uso de dos *Shaders*, los cuales permiten hacer que los elementos interactuables de la escena brillen y se hagan visibles para el usuario que no dispone de un cursor de ordenador para desplazarlo a lo largo de la escena para identificar los elementos interactuables. Por otra parte, se recoloca la interfaz de los juegos en algunos determinados momentos, como en las listas de respuestas, donde las respuestas ya no aparecen como pequeñas líneas de texto en la parte inferior de la pantalla, sino que se presentan como un listado de grandes botones cendrados en la pantalla. Estas características se explican en el apartado 10.2.6, así como en el apartado 10.2.6.

Acerca del objetivo 7: Para dar soporte a los no desarrolladores, para que puedan beneficiarse de las ventajas de uAdventure, y poder utilizar cualquier juego de eAdventure en cualquier plataforma, se ha desarrollado un emulador independiente de videojuegos

de eAdventure, capaz de importar juegos en formato ".jar", y permitir al usuario jugarlos. Este emulador está explicado en el apartado 11.1. Este emulador dispone de distintas vistas, entre las que se presenta un explorador de archivos capaz de explorar el sistema de ficheros para permitir al usuario seleccionar el videojuego que desee importar, así como una pantalla principal donde se presentan todos los videojuegos importados, o una pantalla de configuración.

Acerca del objetivo 8: para mejorar la capacidad de evaluar existente en eAdventure, se ha conectado el sistema con RAGE, el cual provee, mediante una interfaz web, una serie de páginas con información importante y gráficas acerca del progreso y evaluación de los alumnos de forma en grupo e individual, permitiendo al profesor enfocar los esfuerzos en aquellos alumnos que hayan quedado rezagados o que demuestren problemas en su aprendizaje. Para ello se ha dado soporte a un proyecto del grupo e-UCM, que consiste en generar un *Tracker* que se comunique con RAGE. Dicha comunicación está explicada en el apartado 10.4. Por otra parte, y para generar los perfiles de evaluación, se ha facilitado el acceso a RAGE desde eAdventure, generando un editor y una serie de clases comunes con RAGE, capaces de almacenar elementos de este sistema. Dicho editor se explica en el apartado 11.3.

Capítulo 14

Conclusions

After completing this project, including the design, implementation and writing of this report, we have identified certain conclusions that are being addressed in the post mortem of this project.certain conclusions to be addressed in this post mortem of the project are extracted. They will be presented categorized within the different objectives that we identified, establishing a relationship between the objectives and the results.

Under my personal impression, I am quite satisfied with the result, therefore, in greater or lesser quantity, I have met the target list that i wrote, learning many new concepts. Through experimentation I have learned that, at least in my case, I produce better projects making a small design on paper, how I imagine things should work, and developing a prototype that implements that design, and lets me see if the result obtained it is the expected, or need to make a redesign of the application.

One conclusion I get from the project is that is very satisfying to work with a foreign student, a different culture, such as Piotr Marszal. The experience of working with a person who takes his job so seriously, which you can order results and responds to your needs, is a very good experience, because you can grow as a person to know another way of working, and makes you involve yourself in the development of the project, developing a code of higher quality.

The simplified list of objectives identified in Chapter 2 is the following: 1. Make the Checklist videogame work in Unity3D . 2. Make the highest number of games developed by

CATEDU work in Unity 3D. 3. Correctly run any eAdventure game in Unity3D . 4. Design a quality application architecture. 5. Integrate the implemented platform with Piotr Marszal project. 6. Implement a set of interaction improvements for mobile devices. 7. Develop a functionality to allow non-developers to be able to play games generated with eAdventure on any platform, without having Unity installed, as well as benefit from the characteristics of uAdventure. 8. Improve the existing eAdventure assessment functionality.

About the first goal: This goal was one of the first to be completed, since after the construction of the initial prototype, the game Checklist, described in section 4.2 was fully playable. Although the results obtained at that time were not representative of a high quality videogame. The quality of the Checklist videogame improved significantly after several iterations.

To satisfy this objective was necessary to implement the initial prototype of whole project, presented in chapter 9. Researching about ".jar" packages and runnable games, explained in Section 3. After this research, we developed the first project architecture, and we implemented the necessary classes read and store the data specified inside the game's specification file. These classes are known as the Data Classes. Afterwards, we generated the Behavior Classes (classes that interpret and run the Data Classes). Finally, we implemented the classes required for additional functionalities such as support for effects, conversations and a controller known as Game, able to manage the game state and operations over its data model.

About Objective 2 : CATEDU is composed of a set of games such as the First Aid Game, or the game Travel to London. We had to iterate a second time over the project, although not entirely. In this second iteration new features were added such as Timer management , explained in section 10.2.5, the ability to move objects across the screen, explained in section 5.2.3, the support for third person games, explained in section 10.2.3 , and the player movement in these types of games, explained in sections 10.2.3, and 10.2.3.

Finally, for give full suport to this videogames, it was necesary to give support to multi-

media content, like videos and audio. Support to multimedia is described in sections 10.2.7 and 10.2.3.

About Objective 3 : Correctly running every eAdventure game in uAdventure is a very complex task because it requires the implementation of all the original eAdventure functionalities. Currently, it is possible that some of the existing games may not be fully runnable on uAdventure. However, to maximize the amount of functionality implemented, we fully support macros, effects and actions, explained in section 9.3.2.

About Objective 4. To complete this objective we had to iterate several times over the uAdventure source code. We accomplished a source code that is flexible and implements several Software Engeneering design patterns such as Singleton or Factory (implementing Managers and Controllers for the operations process). This refactoring and redesign process is supported by class diagrams for the architecture and the class implementation documented in chapter 5 10.

About Objective 5. To be able to integrate both projects it has been necessary to develop a common data model. We have developed the common data model and a file reader designed for the original eAdventure files and explained in section 10.3. During the integration process, we also ccreated two editors that have been explained in section 11.2.

About Objective 6. To improve the interaction functionality, especially designed for mobile devices, we have developed two different shaders. The shaders highlight elements of interest of the scene. This is useful when there is no cursor available on the screen. We also redesigned parts of the user interface to improve it for mobile devices. All these improvements are described in section 10.2.6.

About Objective 7 In order to allow users with no programming skills use the uAdventure framework and be able to run eAdventure games on different platforms we have developed an emulator able to import and play games formatted as ".jar" packages. The emulator consists of a file explorer that allows the user to select the games to be imported. This is fully explained in section 10.4.

About Objective 8 To improve the current assessment functionality and the evaluation support of eAdventure, we have connected the system with RAGE. RAGE provides the analysis infrastructure required to process the progress and other related learning outcomes. We developed the functionality required to manage the data model of RAGE (such as games, classes and sessions) and we used a tracker asset developed by the e-UCM research group to communicate with the RAGE platform, explained in section 11.3

Capítulo 15

Trabajo Futuro

Para mejorar el proyecto, se plantea una pequeña lista de trabajo futuro que garantizará que, si se realiza correctamente, se alcancen los objetivos planteados del proyecto.

Esta lista contiene:

- Mejorar el acabado gráfico, dando soporte a todos los cursores, pues actualmente sólo se soportan el cursor normal, y el cursor de mano cuando existe un elemento interactuable bajo el cursor.
- Dar soporte a todas las transiciones de eAdventure, pues únicamente se ha dado soporte a la transición *Fade* que conecta dos escenas mediante un efecto de degradado.
- Implementar todos los tipos de efectos disponibles, pues, aunque los efectos más utilizados tienen soporte, aún existen determinados efectos que su soporte es nulo o parcial.
- Mejorar el soporte a las trayectorias, así como implementar la necesidad de que el jugador se encuentre cerca de un elemento de la escena para para poder interactúar con el.
- Implementar el inventario, pues el jugador no dispone de inventario de objetos.
- Añadir la posibilidad de cargar y guardar partida utilizando la clase *GameState* y un elemento de serialización.

- Implementar una pantalla intermedia en el emulador para permitir al usuario ver detalles de los juegos intermedios, cargar y borrar partidas, así como la posibilidad de desinstalar un juego importado.

- Mejorar el soporte con RAGE, añadiendo procesos de generación automática de cálculo de progreso y evaluación, pudiendo generar el primero a partir de la sucesión de escenas que conectan la escena inicial y la escena final, o el segundo, a partir de unas variables que determine el usuario.

- Integrar RAGE en el editor de forma transparente, permitiendo seleccionar desde los menús, que cosas se desean monitorizar, realizando un par de pulsaciones en botones.

- Mejorar la documentación dentro del código, pues el código en si mismo no tiene demasiada documentación, y pese a los diagramas de clases generados, una documentación de calidad incrementa la calidad del software.

- Rediseñar el editor generado por Piotr para integrarlo dentro de Unity. Pese a que la interfaz actual permite al usuario una interacción muy similar a la que tenía trabajando con eAdventure, dicha interfaz es muy antigua y puede ser mejorada mediante la integración completa en Unity, creando un editor híbrido.

- Dar soporte a cualquier objeto prefabricado, pudiendo utilizar elementos tridimensionales dentro de la escena.

Sin embargo, este listado de trabajo futuro no es más que un listado de características que mejorarían la experiencia de usuario de uAdventure, por lo que, cualquier característica adicional que se le desee incorporar al editor que mejore esta experiencia puede considerarse como trabajo futuro del proyecto. Conocer este listado de características será determinado por el uso que los usuarios den al proyecto, mediante la obtención de retroalimentación por su parte.

Capítulo 16

Future Work

For improving the project, a small list of future work is proposed. This list ensures that, if it's correctly satisfied, the objetives shown in this project will also get satisfied.

In the list it is:

- Improve the graphic details, giving support to all the cursors and pointers, because currently only regular and hand cursor are supported.
- Give full support to transitions. Currently only fade transition is supported by overlaying two scenes with different opacity. Thats because fade transition is the most common transition.
- Supporting all the available effect types. Although the most used effect have been supported in uAdventure, they still are some effects with null or partial support.
- Improve trajectory support, and implement the need that the player must be near an item in the scene to interact with it.
- Implement inventory because currently player doesn't have one.
- Add the possibility to load an save game using the class *GameState* and a serialization tool.

- Implement a halfway screen in the emulator for showing details about the imported games to the player, and giving some abilities like removing the game or the different saves.

- Improve RAGE support, developing proccess for automatic progress and evaluation calculation. This can be made by using the scene progression that connects the initial and end scenes. Evaluation can be made by letting the developer set a list of variables that define a score.

- Transparently integrate RAGE with the editor allowing to select form the menus what elements the developer wants to monitorize just by clicking a few buttons.

- Improve code documentation, because currently most of the code doesn't have comments or documentation. Although all the class diagrams generated for this document, a better inline documentation improves the software quality.

- Redesign Piotr's editor, for a better integration with Unity. Although the current UI allows the users to interact in a similar way as they are used to do in eAdventure. In any case, That UI is very old and can be improved by Unity integration, creating an hybrid editor.

- Supporting Unity prefabs, allowing the user to add 3D elements to the scene.

But nevertheless, this future work list is nothing but a list of features that will improve user experiencie in uAdventure, so every new feature that improves that experience can be considered as future work. Knowing this feature list will be determined by the feedback users give about the project.

Bibliografía

[1] Miguel Angel Alvarez. CSS: Separar contenido y presentación, 2015.

[2] Roberto Baelo Álvarez. El e-learning, una respuesta educativa a las demandas de las sociedades del siglo XXI. *Pixel-Bit: Revista de medios y educación*, (35):87–96, 2009.

[3] K Beck. Agile Manifesto, 2001.

[4] Béla Bollobás and Oliver Riordan. Dijkstra ' s Algorithm. *Network*, 69(1959):036114, 1993.

[5] Jorge Cano. Análisis de Mass Effect Trilogía para PlayStation 3), 2012.

[6] FFmpeg Community). About FFmpeg, 2016.

[7] The Game Creators. GameGuru, Create, play and share fun games on your PC with absolutely no technical knowledge needed, 2016.

[8] Centro Aragonés de Tecnologías para la Educación (CATEDU). Página principal del Centro Aragonés de Tecnologías para la Educación, 2016.

[9] Alberto de Vega Luna. Runaway: A Road Adventure., 2001.

[10] Michele D. Dickey. Game design narrative for learning: Appropriating adventure game design narrative devices and techniques for the design of interactive learning environments. *Educational Technology Research and Development*, 54(3):245–263, 2006.

[11] Inc Double Fine Productions. Dr. Fred's mutated purple tentacle is about to take over the world, and only you can stop him!, 2016.

[12] e UCM. Checklist (game abouth safe surgery checklist).

[13] Santiago Lamelo Fagilde. Unreal Tournament, El Redenetor, Vuelve Unreal más multijugador que nunca, nuevas armas, nuevos estilos de juego para reforzar de nuevo la acción y adicción del universo Unreal., 2016.

[14] Javier Fugitivo. Game Maker Studio: Primeros pasos, 2013.

[15] Crytek GmbH. CryEngine Features, Achieve your Vision using our cutting-edge, 100royalty-free technology, 2016.

[16] Chris Jones. Adventure Game Studio Manual - Introduction, 2010.

[17] Jonathan Levin. Dalvik and ART.

[18] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java® Virtual Machine Specification*. 2014.

[19] Cykod LLC. Quintus is an easy-to-learn, fun-to-use JavaScript HTML5 game engine for mobile, desktop and beyond, 2013.

[20] Baltasar Fernández Manjón. Improving the Application of the Surgical Checklist Using Serious Games, Games for Health Europe Conference, Amsterdam, NL. 2013.

[21] D R Michael and S L Chen. Serious Games: Games That Educate, Train, and Inform. *Education*, October 31:1–95, 2005.

[22] V I Middleware. Microsoft . NET. *Framework*, pages 65–85, 2003.

[23] Greg Miller. Turok Review for IGN, 2008.

[24] Jonathan Miller. Gears of War review for IGN - One year in, the Xbox 360 has its Killer App. And we mean killer., 2006.

[25] Wikipedia Multiple Authors. BioShock (Series), 2016.

[26] Wikipedia Multiple Authors. Doom, 2016.

[27] Wikipedia Multiple Authors. Monkey Island (Series), 2016.

[28] Wikipedia Multiple Authors. RPG Maker, 2016.

[29] Juan Ignacio Rodriguez Navarro. Introducción al manual de Adventure Game Studio (AGS), 2008.

[30] M Pearcy. Americas Army: "Playful hatredïn the social studies classroom. *International Journal of Gaming and Computer-Mediated Simulations*, 4(2):19–36, 2012.

[31] Laura Arjona Reina and Gregorio Robles. Mining for localization in Android. *IEEE International Working Conference on Mining Software Repositories*, pages 136–139, 2012.

[32] Technical Report. Analysis of Dalvik Virtual Machine and Class Path Library. *Management*, pages 1 – 42, 2009.

[33] Dead:Code Software. Some of the notable WME features, 2010.

[34] Esoteric Software. Spine is 2D animation, Spine website, 2016.

[35] Hungry Software and contributors. Introduction to SLUDGE Concepts, 2002.

[36] Christina Steiner, Alexander Nussbaumer, Eric Kluijfhout, Rob Nadolski, Samuel Mscarenhas, Moreno Ger, Mihai Dascalu, Stefan Trausan, and Christina Steiner. Realising an Applied Gaming Eco-system D8 . 1 – RAGE Evaluation Framework and Guidelines Name of Author. (644187):1–86, 2016.

[37] Unity Technologies. Manual: Execution Order of Event Functions (Orden de Ejecución de Funciones de Evento).

[38] Unity Technologies. Create games, connect with your audience, and achieve success - Unity 3D, 2016.

[39] Mike Thomsen. History of the Unreal engine, 2010.

[40] Javier Torrente, Ángel Del Blanco, Eugenio J. Marchiori, Pablo Moreno-Ger, and Baltasar Fernández-Manjón. e-Adventure: Introducing Educational Games in the Learning Process. In *IEEE Education Engineering (EDUCON) 2010 Conference*, pages 1121–1126, Madrid, Spain, 2010. IEEE.

[41] Javier Torrente, Ángel Serrano-laguna, Conor Fisk, Breid O Brien, Wanda Aleksy, Baltasar Fernández Manjón, and Patty Kostkova. Introducing Mokap: a novel approach to creating serious games. *5th International Conference on Digital Health*, 1(1):17–24, 2015.

[42] Yolanda Torres Molina. Localización de juegos para móvil. *Tradumàtica: traducció i tecnologies de la informació i la comunicació*, (5), 2007.

[43] Trent Ward. Quake Review - If you're into action games, or even if you're not, you should be playing Quake right now - it's as good as PC gaming gets., 1996.