Ejecución y adaptación de trazas de juegos para la automatización de pruebas

Trabajo de Fin de Grado en el
Doble Grado en Ingeniería Informática y Matemáticas

Luis María Costero Valero

Dirigido por el Doctor
Pedro Pablo Gómez Martín

Departamento de Ingeniería del Software e Inteligencia Artificial
Facultad de Informática
Universidad Complutense de Madrid

Junio 2015

Documento maquetado con TEXIS v.1.0+.

Este documento está preparado para ser imprimido a doble cara.

Ejecución y adaptación de trazas de juegos para la automatización de pruebas

Trabajo de Fin de Grado realizado por
Luis María Costero Valero

Dirigida por el Doctor
Pedro Pablo Gómez Martín

Departamento de Ingeniería del Software e Inteligencia Artificial
Facultad de Informática
Universidad Complutense de Madrid

Junio 2015

A mis padres

Aprender sin reflexionar,
es malgastar la energía
– Confucio –

Agradecimientos

Hace cinco años que empecé este doble grado que hoy termina con está memoria, y durante todo este tiempo ha habido personas en mi camino a las que quiero darles las gracias.

En primer lugar a mi familia, que siempre han estado ahí dándome ánimos y nunca han dudado en que lo conseguiría. Gracias en especial a mis padres, que ya desde muy pequeño me pusieron delante de un ordenador y han hecho que acabe donde estoy ahora.

En segundo lugar a todos los profesores que he tenido durante estos cinco años, por enseñarme casi todo lo que ahora sé. En especial a Pedro Pablo que ha aguantado todas las visitas inesperadas a su despacho, y a su hermano Marco Antonio, que gastó sus tardes en enseñarnos cosas que de otra manera nunca hubiera aprendido.

Agradecer también a todos los compañeros que he tenido durante estos cinco años, que han hecho que todos estos años no sean tan empinados como podrían haber sido.

Y por último, pero no menos importante, a Jenny, que sabe sacar lo mejor de mí y siempre da ideas, que aunque parezcan estupideces, acaban convirtiéndose en caballos ganadores.

Gracias a todos.

Resumen

En todo hombre hay escondido
un niño que quiere jugar

Friedrich W. Nietzsche

El *testing* es una de las fases más importantes en el proceso de desarrollo de un videojuego ya que permite construir un juego de calidad y libre de errores. Sin embargo, características propias de un videojuego como el diseño de niveles, los comportamientos del juego frente a diferentes eventos o la experiencia del usuario no pueden ser comprobadas por herramientas y métodos de testing tradicional.

En este trabajo se propone una nueva forma de testing que intenta solventar alguna de estas carencias. El método propuesto se basa en imitar los movimientos de un jugador experto previamente grabados. Estos movimientos grabados son adaptados a las diferentes modificaciones que se han podido realizar sobre un nivel del juego, con el objetivo de intentar superar el nivel.

Para conseguir este objetivo se hace uso de la arquitectura con la que son diseñados los videojuegos, siendo capaces de capturar todos los eventos que ocurren sin necesidad de que el juego esté diseñado específicamente para este propósito. Gracias a esto, a parte de poder realizar la grabación y posterior reproducción de trazas, se consigue unos test muchos más completos, basados en la detección de acciones que ocurren durante el juego, y no solo en detectar si el nivel ha sido superado o no.

El sistema propuesto ha sido implementado sobre un videojuego real llamado `Time & Space` (Blázquez et al., 2013-2014), creado como proyecto de fin de máster del máster de Desarrollo de Videojuegos de la UCM.

PALABRAS CLAVE: arquitectura de videojuegos, reproducción automática, testing, grabación de trazas, videojuego.

Abstract

In every real man a child is
hidden that wants to play

Frederich Nietzsche

Testing is one of the most important phases in the videogame development process, allowing to build a quality game free of errors. However, specific videogame characteristics like level design, the behaviour of the elements of the videogame when some events happen or player experiences cannot be tested with tools and methods of traditional testing.

Here we propose a new method of testing with the purpose of solving some of this lacks. This method is based on the imitation of the movements of a human player that have been recorded before. This movements are adapted to the different modifications that can be done in the map in order to reach the end of the level.

In order to reach this goal, we take advantage of the game architecture design to read all the events that happen during the gameplay without the necessity of specific methods or classes design. In addition, this allows us to create complete tests, based on the detection of the events during the game, not only if the end of the level is reached or not.

Also, this new testing method has been implemented in a real videogame called `Time & Space` (Blázquez et al., 2013-2014), a videogame created as final project for the Máster de Desarrollo de Videojuegos de la UCM.

KEYWORDS: automatic replay, game architecture, testing, trace recording, videogame.

Índice

Índice de figuras

Índice de Tablas

Capítulo 1

Introducción

Según han ido evolucionando los distintos modelos de desarrollo del software, la importancia de incorporar una fase de *testing* a estos ha quedado más que demostrada. No solo es una herramienta diseñada para buscar errores en la implementación y abaratar costes, sino que cada vez se utiliza para más fines, como puede ser comprobar la compatibilidad en distintas arquitecturas, o medir los tiempos de carga de la aplicación.

Sin embargo, como se verá más tarde en la sección 2.4, las herramientas de testing tradicional desarrolladas no son totalmente válidas para el proceso de desarrollo de un videojuego. Un juego posee ciertas características que lo hacen diferente de cualquier otro software, como una parte de jugabilidad, compuesta por niveles, mapas, retos propuestos al jugador, libertad de movimiento, etc.

Un ejemplo claro es si se cambia de posición de una puerta en un mapa, la cual tiene que atravesar el jugador para superar el nivel. La implementación del juego no ha cambiado, y este puede ejecutarse sin ningún problema, por lo que los test tradicionales no detectarían ningún fallo en el juego. Sin embargo, cambiar la puerta de sitio puede suponer que el jugador ya no sea capaz de superar el nivel, o que la secuencias de acciones que tiene que realizar para superarlo sea totalmente distinta a la diseñada originalmente.

Actualmente, para comprobar que no existen problemas de este tipo, jugadores humanos reproducen el juego y repiten el nivel comprobando que todo sea correcto. Esto, aunque efectivo, es una tarea costosa al tener que dedicar tiempo y recursos para jugar el mismo nivel una y otra vez.

En este trabajo se presenta una nueva forma de testing, basada en imitar los movimientos de un jugador que previamente han sido grabados. Estos movimientos se adaptan a las distintas modificaciones del mapa que han sido introducidas. Al reproducir las trazas, se captura en vivo lo que ocurre en el juego, pudiendo así establecer condiciones de éxito y de fracaso complejas, más allá de detectar si se ha superado el nivel o no.

El sistema propuesto ha sido desarrollado con el objetivo de ser lo menos

intrusivo al programador, evitando que este tenga que modificar en exceso el videojuego. Para ello, nos hemos centrado en una arquitectura concreta basada en componentes con paso de mensajes descrita en la sección 3.4.2. Así mismo, este sistema ha sido implementado en `Time & Space` (Blázquez et al.), un videojuego real creado en el Máster de Desarrollo de Videojuegos[1] de la Universidad Complutense de Madrid.

Este sistema, explicado en los capítulos 4 y 5, se compone de los siguientes pasos:

1. *Generación de trazas.* Mientras el jugador experto juega a cierto nivel, el sistema almacena la traza generada en un archivo de texto de forma totalmente transparente a este.

2. *Generación de test.* El encargado del test genera los archivos de configuración sobre los distintos test automáticos que desea ejecutar. En la implementación sobre `Time & Space`, esta configuración se realiza sobre archivos de texto con formato Json.

3. *Adaptación y reproducción de trazas.* Los test programados se lanzan de manera automática utilizando las trazas grabadas anteriormente. Estas se adaptan al nivel actual mientras son reproducidas.

4. *Detección de objetivos.* Mientras las trazas son reproducidas, el sistema de testing detecta todas las acciones que se producen en el juego, comparando con los objetivos marcados por el creador del test. Al final del test produce un informe con lo ocurrido. (Éxito o fracaso, tiempo empleado y porcentaje de progreso de cada test).

1.1. Trabajo relacionado

Durante mucho tiempo se han intentado crear sistemas inteligentes capaces de aprender a jugar a videojuegos a partir de trazas creadas por jugadores expertos, imitándoles o adaptando sus movimientos. El funcionamiento general de estos sistemas comienza por la grabación de trazas generadas por el jugador. Estas trazas son anotadas por jugadores expertos, para después poder ser adaptadas antes de jugar. Algunos artículos que recogen estas ideas son:

- En Mehta et al. (2009) se muestra un esfuerzo por diseñar un sistema de grabación de trazas lo más genérico posible para que pueda ser anotado por expertos. Sin embargo, el sistema se diseña para que los nuevos videojuegos desarrollados se adapten a él, no al revés como sería lo ideal.

[1] `http://www.videojuegos-ucm.es`

- En Ontañón et al. (2009) se muestra como adaptar las trazas capturadas para poder reproducirlas más tarde. Describen como detectar objetivos y subobjetivos, detectar planes e utilizar redes de Petri para planificar los movimientos del jugador.

Con el crecimiento del tamaño y complejidad del software, realizar y mantener los diferentes test de una aplicación es cada vez más difícil y costoso. Además, probar todas las funcionalidades de un software muchas veces es casi imposible. Un claro ejemplo es el testing aplicado al desarrollo de videojuegos multijugador masivos online. Como se expone en Mellon (2006), un número masivo de jugadores online hace imposible predecir y detectar todos los fallos en el juego. Así mismo, proponen algunas ideas de como aplicar testing a estos casos, intentando simular acciones de los jugadores en un entorno controlado.

Actualmente existen muchos esfuerzos en crear herramientas de testing adaptadas a las nuevas tecnologías de desarrollo. Por ejemplo, el *Monkey testing*[2] consiste en un test de caja negra diseñado para probar aplicaciones en dispositivos Android con interfaz gráfica. Este se basa en introducir flujos de datos aleatorios a la aplicación, y esperar que falle con alguna de estas entrada. Aunque estos test se ejecuten sin ningún objetivo en concreto, es muy útil para detectar bugs escondidos.

Así mismo, el trabajo aquí descrito ha sido aceptado para su presentación en el II Congreso organizado por SECiVi[3]

1.2. Estructura del documento

A continuación se muestra la secuencia de capítulos elegida para tratar los diferentes temas desarrollados. Se comienza tratando temas de testing tradicional y de como éste no es del todo válido para videojuegos. Continúa mencionando las distintas arquitecturas con las que se puede llegar a implementar el motor de un juego. A continuación muestra el sistema de testing diseñado e implementado sobre un videojuego real. Finaliza con unas conclusiones y el trabajo futuro que habría que realizar si se desea continuar en la línea de este trabajo.

Capítulo 2: *Testing.* En este capítulo se presenta que es el testing, la importancia que este tiene en el desarrollo del software, y como ha ido evolucionando a lo largo el tiempo. También se presentan distintos tipos de testing utilizados tradicionalmente. A continuación se mencionan distintos tipos de testing usados en el desarrollo de los videojuegos. Finalmente de describe los problemas que presenta aplicar técnicas de

[2] `http://developer.android.com/tools/help/monkey.html`
[3] SECiVi: Sociedad Española para las Ciencias del Videojuego

testing tradicional a los videojuegos, y la necesidad de crear nuevas técnicas.

Capítulo 3: *Arquitectura de videojuegos.* En este capítulo se presenta una idea general de cómo está implementado un videojuego. En una primera sección se muestra cómo un videojuego está gobernado por una máquina de estados, que dependiendo del estado actual se comportará de una manera o de otra. En secciones sucesivas se muestra dos formas de organizar los distintos elementos que forman el juego, y se dan razones para utilizar la segunda frente a la primera. El capítulo acaba introduciendo la forma de comunicación existente entre los distintos componentes del juego.

Capítulo 4: *Diseño de un sistema de testing para videojuegos.* En este capítulo se muestra la arquitectura general del sistema de testing. Se detalla qué datos se recogen y cómo se realiza. Así mismo, se explica la estructura modular del sistema. El capítulo finaliza describiendo los ficheros de configuración de los test, y los campos que estos deberían tener.

Capítulo 5: *Un videojuego real: Time & Space.* Mientras que en el capítulo 4 se detalló en que consiste el nuevo sistema de testing propuesto, las partes que lo componen, y como integrar el sistema en un videojuego, en este capítulo se muestra la implementación sobre un videojuego real, producto del máster en desarrollo de videojuegos. También se muestra el formato elegido para escribir los test, y la forma en la que los resultados son mostrados.

Capítulo 6: *Conclusiones y trabajo futuro.* En este capítulo se hace un mínimo recorrido a través de todos los capítulos anteriores, recalcando los detalles más importantes de cada uno de ellos. Se muestra como más allá de ser una idea teórica, el nuevo método de testing tiene una utilidad real. El capítulo finaliza comentando posibles ampliaciones del trabajo o la creación de nuevas herramientas que faciliten la tarea de crear nuevos test.

Capítulo 2

Testing

Beware of bugs in the above code; I have only proved it correct, not tried it.

Donald Knuth

Resumen: En este capítulo se presenta que es el testing, la importancia que este tiene en el desarrollo del software, y como ha ido evolucionando a lo largo el tiempo. También se presentan distintos tipos de testing utilizados tradicionalmente. A continuación se mencionan distintos tipos de testing usados en el desarrollo de los videojuegos. La sección finaliza describiendo los problemas que presenta aplicar técnicas de testing tradicional a los videojuegos, y la necesidad de crear nuevas técnicas.

2.1. ¿Qué es el testing?

Aunque las primeras computadoras digitales aparecen en la década de 1940, no es hasta finales de los años 50 principios de los 60 cuando surgen los primeros lenguajes de programación generales como *Fortran (1957)*, *ALGOL (1958)* o *COBOL (1959)*. Junto a estos lenguajes, el software demandado comienza a ser cada vez más grande y complejo, lo que implica que aparezcan retrasos inesperados, sobrecostes no planificados o graves errores entregados al cliente en el producto final. Para intentar evitar todas estos problemas no deseados, se vio la necesidad de crear métodos, métricas y herramientas para estructurar el proceso de desarrollo y otorgarle de etapas y procedimientos a realizar para conseguir un software de calidad, adaptado a las necesidades del cliente y eliminando los retrasos y costes inesperados.

Aparte de la necesidad de estructurar el proceso de desarrollo en diferentes etapas, se dotó al proceso de una nueva herramienta o fase denominada

testing, encargada de comprobar que el software creado cumple los requisitos deseados y se encuentra libre de errores. Con el paso de los años esta fase ha ido evolucionando tanto en el lugar que ocupa dentro del proceso de desarrollo, como en los métodos que se utilizan o los objetivos que persigue. En los primeros modelos de desarrollo propuestos (como puede ser el modelo en cascada propuesto por Royce en el año 1970) la fase de testing ocupa los últimos lugares en el proceso con el objetivo de revisar el trabajo antes de entregárselo al cliente. Si un error es detectado, se debe retroceder a etapas anteriores para solucionar el problema y continuar el proceso desde este punto. Si el problema detectado es leve, este no supone ningún problema añadido, ya que volver al punto anterior para solucionarlo y continuar desde ahí no debería suponer mucho esfuerzo ni muchos cambios en el código. Sin embargo, si el problema es grave, al detectarse en las últimas fases del desarrollo puede suponer un cambio importante en todo el proceso, provocando la aparición de retrasos y costes adicionales no deseados. En la tabla 2.1 se muestran los costes de solucionar distintos tipos de errores según en que fase del proceso se detecten, basados en los datos publicados en McConnell (2004)

		Fase en la que se detecta el error			
		Diseño	**Codificación**	**Testing**	**Final**
Error en	**Requisitos**	3x	5-10x	10x	10-100x
	Diseño	1x	10x	15x	25-100x
	Codificación	-	1x	10x	10-25x

Tabla 2.1: Coste al solucionar diferentes tipos de errores según en que fase del desarrollo son detectados

Viendo que muchos de estos problemas añadidos se podrían haber evitado si se hubieran detectado los errores en fases más tempranas del desarrollo, con el paso del tiempo los nuevos modelos de desarrollo (como puede ser el modelo en espiral propuesto en (Boehm, 1986) o las metodologías ágiles surgidas a mediados de los 90) evolucionan considerando la fase de testing en posiciones más tempranas del modelo. A su vez, el proceso se divide en varias iteraciones, cada una de ellas con su fase de testing correspondiente. Gracias a ello, muchos errores que hubieran sido graves si se hubieran detectado en etapas tardías, se solucionan de manera sencilla al ser detectados en una etapa más temprana.

Sin embargo, no es hasta 1998 cuando se estandariza el proceso de testing. Según esta definición, propuesta en IEEE Std 829, (Def. 3.9), se entiende por testing al proceso de analizar un elemento del software con el objetivo de detectar diferencias entre los requisitos deseados y los que este satisface actualmente (es decir, encontrar fallos o *bugs*), o para evaluar ciertas características del software como puede ser el rendimiento, los tiempos de carga,

el número de peticiones simultáneas que la aplicación puede aceptar, la seguridad que posee, etc. Es decir, los objetivos del testing se pueden resumir (pero no limitar) a los siguientes puntos:

1. Comprobar si el software desarrollado verifica los distintos requisitos esperados de él cuando fue diseñado.
2. Verificar si el software funciona correctamente bajo diferentes casos de prueba: ejecutar la aplicación con distintas entradas, cada una intentando satisfacer un aspecto diferente de la aplicación y comprobar que la salida resultante coincide con la esperada.
3. Comprobar que la ejecución de la aplicación para entradas específicas no sobrepasa un tiempo razonable para esa entrada.
4. Comprobar si el software puede ser instalado y ejecutado en todos los entornos para los que ha sido diseñado.
5. Comprobar si la aplicación puede ser usada correctamente por el usuario final para el que ha sido diseñado.

Esta definición pone de manifiesto que el proceso de testing no solo trata de encontrar errores, sino también de detectar ciertas características que el software desarrollado debería poseer. Es por ello que existen multitud de tipos de test, cada uno con unos objetivos y características diferentes. Los tipos más importantes son descritos en la siguiente sección.

2.2. Tipos de testing

Como los objetivos del testing son muy diversos, existen multitud de tipos de test, cada uno diseñado con un propósito distinto. Según el criterio que se desee utilizar para clasificar los distintos test, se pueden distinguir distintas taxonomías. Myers (2004), clasifica los test en las siguientes categorías:

- Atendiendo a la *finalidad* del test. Se distinguen test de corrección, test de rendimiento, test de fiabilidad y test de seguridad.
- Atendiendo a la *fase del proceso* del desarrollo en la que se aplica el test.
- Atendiendo al *ámbito* que contempla el test. Pueden ser test de unidad, test de componentes, test de integración y test de sistema.

2.2.1. Test de corrección

Son los primeros test creados, usados para comprobar si el software desarrollado es correcto, es decir, verifica los requisitos para los que ha sido diseñado. Los test de corrección se pueden clasificar en dos categorías:

- *De caja negra.* Son test creados sin saber la implementación del software, utilizando únicamente la especificación de los requisitos de este. El problema de estos test radica en que no es posible comprobar el comportamiento de la aplicación para todos los diferentes datos que puede manejar, por lo que se necesitan test amplios que intenten abarcar la mayoría de casos posibles. Se distinguen test dirigidos por datos (si la aplicación tiene que tratar estos), test dirigidos por entrada/salida (a partir de cierta entrada se espera cierta salida) o test basados en requisitos (comprobar que el software cumple los requisitos pedidos, sin entrar en la implementación de estos).

- *De caja blanca.* Al contrario que en los test de caja negra, el encargado del test si conoce la implementación de este. Los test son diseñados atendiendo a la implementación de la aplicación, al lenguaje usado, algoritmos implementados, ... Se puede comprobar si el flujo del programa es correcto, si todas las excepciones son capturadas, si se contemplan errores como divisiones entre 0, etc.

2.2.2. Test de unidad

Se denomina *test de unidad* a una pieza de código encargada de probar una unidad básica de una aplicación. Una unidad básica puede referirse desde una función, hasta una clase o un módulo completo del software. El tamaño de la unidad dependerá de quién o quiénes son los encargados de realizar la funcionalidad que se desea poner a prueba. Estos test ejecutan estas unidades de la aplicación bajo un estado concreto del programa (simulado por el test), y comprueba que su resultado final sea el esperado.

Aunque diseñar y crear un conjunto de test de unidad que contemple la mayoría de casos posibles es una tarea compleja, son ampliamente usados en la actualidad ya que posee una serie de beneficios que compensa el tiempo invertido en crearlos:

- *Detección temprana de problemas.* Al poder ser ejecutados los test a la vez que se desarrolla el código, los errores que puede haber son detectados y corregidos antes de dar por terminado esa unidad del software. A su vez, el programador al escribir los test es consciente de las restricciones o errores que puede haber en el código, estando prevenido para no cometer estos errores. En muchos modelos de desarrollo modernos, los test de unidad son creados antes de empezar a codificar el software. De esta manera intentan que el software se adapte a estos test y evitar posibles errores desagradables. A esta práctica se la conoce por *desarrollo guiado por test.*

- *Facilidad para el cambio.* Puesto que los test verifican el correcto funcionamiento de una unidad invocándola y recreando un estado, el pro-

gramador puede cambiar el funcionamiento interno de la unidad sin necesidad de cambiar los test.

- *Sirven de documentación.* A través de los test de unidad se puede observar la forma de interactuar con una pieza de la aplicación en concreto, los datos que ésta espera de entrada, etc. Esto puede servir de documentación a otro miembro del equipo que desee interactuar con esta pieza del software.

2.2.2.1. Estructura general de los test de unidad

Los test de unidad, al estar diseñados para probar ciertas unidades del código, la implementación de estos no debería considerar el uso de otras piezas del código no necesarias para el test. Además, como el objetivo es comprobar unidades del código, los test no deberían ser demasiado extensos. Para simplicidad de los test, estos deben devolver un valor booleano indicando si se ha superado el test con éxito o ha producido algún error inesperado.

En la figura 2.1 se muestra un ejemplo de test de unidad en Java. Dada una clase que suma dos números llamada `AdderImpl`, la cual posee el tipo `Adder`, se desea comprobar que es capaz de sumar dos número de distinto signo. Como se ha mencionado, el test solo utiliza los recursos que se quieren probar y ningún otro (en este caso `AdderImpl`), y además devuelve un valor booleano como resultado del test.

```
1  // ¿puede sumar números positivos y negativos?
2  public void testSumPositiveAndNegative() {
3    Adder adder = new AdderImpl();
4    assert(adder.add(-1, 1) == 0);
5  }
```

Figura 2.1: Ejemplo de un test de unidad para un sumador en Java

Hay que notar que el ejemplo mostrado no comprueba en su totalidad si la clase es capaz de sumar números positivos y negativos. Aunque la única forma de comprobar el correcto funcionamiento sería probar todos los casos posibles (cosa que no es objetivo de los test de unidad), faltaría por comprobar que ocurre si el primer argumento es un número positivo y el segundo es un número negativo (al revés que en el ejemplo). Este ejemplo tan sencillo muestra la dificultad de crear test unitarios y la gran cantidad de recursos que hay que invertir para su creación. Además de la dificultad de crear los test, existe otra dificultad añadida: si la implementación de la clase cambia, los test hay que mantenerlos actualizados. Por ejemplo, si se decide que la clase `AdderImpl` ya no va a recibir dos números, sino un array de números que va a sumar entre ellos, los test habría que cambiarlos para que se invoque

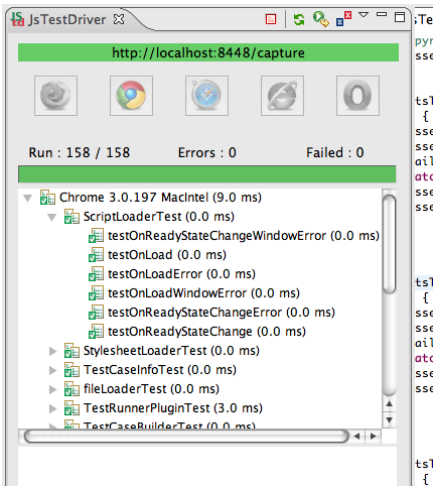


Figura 2.2: Test de unidad integrados en el IDE eclipse.
Fuente: https://code.google.com/p/js-test-driver/

de la nueva manera. Además, habría que añadir nuevos test para comprobar las nuevas funcionalidades, y probablemente eliminar test que ya no tengan sentido con la nueva implementación.

Debido a la sencillez de los test de unidad, en especial a que devuelven valor un booleano indicando si se ha superado el test o no, existen muchas soluciones visuales integradas en distintos entornos de programación que muestran de manera gráfica y a través de colores el resultado de los test ejecutados. En la figura 2.2 se muestra una solución de test de unidad integrada en eclipse, la cual muestra en forma de colores verde/rojo el resultado de los diferentes test. Así mismo, existen otros frameworks que muestran el resultado en formato de un fichero de texto con un log de la ejecución de los test.

2.3. Testing en videojuegos

En una versión muy simplificada, un videojuego es un software de gran tamaño encargado de gestionar distintos recursos y las relaciones que existen entre ellos con el objetivo de crear una experiencia agradable para el usuario final. Y como todo software, y en especial por ser de gran tamaño, es necesario la incorporación del testing al desarrollo de este con el objetivo de evitar retrasos y crear un juego de calidad con el que los usuarios disfruten de la experiencia al jugarlo.

Una de las principales herramientas de testing usadas son los test de unidad, que garantizan que cada elemento del juego se encuentra libre de errores, e incluso que la forma de relacionarse entre los distintos componen-

tes del juego funciona correctamente bajo ciertos estados simulados por el encargado del test. Mientras que los test de unidad solamente comprueban partes del código, el *beta testing* se encarga de comprobar el juego en su totalidad. Éste consiste en que un gran número de jugadores humanos juegan al videojuego en una etapa previa a su lanzamiento y detectan y reportan los distintos errores que van encontrando. Además, los beta-testers son capaces de buscar problemas relacionados con la experiencia de jugar el juego, cosa que los test automáticos como los test de unidad no son capaces evaluar.

Además, los videojuegos diseñados para ser jugados en múltiples arquitecturas hardware (como pueden ser los videojuegos para pc, o los videojuegos para móviles) presentan un problema adicional frente al resto de videojuegos, y es que tienen que poder ejecutarse sobre arquitecturas que poseen diferentes requisitos y características. Por ello se utilizan los llamados *test de compatibilidad*. Los test de compatibilidad ejecutan el videojuego sobre distintas arquitecturas comprobando que no existe ningún error, como puede ser problemas con los gráficos, el sonido, la velocidad de carga, componentes no compatibles, etc.

Cuando se produce un gran cambio en la implementación de algún elemento del software, es necesario comprobar que la nueva implementación se encuentra libre de errores que ya habían sido solucionados anteriormente. Para ello se utilizan los *test de regresión*. Vuelven a ejecutar la aplicación comprobando que los distintos errores que habían sido solucionados en el pasado no han vuelto a aparecer con la nueva implementación.

2.4. Retos del testing tradicional en videojuegos

En el desarrollo de un videojuego, al igual que cualquier otro software de gran tamaño, introducir herramientas de testing facilita el desarrollo de este ya que evita o detecta errores que en caso de llegar a producirse ponen en riesgo el desarrollo del juego. La mayoría de estas herramientas de testing son métodos tradicionales adaptados a las características propias de un software de este tipo. Sin embargo, un videojuego difiere en muchos aspectos de cualquier otra aplicación de gran tamaño, lo que hace que el testing tradicional no sea totalmente válido para el desarrollo de un videojuego. Algunos de estos problemas presentados son:

- *Dificultad de crear y mantener los test unitarios.* Un videojuego, al tratar una gran cantidad de recursos y ser un software tan grande, hace que crear test para comprobar que se encuentra libre de errores supone un esfuerzo considerable. Una vez creados, estos test tiene que actualizarse frente a los múltiples cambios que pueda sufrir la implementación. Así mismo, recrear ciertos estados del videojuego en un test puede ser una tarea difícil o imposible.

- *No determinismo.* Los videojuegos están sujetos a la entrada que puede producir un jugador, luego es imposible predecir todos los estados posibles en los que se puede encontrar el juego y simularlos.

- *Juegos multijugador.* Como se expone en Mellon (2006), los test tradicionales no están diseñados para probar juegos en los que la entrada está formada por múltiples jugadores a la vez, y como estos pueden llegar a interactuar entre ellos.

- *Beta-testing.* Un juego difiere en muchos aspectos de un software tradicional. Actualmente es necesario contratar a jugadores para que prueben el videojuego y detecten errores. Estos son los únicos que pueden detectar problemas asociados con la experiencia de jugar el juego, ya que los test automáticos son incapaces de medirla.

- *Fuerte dependencia de datos.* Con el objetivo de que el videojuego sea modular y puedan trabajar varios equipos de manera simultánea y sin interferencias, muchas características como los gráficos, los mapas o el diseño de niveles se extrae de la implementación del videojuego y se considera como archivos externos a esta. Cambios que afecten a estos datos pero no a la implementación del juego (como puede ser un cambio en un nivel, el nivel de vida de un enemigo o alguna otra propiedad de un elemento que componen el juego) tienen que ser probados por un jugador humano ya que los test tradicionales están diseñados para buscar errores en la implementación, no en los aspectos que surgen durante la ejecución.

- *Salidas difíciles de comparar.* Existen muchas características de un videojuego que son difíciles de comparar por un test automático. Por ejemplo, comprobar si las imágenes mostradas son las correctas, si se escucha el sonido esperado en los momentos debidos, si la velocidad de respuesta es la adecuada o si la sensación física que se obtiene al reaccionar los distintos elementos del videojuego es la esperada. Para comprobar estas cosas, es necesario invertir recursos en un jugador humano que detecte los posibles fallos de este tipo que pudieran haber.

Conclusiones

En este capítulo se ha presentado la importancia del testing en el proceso de desarrollo acompañado de una introducción a los distintos tipos de testing que existen. Sin embargo, se ha mostrado como los tipos de testing utilizados en la actualidad para el desarrollo de videojuegos no son del todo válidos ya que no son capaces de comprobar todos los aspectos del juego, teniendo que ser comprobados por un jugador humano.

Para mostrar el sistema de testing desarrollado que intenta solucionar alguno de estos problemas, en el capítulo siguiente se describe la arquitectura con la que está implementada un videojuego, y posteriormente en el capítulo 4 se muestra el nuevo sistema de testing, el cual se aprovecha de la arquitectura descrita para capturar todos los eventos que ocurren durante el juego.

Capítulo 3

Arquitectura de videojuegos

With good program architecture
debugging is a breeze, because bugs will
be were they should be.

David May

Resumen: En este capítulo se presenta una idea general de cómo está implementado un videojuego. En una primera sección se muestra cómo un videojuego está gobernado por una máquina de estados, que dependiendo del estado actual se comportará de una manera o de otra. En secciones sucesivas se muestra dos formas de organizar los distintos elementos que forman el juego, y se dan razones para utilizar la segunda frente a la primera. El capítulo acaba introduciendo la forma de comunicación existente entre los distintos componentes del juego.

3.1. Introducción

Un videojuego está formado por una gran cantidad de sistemas y recursos diferentes que cooperan entre ellos para hacerlo funcionar, desde drivers específicos integrados con el sistema operativo, pasando por librerías del lenguaje, hasta módulos de dibujado 3D o control de las distintas partes del juego. En la figura 3.1 se muestra un esquema general de todos los componentes que forman un videojuego. Aunque el esquema mostrado no esté completo, da una clara idea del gran tamaño que tiene un videojuego y de la gran cantidad de recursos que hay que invertir para crear un juego serio. Es por ello, que intentar reducir los costes de desarrollo con la detección temprana de errores es uno de los temas más importantes para el desarrollo de un videojuego.

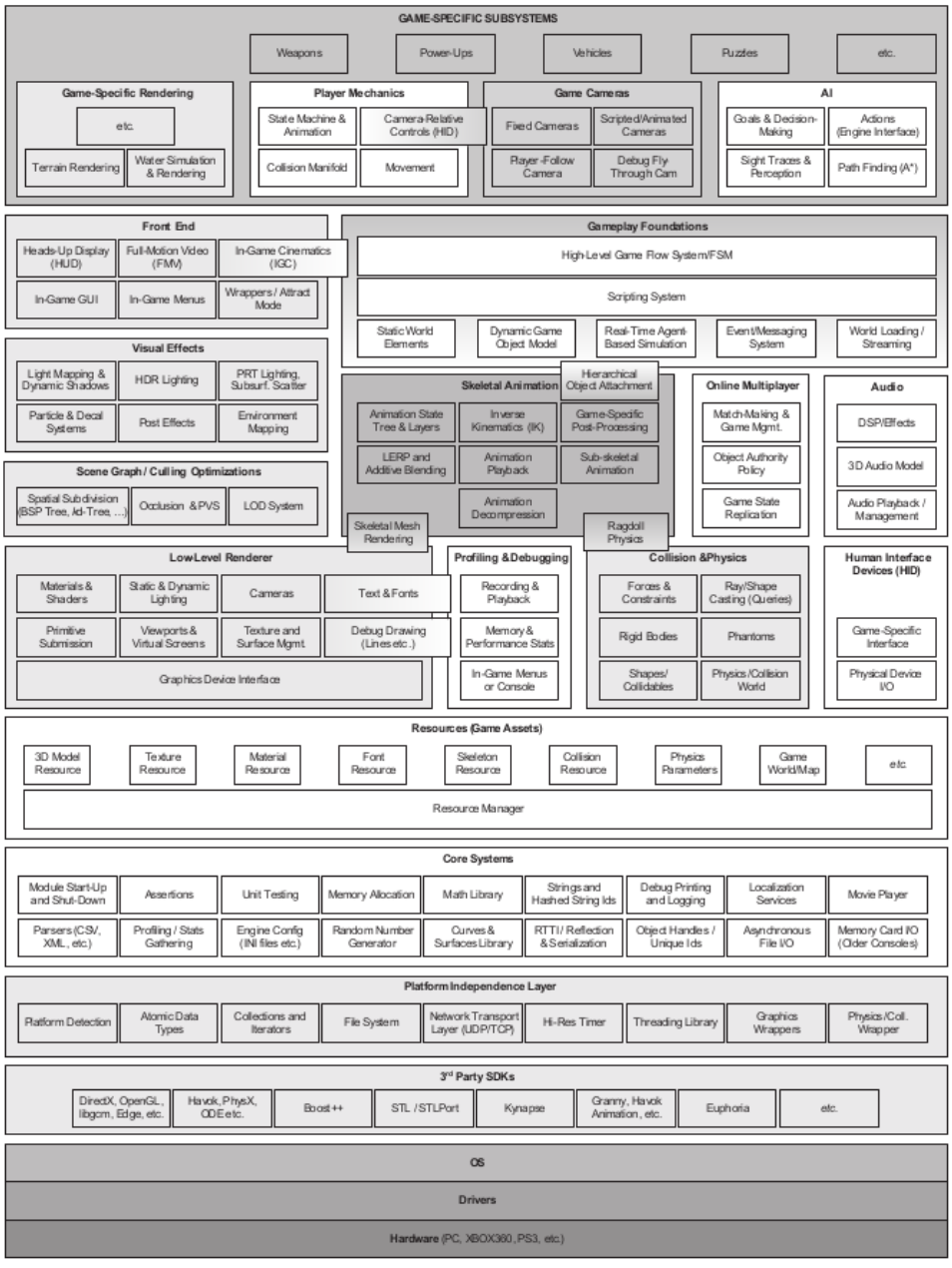


Figura 3.1: Estructura general de los elementos que forman un videojuego. *Fuente: (Gregory, 2009, p. 29)*

En la estructura de un videojuego se pueden distinguir dos secciones diferenciadas en varios aspectos. La primera corresponde a la tecnología, representada en la mitad inferior de la figura. La componen los sistemas de integración con el hardware, comunicación con el sistema operativo, librerías esenciales del lenguaje, dibujado 3D, salida de audio, etc. La segunda sección, representada en la parte superior, corresponde a los elementos que proporcionan la experiencia del juego, como puede ser la IA, los mapas y niveles, el sistema de scripting, los efectos visuales, las texturas, etc. La diferenciación de estas dos secciones no solo corresponde a un nivel de implementación, sino que también se diferencian a nivel de la fase de testing. Así, mientras que en la parte de tecnología los test tradicionales son perfectamente válidos y capaces de detectar la mayoría de posibles fallos, es en la segunda sección donde fallan los test tradicionales como se ha mostrado en 2.4. Es por ello que el nuevo sistema de testing propuesto en este trabajo se enfoca en detectar los problemas relacionados con aspectos de esta segunda parte relacionada con la jugabilidad, y no con aspectos relacionados con la tecnología del videojuego.

3.2. El motor del juego

El módulo del software encargado de la gestión de las diferentes partes del juego y las relaciones entre ellas se denomina *motor del juego*. Pero no es hasta mediados de los 90 cuando se acuña el término.

El término surge a mediados de los años 90 en referencia a los juegos en primera persona tan populares en aquella época como `Doom` (id Software, 1993). La mayoría de estos videojuegos poseen la característica de estar desarrollados con una arquitectura en la que se separa de manera clara la gestión de los distintos sistemas que componen el juego (como el sistema de rendering 3D, el sistema de colisiones o el sistema de audio) de los recursos de arte del juego (mapas, posiciones de los objetos, niveles, texturas, etc.) Antes de esto, la implementación de los videojuegos no separaba la parte del motor del resto de componentes, haciendo muy difícil algún cambio en el juego, ya que este implicaba modificar parte de la programación de este. A partir de la publicación de Doom, los juegos empiezan a diseñarse separando la parte relacionada con el arte (mapas, mundo, niveles, etc.) a ficheros externos al motor, permitiendo realizar modificaciones en la jugabilidad de manera sencilla, e incluso publicando distintos videojuegos aprovechando el motor ya programado. Aunque todos los motores comparten entre ellos muchas características, cada género especializa la arquitectura del motor en formas distintas adaptándolo a sus necesidades. Así, el motor de un juego de plataformas estará especializado de manera distinta que el motor de un juego de lucha.

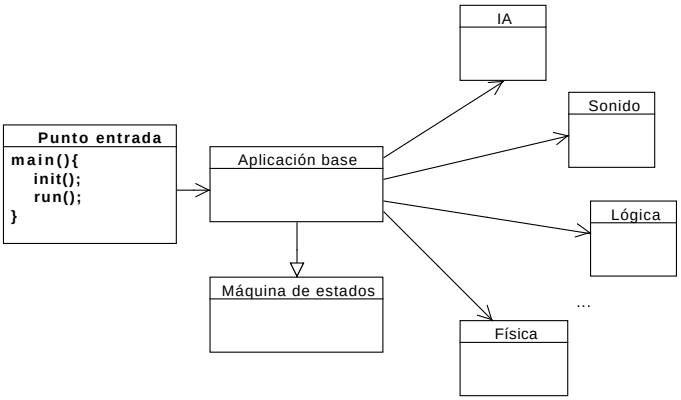


Figura 3.2: Diagrama con la arquitectura general de un videojuego

La arquitectura general de un juego está formada por una módulo principal encargado de inicializar los distintos servicios del juego (gráficos, IA, física, sonido, ...), y una máquina de estados que indica en que fase del juego se encuentra el jugador (menú de pausa, carga de nivel, jugando, ...). En la figura 3.2 se muestra un esquema con esta arquitectura. El tiempo del juego se encuentra gobernado por un único reloj. En cada tick de este, se avisa a todos los servicios para que se actualicen, y a la máquina de estados para que modifique el estado actual si fuese necesario. La sección siguiente explica el funcionamiento de la máquina de estados para la ejecución del juego.

3.3. La máquina de estados

La ejecución de un videojuego se basa en una máquina de estados que en cada tick de reloj actualiza su estado actual. Cada estado tiene una funcionalidad específica, que dependerá de la implementación del juego. En la tabla 3.1 se muestra un ejemplo de los posibles estados que puede tener una aplicación.

Cada estado se encarga de gestionar la aplicación de una manera diferente, por ejemplo, un estado que represente la carga de un nivel desactivará la lectura de pulsaciones del teclado por parte del usuario, mientras que un estado de juego enviará las pulsaciones al componente encargado de mover al avatar. Durante la ejecución de la aplicación se realizan transiciones entre los distintos estados, haciendo que el estado activo cambie y así el comportamiento de la aplicación.

Así mismo, la aplicación base es la encargada de escuchar los eventos del teclado y del ratón, delegándolos al estado activo de la máquina, el cuál es el

Estado	Descripción del estado
Menú Inicio	Menú inicial del juego. Es el primero en cargar. Muestra distintas opciones al usuario (nueva partida, cargar partida guardada, ...).
Carga Nivel	Un nivel está siendo cargado. Puede avisar al responsable de pintar una pantalla de progreso de carga.
Menú pausa	El jugador ha pulsado pausa en medio de una partida.
Juego	Estado principal de la máquina. Activo cuando el jugador se encuentra jugando a un nivel.
Game over	Fin de la partida.

Tabla 3.1: Ejemplo de los estados que puede gestionar la máquina de estados

encargado de notificar a los distintos servicios que se encuentran esperando a estos eventos.

3.4. El componente lógico del juego

Cada uno de los elementos que componen el juego y que el motor tiene que gestionar se denomina *entidad*. Una entidad es la pieza de software capaz de interactuar con el jugador humano o con otras entidades por ella misma. Así, una entidad se caracteriza por un conjunto de comportamientos representados por una serie de métodos y atributos. La parte del motor encargado de gestionar estas entidades es el *sistema de lógica*.

Desde los años 90 que se creó el primer motor hasta la actualidad, la arquitectura que se ha utilizado para gestionar las distintas entidades que componen el juego ha ido variando. Dentro del paradigma de la programación orientada a objetos, se describe a continuación un primer diseño de una arquitectura basada en herencia y los distintos problemas que esta presenta. Posteriormente se presenta una alternativa a esta arquitectura, basada en mensajes, para la cual ha sido desarrollado el sistema de testing.

3.4.1. Arquitectura basada en herencia

Siguiendo un paradigma orientado a objetos, la forma más natural de implementar una entidad formada por múltiples comportamientos es encapsulando cada comportamiento en una o varias clases. Así, una entidad será una clase base a la cual se le van agregando distintas funcionalidades a través del mecanismo de herencia. En la figura 3.3, extraída de Gómez Martín (2007), se muestra una jerarquía parcial de las entidades que forman el videojuego Half-Life 1 (Valve Software, 1998). Se puede observar como todas las entidades heredan de la misma clase base `CBaseEntity`, y según van heredando de diferentes clases se les dota de distintos comportamientos. Así, un enemigo,

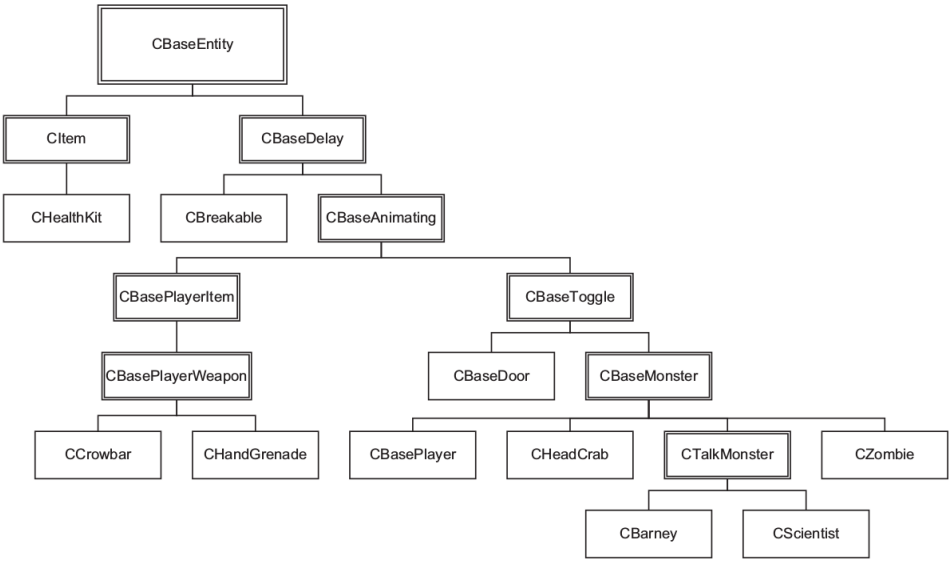


Figura 3.3: Jerarquía parcial de los tipos de entidad de Half-Life 1

por ejemplo un *zombie* representado mediante la clase `CZombie`, hereda de la clase que representa a los monstruos `CBaseMonster`, el cual es a su vez una entidad animada, por lo que hereda de la clase `CBaseAnimating` que le otorga este comportamiento. Aparte, existen monstruos con los que se puede hablar e interactuar, por ejemplo `CScientist` o `CBarney`. Para otorgarles esta habilidad especial, se considera un nivel más de herencia, incluyendo la clase `CTalkMonster` en la cadena de clases heredadas.

Este método, aparte de necesitar invertir una gran cantidad de tiempo en diseñar una estructura de clases y jerarquía, presenta ciertos problemas adicionales descritos en Gregory (2009):

- *Difícil de entender.* Cuanto más amplia sea la jerarquía, más difícil resulta de entender el comportamiento de una clase, ya que se necesita entender el comportamiento de todas sus clases padre.
- *Difícil de mantener.* Un cambio en un método de una clase puede estropear el comportamiento de las clases derivadas, ya que el cambio puede modificar suposiciones que habían sido hechas por las clases hijas.
- *Difícil de modificar.* Añadir una nueva funcionalidad o realizar algún cambio supone entender perfectamente toda la jerarquía para no introducir ningún error en el desarrollo.
- *Herencia en diamante.* Supongamos que se desea incorporar a Half-life 1 un nuevo personaje que posea las características de un *zombie*, pero

además pueda hablar con el jugador. Según la jerarquía mostrada en la figura 3.3, la forma de implementarlo sería a través de una clase que herede de `CTalkMonster` y de `CZombie` simultáneamente, produciéndose la herencia en diamante.

- *Dificultad para definir la jerarquía de clases.* Cada nivel de herencia en la jerarquía de clases supone decidir por que característica realizar la distinción de entidades. Así, podría dividirse por comportamiento, o decidir separar entre entidades que tengan sonido y las que no, o las que necesiten un sistema de partículas de las que no. Esto hace que existan múltiples taxonomías entre las que elegir, y según las distinciones que se realicen, puedan surgir unos problemas u otros.

- *Efecto burbuja.* Añadir funcionalidades nuevas a las entidades supone muchas veces mover métodos hacia arriba en la jerarquía de clases, produciendo clases base muy grandes y sobrecargadas. En el ejemplo de Half-life 1, supongamos que al pintar un *kit de vida* (`CHealthKit`) se desea que aparezca un sistema de partículas. Y que de igual forma al tirar una *granada de mano* (`CHandGranade`) deba aparecer un sistema de partículas también. Para evitar tener duplicado el código del sistema de partículas en cada clase, es necesario mover este sistema a la clase `CBaseItem`. A causa de esto, todas las entidades serían capaces de dibujar un sistema de partículas aunque no necesiten usarlo. Esto produce que juegos como Half-life 1 posea 87 métodos en su clase base con 20 atributos públicos, o que la clase base del juego *los Sims* posea más de 100 métodos.

- *Mayor tiempo de compilación.* El uso de herencia en C++ es uno de los mecanismos que provoca mayor retraso en tiempo de compilación. El compilador tiene que procesar todos los ficheros de cabecera de las clases padre, y el enlazador acceder a todos los ficheros objetos para resolver las dependencias. Como se muestra en Lakos (1996) esto provoca un aumento significativo en el tiempo de generación del ejecutable final.

3.4.2. Arquitectura basada en componentes

La forma mas habitual para solucionar estos problemas consiste en sustituir la herencia de clases por relaciones de composición o agregación, de esta manera una entidad estará compuesta por un conjunto de objetos relacionados entre si a través de un objeto principal que contiene al resto. A las objetos que definen comportamientos y forman una entidad se las denomina *componentes*. Esta forma de gestión de entidades a partir de un conjunto de componentes se denomina *Arquitectura basada en componentes*.

```
<BluePrints>
  ...

  <entity type="HealthKit">
    <component type="CStaticGrpahicComponent"/>
    <component type="CParticleSystemComponent"/>
    <component type="CTriggerComponent"/>
    <component type="CHealthKitComponent"/>
  </entity>

  ...
</BluePrints>
```

Figura 3.4: Fichero blueprints con la definición de dos entidades de Half-life 1

Como cada componente define una única característica, estos pueden organizarse por características a través del mecanismo de la herencia. Así puede existir una clase encargada de comunicarse con el motor gráfico del juego, y de esta heredar distintos componentes encargados de dibujar objetos estáticos, o la geometría de un modelo con huesos, o el sistema de partículas. Al encontrarse agrupados por una característica en común y no tener que diferenciar entre múltiples características sin relación, los problemas vistos en la sección anterior a causa de la herencia no se presentan.

De esta forma, el diseñador del juego ya no tiene que preguntarse por qué características diferenciar unas entidades de otras, sino qué características posee cada entidad. Por ejemplo, un *kit de vida* debería ser capaz de pintarse con un modelo estático, poseer un sistema de partículas asociado, y avisar cuando sea tocado para que se recoja. De igual manera, necesita saber cuánta cantidad de vida va a recuperar cuando sea usado. Tras este análisis, el programador debe crear un componente por cada uno de estas características y formar la entidad como la unión de todos estos componentes.

Gracias a este nuevo mecanismo, la descripción de una entidad se puede extraer a un fichero externo. Estos ficheros externos, normalmente denominados `BluePrints`, definen cada entidad como el conjunto de componentes que la forman. En la figura 3.4 se muestra un ejemplo de un fichero blueprints con formato xml, correspondiente a la definición de la entidad `HealthKit` a partir del análisis realizado en el párrafo anterior.

Así, cada entidad del juego podrá ser definida en un fichero externo como una lista de componentes que la forman. De esta forma, crear una entidad es generar una factoría que procese este fichero y cree una clase base a la que le añada distintos componentes. En la inicialización de cada componente, se considera una referencia a la clase base que define la entidad para poder acceder a atributos comunes a la misma. En este diseño, una entidad no es más que una colección de componentes.

Sin embargo, en este nuevo método, al no tener una jerarquía bien definida, la comunicación entre componentes no puede realizarse a través de llamadas a métodos heredados, por lo que es necesario un nuevo mecanismo de comunicación entre componentes. En la sección siguiente se detalla un mecanismo basado en paso de mensajes entre entidades.

3.4.3. Comunicación entre componentes: paso de mensajes

Un videojuego es una aplicación en tiempo real, la cual reacciona frente a los eventos que recibe. Un *evento*, según (Gregory, 2009, p. 773), es cualquier cosa de interés que ocurre mientras el jugador está jugando al videojuego. Una explosión en el juego, cuando un enemigo golpea al jugador o cuando un objeto es recogido son ejemplos de distintos eventos que pueden ocurrir en el juego.

Frente a cada evento producido, las distintas entidades tienen que reaccionar actualizando sus estados. La forma de actualizarse es a través de los componentes que la forman, que según el evento, producirán un comportamiento u otro en la entidad. Así, dependiendo del evento, habrá componentes que reaccionen ante él, y otros que no realicen ningún comportamiento.

En una arquitectura basada en herencia, la notificación de los distintos eventos a las entidades no supone ningún problema. Al estar correctamente tipadas, se pueden crear métodos específicos para cada evento soportado, y dependiendo del tipo de evento, llamar a un método u otro. Sin embargo, una arquitectura basada en componentes no posee una taxonomía de clases tan jerarquizada, por lo que para avisar a las entidades no puede realizarse a través de llamadas a métodos específicos para cada evento. La forma de notificar a las distintas entidades es a través de mensajes. Cada evento se encapsula en un mensaje genérico, el cual es enviado a las entidades. Sin embargo, cada evento tiene asociados unos atributos distintos, por lo que un mismo tipo de mensaje para todos los eventos no es posible. La solución consiste en crear una jerarquía de mensajes para encapsular distintos tipos de eventos. Cada mensaje almacenará los atributos propios del evento.

Se presentan dos formas de enviar estos mensajes a las entidades. La primera es que las entidades se registren como listeners de los mensajes que le interesa. El problema es que esto proporciona una implementación complicada y liosa. La segunda forma (y utilizada en la implementación del sistema de testing), consiste en que todas las entidades reciban todos los mensajes, y decidan ante qué mensajes van a modificar su estado, y ante cuales no. Para ello, todos los mensajes deben heredar de una clase base que define el tipo, y además, tendrán un atributo indicando el tipo de evento que contienen. De esta manera, será más fácil para un componente detectar si tiene que reaccionar ante ese mensaje, y en caso de ser así, realizar el casting al tipo de mensaje correspondiente que encapsula ese evento.

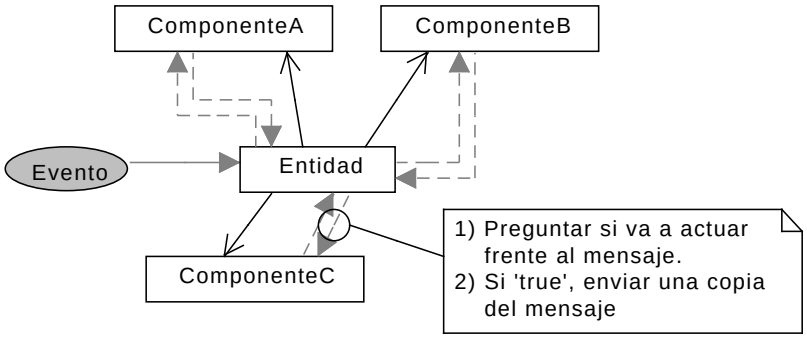


Figura 3.5: Envío de mensajes entre los componentes de una entidad

Sin embargo, el enviar una copia de todos los mensajes a todos los componentes del juego es una tarea muy costosa tanto en tiempo como en recursos. Para evitar esto, se divide el proceso en dos subprocesos. Antes de enviar una copia del mensaje al componente, se le pregunta a este si va a reaccionar a ese mensaje o no. Y en caso de que responda afirmativamente, se envía la copia del mensaje al componente para que modifique el estado de la entidad a partir del evento producido. En la figura 3.5 se muestra un esquema con este proceso. Debido a la carga de trabajo que puede suponer procesar un evento, y al número de mensajes que se puede llegar a recibir, una estrategia comúnmente utilizada consiste en guardar todos los mensajes en una cola, y no reaccionar a ellos hasta el siguiente ciclo de reloj.

En ocasiones, un componente puede querer comunicarse con otro componente. Para ello este envía un mensaje a la entidad que contiene el componente. La entidad envía el mensaje a todos los componentes que posee, repitiendo el proceso mencionado anteriormente de preguntar si acepta el mensaje antes de enviarle una copia. Si la entidad destino es la misma que la entidad de origen, esta debe tener cuidado con no enviar el mensaje al componente que lo ha originado.

Conclusiones

En este capítulo se ha mostrado la gran cantidad de elementos con los que trata un videojuego. Se ha diferenciado una parte relacionada con la tecnología y otra parte relacionada con la jugabilidad o experiencia del juego. Aplicar testing sobre la parte relacionada con la tecnología está cubierto por las herramientas de testing tradicionales, pero al intentar aplicarlo sobre la parte relacionada con la jugabilidad surgen problemas como se mostró en el capítulo anterior. En este trabajo se propone un sistema de testing novedoso

para intentar eliminar las carencias que poseen las herramientas de testing tradicionales.

Gran parte del capítulo de ha dedicado a detallar la arquitectura general con la que se diseña un videojuego y la forma utilizada para tratar los distintos eventos que suceden. En el próximo capítulo se muestra el sistema de testing propuesto, y como se aprovecha de esta arquitectura para intentar conseguir los objetivos marcados.

Capítulo 4

Diseño de un sistema de testing para videojuegos

The cleaner and nicer the program, the faster it's going to run. And if it doesn't, it'll be easy to make it fast.

Joshua Bloch

Resumen: En este capítulo se muestra la arquitectura general del sistema de testing. Se detalla qué datos se recogen y cómo se realiza. Así mismo, se explica la estructura modular del sistema. El capítulo finaliza tratando los ficheros de configuración de los test, y los campos que estos deberían tener.

4.1. Introducción

Como se ha mencionado en la introducción y en el capítulo 2, comprobar que un videojuego se encuentra libre de errores no es una tarea fácil. Las herramientas de testing desarrolladas hasta el momento no son capaces de comprobar la aplicación en su totalidad, limitándose a aspectos de la implementación o de la tecnología del videojuego. Un cambio en la jugabilidad del videojuego (como puede ser el cambio de posición de una puerta en un mapa explicado en la introducción) puede producir que el nivel no pueda ser superado, sin embargo, este tipo de fallos no los contempla el testing tradicional.

En este trabajo se está describiendo un nuevo sistema de testing basado en imitar los movimientos de un jugador experto que previamente han sido grabados. Estos movimientos son modificados para adaptarlos al nuevo mapa

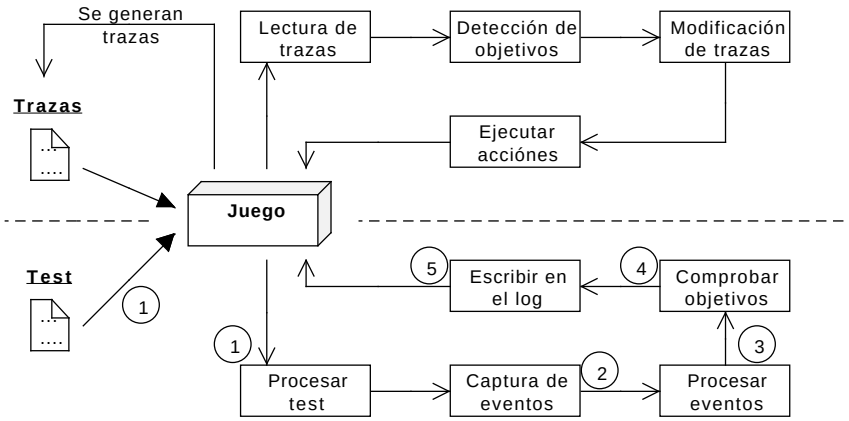


Figura 4.1: Estructura general de la herramienta de testing

mientras son reproducidos de manera automática en el juego. Mientras son reproducidos, se capturan todos los eventos que ocurren en el juego, y con ellos se comprueba si se satisfacen las condiciones de los test creados por el programador.

En la figura 4.1 se puede ver un esquema general del nuevo método de testing. Como se puede observar, se encuentra dividido en dos zonas. La parte superior, que corresponde a la generación de trazas y a la modificación de estas para adaptarlas a los nuevos mapas con su reproducción posterior, se describe en Hernández Bécares (2015). En este documento se describe la parte inferior del diagrama, que se corresponde a la ejecución de los test, la recogida de los eventos que se producen y su análisis para comprobar si se satisfacen los objetivos del test, similar a la estructura descrita en la sección 2.2.2.1 sobre los test de unidad. Esta parte se compone de:

1. *Diseño de test.* Es necesario definir la forma en la que se van a detallar los test, los objetivos que estos persiguen y características adicionales.

2. *Captura eventos durante el juego.* Determinar de que manera se van a capturar los distintos eventos que se producen durante la ejecución del videojuego. Se describe como realizar esta tarea sin que el programador tenga que realizar ningún esfuerzo extra aprovechando la arquitectura descrita en el capítulo anterior.

3. *Procesar los eventos capturados.* Durante el juego se produce una gran cantidad de información. Es necesario procesar y filtrar toda esa información para poder comprobar los objetivos del test.

4. *Comprobar objetivos.* Por cada evento capturado, el sistema detecta si se ha cumplido un objetivo marcado por el test o una parte de este.

5. *Escribir en el log.* Todo el proceso de testing va acompañado de un log que documenta lo que ha ocurrido, así como el resultado de los test.

En las siguientes secciones se describen los distintos elementos que forman el sistema, prestando especial atención a los módulos encargados del diseño de los test, reproducción automática de test, detección de objetivos, y creación del *log* con los diferentes datos. Aunque existen múltiples arquitecturas para el desarrollo de un videojuego, el sistema desarrollado y descrito en esta memoria está pensado para arquitecturas basadas en componentes, utilizando paso de mensajes para la comunicación entre estos, detalladas en la sección 3.4.2.

El sistema propuesto, a parte de intentar solucionar los problemas presentados en 2.4, ha sido diseñado intentando verificar las siguientes condiciones:

- *Test sencillos de crear.* Como la creación de los test es una tarea costosa se ha intentado reducir la complejidad de este proceso. Para ello se han propuesto ficheros de configuración con campos sencillos de rellenar. Además, en la implementación sobre el videojuego real, se ha utilizado Json como lenguaje de configuración, ya que simplifica la notación.

- *Comprobar múltiples datos.* El sistema diseñado permite comprobar múltiples condiciones para superar o no los test. Así, se detectan acciones que ocurren en el juego, el tiempo que lleva ejecutándose el test, y si el usuario supera el nivel.

- *Condiciones de fracaso.* Muchas veces se desea crear test para comprobar que no ocurren ciertos comportamientos inesperados. El sistema propuesto permite indicar condiciones de fracaso que, en caso de cumplirse, determinaría que el test es inválido. (Aunque si se hubiera seguido ejecutando podría haberse superado el nivel).

- *Condiciones complejas.* El sistema permite diseñar condiciones complejas basadas en la concatenación de varias condiciones más simples a través de operadores `and` y `or`.

4.2. Ejecución de los test

El sistema de testing ha sido desarrollado con la idea de que una vez lanzado, no requiera de intervención humana en todo el proceso. Para conseguir esto, es necesario modificar la forma en la que se lanza el videojuego, pero sin obligar al programador a adaptarse al nuevo sistema.

Para conseguir esto, el sistema se ha diseñado de la siguiente manera:

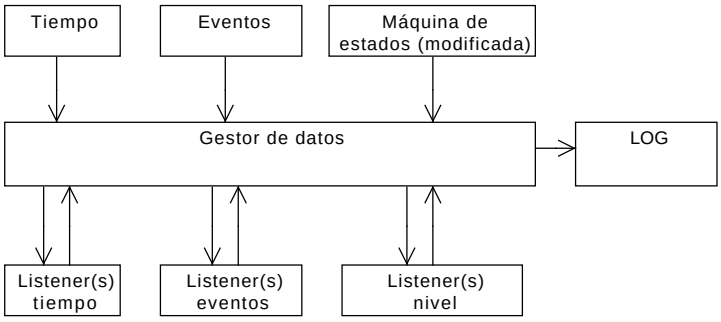


Figura 4.2: Esquema general con la arquitectura del gestor de datos

- *Ejecución de varios test de manera consecutiva.* Para poder lanzar varios test de manera consecutiva, es necesario reiniciar todo el videojuego por cada test. Para ello, por cada test a ejecutar se crea el videojuego de cero (invocando a todos los métodos `new` necesarios en C++), y se destruye al acabar de ejecutar el test (invocando a todos los métodos `delete` de C++).

- *Integración con el resto del código.* El sistema se encuentra implementado igual que cualquier otro sistema del videojuego. Así, crearlo y destruirlo no supone ningún esfuerzo para el programador, y se puede inicializar a la vez que el resto de sistemas.

- *Inicio del test.* Aunque el sistema de testing se inicialice junto con el resto de sistemas, antes de activar el reloj del juego se avisa al gestor de datos para que comience a recoger todos los datos y procesarlos para comprobar si se consiguen los objetivos del test.

4.3. Recolección de datos

El gestor de datos es la pieza que, una vez iniciada la reproducción de las trazas, se encarga de recoger toda la información producida durante el juego, tratarla y producir un resultado de acuerdo al test ejecutado en el momento. El objetivo es diseñar un sistema modular que reciba todos los datos y delegue el tratamiento de estos a módulos especializados según el tipo de dato. Este sistema se debe comunicar con los distintos módulos para transmitirles la información, y a su vez recibir los resultados de procesarlo. Con la información recibida de procesar los datos, este escribe en el log del test. La figura 4.2 muestra un esquema de la organización propuesta.

Una vez producido un dato, este es enviado al gestor a través de invocaciones a métodos conocidos de este. La forma con la que se obtienen los datos

se explica en las secciones siguientes. En el sistema desarrollado se considera que la información del juego interesante puede ser de tres tipos:

- *Tiempo.* El tiempo transcurrido desde que comenzó el test. Un tiempo excesivo puede suponer que el test nuca va a superarse, o que es demasiado complejo como para alcanzar los objetivos.
- *Eventos.* Todos los mensajes producidos durante la reproducción de trazas sirven para detectar objetivos parciales o finales. Es la fuente de información principal del sistema.
- *Máquina de estados.* Debe informar de cuando se comienza un nivel y se finaliza. También debe detectar casos de *game over* si los hubiera.

Para gestionar los módulos encargados de procesar los datos existen múltiples alternativas, cada una con sus ventajas y sus desventajas. La solución aquí propuesta es hacer que cada módulo se registre como listener en el gestor. El gestor de datos posee diferentes tipos de listener, para que según que tipo de dato obtenga, comunicárselo solamente a los módulos que quieren tratar ese dato. El resultado de esos datos se comunica al gestor a través de métodos conocidos de este.

4.3.1. Modificando la máquina de estados

Como se describió en la sección 3.3, la máquina de estados es una de las piezas más importantes en la ejecución de un videojuego ya que indica en que estado se encuentra este, y las acciones que están permitidas realizar. Sin embargo, a la hora de realizar un test, muchas de las funcionalidades de la máquina no son necesarias, llegando incluso a incomodar. Para ello, a la hora de realizar los test se debe sustituir la máquina de estados original por una nueva implementada para este propósito. La nueva máquina de estados posee las mismas funcionalidades que la máquina de estados original, pero con unas ligeras modificaciones:

- *Eliminar estados.* Muchos estados de la máquina son innecesarios a la hora de realizar los test. La nueva máquina de estados elimina estos estados que no afectan a la ejecución de los test (por ejemplo el menú de pausa o el menú de inicio descritos en la tabla 3.1).
- *Aviso de carga y finalización de nivel.* Cuando la máquina de estados cargue o finalice un nivel, esta debe avisar al gestor de datos para que trate la información.
- *Aviso de* game over. Igual que cuando se carga un nivel, se debe avisar al gestor si el juego ha finalizado, para notificar a los módulos encargados de manejar esta información.

- *Carga de niveles a demanda.* Para facilitar la tarea de reproducir un test, la máquina de estados debería ser capaz de cargar el nivel solicitado en cualquier momento. Como se detalló en el capítulo 3, la información asociada a los niveles y demás recursos de arte se encuentra en ficheros externos a la aplicación. Cargar un nuevo nivel supone reiniciar todos los servicios (sonido, IA, física, ...), y procesar el fichero de texto con el nivel, cargando todos los datos necesarios.

- *Finalización del juego a demanda.* Cuando un test no es superado, el videojuego tiene que ser finalizado para poder lanzarlo otra vez más. Para ello, la máquina de estados debe ser capaz de recibir la notificación de finalizar el juego y proporcionar mecanismos para parar o reiniciar todos los sistemas asociados.

4.3.2. Capturando eventos

La característica principal de este nuevo método de testing consiste en la reproducción de trazas grabadas previamente a un jugador experto y modificadas para adaptarlas a las nuevas condiciones del juego. Además, el sistema de testing es capaz de capturar los distintos eventos que ocurren en el juego en tiempo real, y tratarlos para comprobar si se verifican las condiciones del test actual o no.

Al estar desarrollado para una arquitectura basada en componentes que utiliza paso de mensajes para comunicar los distintos eventos, capturar todos los eventos producidos durante el juego es lo mismo que leer todos los mensajes que se intercambian en la aplicación. Con el objetivo de que el programador no tenga que diseñar el juego de manera específica para el sistema de testing, la forma de interceptar todos esos mensajes consiste en crear un nuevo componente que acepte todos los mensajes (ver figura 3.5). Este nuevo componente, en vez de modificar el estado de la entidad asociada como haría cualquier otro componente, reenvía todos los mensajes al gestor de datos. De esta manera, el gestor posee una copia de los mensajes de cada entidad, pudiendo detectar las condiciones del test. En la figura 4.3 se muestra un esquema del funcionamiento de este nuevo componente.

Este método presenta una ventaja adicional, y es que al tratarse un componente, no es necesario que se registre en todas las entidades del juego, sino solamente en aquellas que sean interesantes para superar los objetivos del test. Por ejemplo, para un test común no debería ser necesario leer los mensajes que se intercambian las entidades encargadas del sonido, por lo que no haría falta registrar el nuevo componente en estas entidades. La forma de registrar este componente sería a través del mecanismo de `BluePrints`, como puede verse en la figura 3.4.

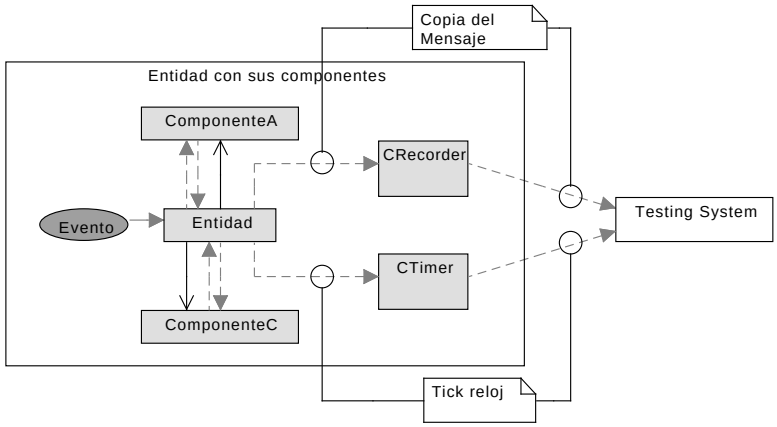


Figura 4.3: Obtención de los eventos y del click del reloj durante el juego

4.3.3. Registrando el tiempo

Un tiempo excesivo en la ejecución de un test puede suponer que los objetivos del test ya no son alcanzables. Esto puede ser a causa de que el nuevo mapa no puede ser superado, el sistema no ha adaptado bien las trazas, el nivel es demasiado complejo o que las trazas grabadas no son válidas para la ejecución actual. Por ello, el creador del test debe ser capaz de indicar un tiempo máximo en el que el test debe superarse (o fracasar, dependiendo del caso). Si el tiempo indicado es excedido, el sistema reportará un fracaso.

La gestión del tiempo en un videojuego es un tema muy complejo como puede verse en Rollings y Morris (2004). Aunque exista una única fuente de tiempo en la aplicación, las distintas partes del juego no siempre manejan el reloj de la misma manera, pudiendo no funcionar a la misma velocidad. Por ejemplo, el sistema de dibujado en pantalla puede funcionar a una frecuencia más alta que el sistema encargado de leer la entrada del jugador. Además, esta fuente de tiempo puede variar según el hardware sobre el que se esté ejecutando el videojuego (si se usan los ticks de CPU como reloj, la medida del reloj dependerá de la frecuencia del procesador).

Para la extracción de tiempo, suponemos que el servidor lógico tiene un reloj fijo independiente de la arquitectura. Así mismo, la implementación de la arquitectura avisa en cada tick de reloj a los distintos componentes para que actualicen sus estados. Gracias a esto, avisar del tiempo al gestor de datos consiste en crear un nuevo componente que en cada tick avise al gestor, similar que el componente encargado de transmitir los mensajes. Sin embargo, este componente sólo hay que registrarlo en una entidad, ya que si se registra en varias entidades, se avisaría al gestor más de una vez por ciclo. Este sistema también se muestra en el esquema de la figura 4.3.

El gestor de datos posee un reloj interno alimentado con los tick provenientes de la lógica. En cada tick avisa a los listeners registrados para escuchar notificaciones relacionadas con el tiempo.

4.3.4. Sistema de log

El sistema de log es el encargado de escribir en un fichero el log con los resultados que se producen durante la ejecución de los test. Este sistema no es encargado de decidir que mostrar por la salida, sino que es el gestor de datos el encargado de indicarle qué mensajes guardar en el log. Gracias a separar el sistema de log del gestor de datos, el formato del log puede variar de una implementación a otra, pudiendo ser texto plano, html o cualquier otro formato que se desee.

Así mismo, el sistema de log implementa una categoría con distintos tipos de mensaje, permitiendo al gestor de datos elegir qué tipo de mensaje es, y mostrarlos en el log de una u otra manera. Esta categoría sirve para distinguir mensajes de test superado, test fallidos o mensajes de información, entre otros.

4.4. Configuración de los test

Crear los test de una aplicación es una tarea difícil, y si encima la forma de implementarlos es también difícil, hacen que esta tarea se convierta en una tarea imposible. Al contrario que pasaba en los test de unidad descritos en 2.2.2.1, el sistema propuesto se basa en configurar los test a través de archivos de texto, y no a través de código.

Así, un conjunto de test estará definido por un conjunto de archivos, uno por cada test, y un archivo extra que sirve para reunir los diferentes test. En el fichero principal (o fichero de configuración de los test), la información requerida es la ruta y el nombre del fichero donde se va a generar el log, y los distintos ficheros que contienen los distintos test a ejecutar. Además, cada test va acompañado del número de repeticiones que se van a realizar en caso de que el test no se supere. Esto es debido a que muchas veces superar o no el nivel depende de factores externos al test (como puede ser el comportamiento de un enemigo), y aunque en una primera vez no se supere el test, en una segunda vez puede que si se consiga. En la tabla 4.1 se muestra un resumen con la información necesaria.

Cada test que forman el conjunto de test está definido en un archivo individual, permitiendo reutilizar test para varios conjuntos, o editarlos de manera más sencilla. En el fichero se tiene que indicar el sistema de reproducción que se desea utilizar (explicado en Hernández Bécares (2015), pueden reproducirse las trazas adaptadas, o las pulsaciones de teclas sin modificar del usuario experto), y los ficheros donde se han guardado las trazas al ser

Parámetro	**Descripción**
Fichero de log	Nombre y ruta donde se va a almacenar el log resultante.
Test:	— *Se repite tantas veces como test haya.* —
Fichero de test	Nombre del fichero que almacena el test
Núm. repetic.	Número de veces que se va a ejecutar el test en caso de fallo.

Tabla 4.1: Campos necesarios para crear un conjunto de test

Parámetro	**Descripción**
Tipo test	Indica el método que se va a utilizar para reproducir las trazas.
Nivel	Indica el nombre del nivel sobre el que se va a ejecutar el test.
Ficheros de trazas	Nombre de los ficheros donde se han guardado las trazas.
Tiempo máximo	Condición de fallo del test en caso de superar el tiempo indicado.
Cond. éxito	— *Colección de mensajes con clausulas and y or* —
Tipo	Indica si los mensajes tienen que producirse de manera ordenada o no.
Mensajes	Definición de los distintos mensajes esperados.
Cond. fallo	
Tipo	— *Igual que la condición de éxito.* —
Mensajes	

Tabla 4.2: Campos necesarios para crear un test específico

grabadas. Así mismo, es necesario indicar el nombre del nivel sobre el que se desea realizar el test.

Estos datos definen la configuración del test para ejecutarse. Aparte de esto, son necesarios las condiciones de éxito y fracaso del test. Se distinguen tres tipos de condiciones: tiempo máximo para ejecutar el test, condiciones de éxito y condiciones de fracaso. El tiempo máximo es indicado en formato número, e indica el tiempo máximo permitido para ejecutar el test medido con el reloj de la parte lógica del juego. Respecto a las condiciones de éxito o fracaso, estas son un conjunto de mensajes relacionados mediante clausulas `and` y `or`. A su vez, el usuario puede indicar si los mensajes tienen que producirse en orden, o pueden ocurrir en desorden. La tabla 4.2 muestra un resumen del formato.

Conclusiones

Tras ver los problemas del testing en el capítulo 2, y la arquitectura de un videojuego en el capítulo 3, en este capítulo se ha introducido de manera genérica el sistema de testing desarrollado con el objetivo de subsanar las carencias que el testing tradicional presenta.

En el próximo capítulo se explica como esta descripción general se ha aplicado a un videojuego real.

Capítulo 5

Un videojuego real: Time & Space

Programming: when the ideas turn into the real things.

Maciej Kaczmarek

Resumen: En el capítulo anterior se detalló en que consiste el nuevo sistema de testing propuesto, las partes que lo componen, y como integrar el sistema en un videojuego. En este capítulo se muestra la implementación sobre un videojuego real, producto del máster en desarrollo de videojuegos. También se muestra el formato elegido para escribir los test, y la forma en la que los resultados son mostrados.

5.1. Introducción

El sistema de testing descrito en la sección anterior ha sido implementado sobre un videojuego real llamado `Time & Space` (Blázquez et al., 2013-2014). Se trata de un videojuego desarrollado como proyecto final en el máster de desarrollo de videojuegos de la Universidad Complutense de Madrid para el curso académico 2013-2014. Se ha escogido este juego por dos razones principalmente: (i) los autores han dado permiso para acceder al código fuente y modificarlo, y (ii) dispone un reloj lógico implementado para que funcione a un tiempo fijo independiente de la arquitectura, cosa necesaria para el sistema de reproducción como se ha explicado en la sección 4.3.3.

`Time & Space` está implementado en C++ para el sistema operativo Windows. La estructura general se basa en Galeón, un esqueleto de juego usado como ejemplo en el máster para aplicar los conocimientos teóricos

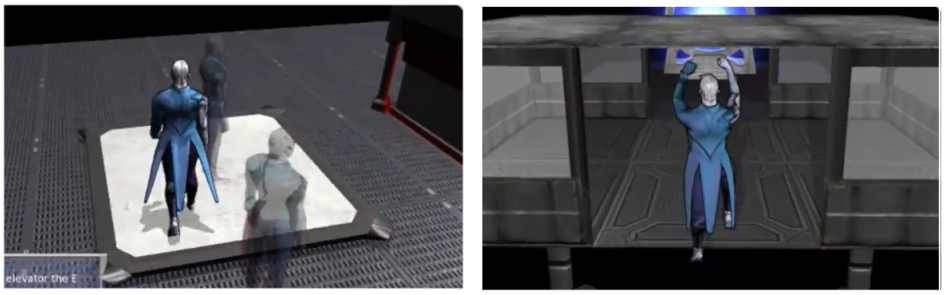


Figura 5.1: Capturas de pantalla de `Time & Space`

aprendidos. Este se encuentra implementado mediante una arquitectura basada en componentes con paso de mensajes, como se detalló en el capítulo 3.

5.1.1. Descripción de `Time & Space`

`Time & Space` es un videojuego organizado por niveles donde el objetivo de cada nivel es llegar a un portal que se encuentra en algún punto del mapa. El jugador únicamente controla al personaje principal, que es dirigido en primera persona. El personaje se encuentra formado por tres atributos: la vida, el arma principal que puede recoger del suelo, y un arma secundaria. Al no haber puntuación en el juego, el único objetivo es llegar a la salida del nivel, no existiendo otros objetivos como conseguir la máxima puntuación posible como existe en otros videojuegos.

Lo distintivo de este juego es la forma de superar los distintos niveles. Para ello, el jugador puede realizar en cualquier momento una copia de sí mismo. Cuando crea la copia, el nivel se reinicia, y la copia creada imita los movimientos del jugador en el turno anterior. Así, el jugador puede ayudarse de sus copias para pulsar botones o activar puertas mientras él realiza otras acciones al mismo tiempo. Otro uso que se le puede dar a las copias es servir de barrera entre los distintos enemigos y el personaje, sacrificando la copia para que el jugador pueda continuar su camino. Aunque el número de copias es ilimitado, el jugador puede disparar en cualquier momento a alguna de sus copias y esta desaparecerá del juego. Hay que destacar que en este juego no existe el concepto de *game over*, sino que una vez muerto el jugador, el nivel se reinicia y se introduce una nueva copia al juego que imitará los movimientos del turno anterior en el que el jugador fue asesinado. Si el jugador no está conforme con sus movimientos y no quiere crear una copia que lo imite, puede reiniciar el nivel sin crear una copia, pero manteniendo todas las copias anteriores.

5.2. Modificaciones introducidas

Para integrar el sistema de test en el videojuego, se han introducido modificaciones en el juego original. Como se puede comprobar a continuación, estas modificaciones no han afectado el código antiguo, sino que proporcionan formas alternativas de realizar distintas acciones que ya se realizaban antes. El nuevo código solo es utilizado cuando se desea realizar el testing, utilizando el código original para jugar al videojuego. Los cambios introducidos en el juego han supuesto un tamaño aproximado de $\approx 130KB$.

5.2.1. Inicialización del juego

`Time & Space` se encuentra formado por un gran número de clases y sistemas, y aunque cada sistema se encuentre implementado de manera oculta al resto, todos los sistemas tienen un servidor principal que sirve de punto de comunicación con el sistema. A su vez, es el propio servidor el que inicializa el sistema, por lo que basta con crear todos los servidores para tener inicializados los sistemas. Los servidores se encuentran implementados mediante un patrón *singleton* para poder ser accedidos desde cualquier parte del código.

La inicialización de `Time & Space` consiste en un módulo principal encargado de inicializar todos los sistemas que componen el juego, y una vez inicializados, crear la máquina de estados que gobierna a la aplicación. Con todos los sistemas inicializados, se llama al método `run()` que activa el reloj de la aplicación, y comienza a funcionar el videojuego.

Para integrar el sistema de testing, se han creado clases que imitan el comportamiento descrito anteriormente, pero con ciertas modificaciones. No se inicializan todos los sistemas, solamente aquellos necesarios para ejecutar el test (por ejemplo el sistema de sonido o el de gráficos no es necesario inicializarlo), así mismo, se inicializa el sistema de testing junto al resto de sistemas. La máquina de estados creada no corresponde con la máquina de estados original, sino que es una modificación de esta, explicada en la siguiente sección.

Para poder ejecutar los test, se ha duplicado el método `main()`, otorgando a la aplicación un nuevo punto de entrada. En este nuevo `main`, se procesa el número de test que se desean realizar, y por cada uno de ellos, se reinician todos los sistemas que forman el videojuego (incluido el sistema de test) y se construye una máquina de estados nueva que gobierne la nueva ejecución del juego. Así, se consigue que el estado sobre el que se ejecuta un test sea independiente de lo realizado por el test anterior. Al tener dos puntos de entrada a la aplicación, se consigue que el programador no tenga que modificar su implementación para incorporar la ejecución de los test, sino que basta con que cambie el punto de entrada a la aplicación.

Estado	Descripción del estado
Menu	Menú inicial del juego. Es el primero en cargar. Muestra distintas opciones al usuario (comenzar partida o salir del juego).
Loading	Carga un nivel. Muestra una pantalla de carga mientras se inicializan los distintos recursos asociados al nivel.
Pause	El jugador ha pulsado la tecla de pausa en medio de un nivel.
Game	Estado principal de la máquina. Activo cuando el jugador se encuentra jugando a un nivel. Comunica las teclas pulsadas al componente encargado de controlar al jugador.
Exit	El juego ha finalizado. Libera los recursos creados para el juego.

Tabla 5.1: Estados de la máquina de estados de `Time & Space`

5.2.2. La máquina de estados

La máquina de estados de `Time & Space` se encuentra formada por cinco estados, detallados en la tabla 5.1. Sin embargo, para realizar el testing, alguno de estos estados no son necesarios. La implementación de la nueva máquina de estados solo considera tres de los seis estados originales: *loading*, *game* y *exit*. Los otros estados, *menu* y *pause*, no son necesarios para el test ya que representan a distintos menús del juego que no van a ser comprobados durante la ejecución del test.

A los tres estados de la nueva máquina se les han realizado distintas modificaciones para integrarlos dentro del sistema de testing. Así, el estado *loading* avisa al gestor de datos cuando un nuevo nivel es cargado, y permite la carga de niveles a demanda. El estado *game* ejecuta el juego, y además avisa al gestor de datos cuando comienza el juego para empezar la medición del tiempo. El estado *exit* sirve para finalizar la máquina de estados y liberar todos los sistemas que se habían inicializado.

En el apéndice A.1 se muestran los cambios realizados a los nuevos estados.

5.2.3. Capturando los eventos y el tick del reloj

Como se explicó en la sección 4.3.2, la forma de capturar eventos consiste en crear un nuevo componente denominado `CRecorder` que acepte todos los mensajes y envíe una copia de estos al sistema de testing. La forma que tiene `Time & Space` de definir que componentes forman una entidad es a través del fichero de texto plano denominado `blueprints.txt`. Luego añadir este nuevo componente a las entidades solamente implica modificar ese fichero.

Para capturar el tick del reloj, se ha implementado la idea propuesta en 4.3.3. Se ha creado un nuevo componente denominado `CTimer` que avisa al gestor de datos en cada tick. Al contrario que el componente encargado de capturar los mensajes, este componente solamente se ha registrado en la

entidad que representa el avatar del usuario, y no en varias a la vez.

Los nuevos componentes, denominados `CRecorder.cpp` y `CTimer.cpp` respectivamente, se muestran en el apéndice A.2.

5.3. Gestor de datos

Este sistema es el encargado de recibir los diferentes datos del juego (tick del reloj, mensajes de eventos, y notificaciones de la máquina de estados), y notificarlo a los distintos módulos para que los gestionen, a través de un mecanismo de *listener*. Al inicializarse, procesa los ficheros de test detectando las condiciones de éxito y de fracaso, y se lo comunica a los módulos encargados de tratar con estos datos. En el apéndice A.3 se muestra el código que parsea el fichero de configuración de los test y inicializa los distintos módulos.

Las condiciones de éxito y fracaso vienen determinadas por una serie de mensajes que se espera que aparezcan durante la ejecución. Para detallar estos mensajes en modo texto, se ha creado una especificación en lenguaje Json, y creado múltiples clases para comparar estas descripciones con los mensajes que ocurren en el videojuego (en el apéndice A.4 se puede ver la implementación del método encargado de comparar mensajes de tipo `CTouchedMessage`). Estas clases de comparación son utilizadas para detectar si se cumplen los objetivos marcados o no en el test. Así mismo, ampliar la funcionalidad del sistema consiste en crear nuevas clases encargadas de comparar nuevos tipos de datos.

Como las condiciones de éxito y fracaso vienen determinadas por una lista de mensajes relacionados mediante `and` y `or`, la forma de implementar esto ha sido separar cada condición `or` en un módulo distinto, de tal manera que para superar el test es suficiente con que uno de estos módulos detecte todos los mensajes que contiene. Así mismo, para facilitar la tarea, se distinguen módulos con condiciones de éxito y de fracaso, informando de manera distinta al gestor según qué módulo se satisfaga.

Al diseñar un test, en la lista de mensajes que forman una condición de éxito o de fracaso puede especificarse si se desea que estos aparezcan en orden o no durante la ejecución de la aplicación. Internamente, esta distinción se encuentra reflejada en el contenedor de datos elegido para crear el módulo encargado de detectar las condiciones:

- *Mensajes ordenados.* En este caso, los mensajes se almacenan en una cola implementada en la librería estándar de C++ (`std::queue`). Cuando el módulo recibe un mensaje que concuerda con el mensaje que se encuentra en la cabecera, el mensaje es eliminado de la cola. Cuando la cola se vacía, se avisa al gestor de que se ha satisfecho una condición.

- *Mensajes desordenados.* Los mensajes son almacenados en un vector

```
enum LOG_TYPE{
  COND_OK,   // una condición ha sido satisfecha
  ERROR,     // se ha producido un error
  INFO,      // mensajes de otro tipo para el gestor
  TO_LOG     // mensajes para escribir en el log directamente
};
```

Figura 5.2: Tipos de mensaje en el gestor de datos

de C++ (`std::vector`) junto a una máscara de bits inicializada a 0. Cuando un mensaje es detectado, se activa el bit correspondiente a su posición. Cuando todos los bits se encuentran activos se avisa al gestor de contenido.

El mecanismo utilizado para notificar al gestor consiste en la invocación de un método, el cual recibe como parámetro el tipo de información que se está comunicando. En la figura 5.2 se muestran los distintos tipos de notificaciones que existen en `Time & Space`. Así mismo, este sistema de prioridades es utilizado para informar al sistema de log del tipo de mensaje que se desea escribir. Es el sistema de log el que decide en que formato se va a mostrar el mensaje. Este sistema se muestra en la siguiente sección.

5.4. Ficheros de test

El formato diseñado para que el programador escriba los test corresponde al formato mostrado en la sección 4.4. El lenguaje elegido ha sido Json por su facilidad y claridad a la hora de escribirlo, y la sencillez con la que puede ser leído desde el código.

Cada test se encuentra especificado en un fichero independiente con el objetivo de que sea más fácil trabajar con ellos, y que un mismo test pueda formar parte en varias colecciones de test. El fichero se encuentra formado por el tipo de test que se desea realizar (explicado en Hernández Bécares (2015)), el nivel sobre el que se desea realizar los test, los ficheros con las grabaciones de las trazas, el tiempo máximo permitido para superar el nivel, y las condiciones de éxito y fracaso. En la figura 5.3 se puede ver un ejemplo de un test. Como se observa, las condiciones de éxito o fracaso pueden venir dados por múltiples condiciones. Estas condiciones se encuentran especificadas en un array de Json, simulando relaciones `or`. Con que una de las condiciones se satisfaga, se considera que el test ha sido superado. A su vez, cada condición está formada por un conjunto de mensajes que se espera que ocurran. Se puede indicar si estos mensajes deben aparecer en orden o desorden durante la ejecución. El conjunto de mensajes se representa mediante otro array de Json, simulando las condiciones `and`.

```
{"type": "msg",
 "level_name": "Level2",
 "user_file": "../../level1_user_in.txt",
 "msg_file": "../../events.json",
 "max_time": 0,

 "success_conditions":[
     //cond exito 1
     {"type": "ordered",
      "msg" : [
        // el jugfador toca el boton 1
        { "type" : "TOUCHED",
          "associatedEntity": { "name" : "Player" },
          "entity" :          { "name" : "puertaTrigger1"}
        },
        //el jugador pulsa el boton 2
        { "type" : "TOUCHED",
          "associatedEntity": { "name" : "Player" },
          "entity" :          { "name" : "puertaTrigger2"}
        }
      ]
     },
     // Condicion 2
     {"type": "ordered",
      "msg" : [
        { "type" : "TOUCHED",
          "associatedEntity": { "name" : "Player" },
          "entity" :          { "name" : "ButtonTrigger1"}
        }
      ]
   }
 ], //condiciones de exito

  "failure_conditions":[
     //cond exito 1
     {"type": "ordered",
      "msg" : [
        { "type" : "TOUCHED",
          "associatedEntity": { "name" : "Player" },
          "entity" :          { "name" : "puertaTrigger1"}
        }
      ]
     }
  ]
}
```

Figura 5.3: Ejemplo de test para un nivel de `Time & Space`

```
{ "log_file": "log.txt",
  "path"     : "../../tests/",
  "tests"    :[
      {"filename" : "test1.json",
       "max_rep": 1},
      {"filename" : "test_user.json",
       "max_rep": 2},
      {"filename" : "test2.json",
      "max_rep": 2}
  ]
}
```

Figura 5.4: Fichero de configuración de los test

El archivo principal de configuración de los test, mostrado en la figura 5.4, contempla el nombre del fichero donde se va a escribir el log, así como una ruta relativa a este. A continuación se muestra en formato de array los distintos test que se van a ejecutar, junto con el número máximo de repeticiones. Como se mencionó anteriormente, este número de repeticiones es debido a que el juego tiene componentes no deterministas (como el comportamiento de un enemigo), y aunque en una primera ejecución no se consiga satisfacer el test, este puede superarse en ejecuciones posteriores.

5.5. Sistema de log

El sistema de log es encargado de escribir en un fichero la información que ocurre durante la ejecución de los test. La información se le suministra a través de un método que recibe el texto a mostrar en el log, y el tipo de mensaje. El tipo viene dado por la figura 5.2.

En la figura 5.5 se muestra un ejemplo de log al pasar dos test de manera consecutiva. En este caso, el sistema se ha implementado para escribir en texto plano, pero podría implementarse para escribir en cualquier otro formato como html, o incluso en un formato para integrarlo en otra aplicación que procese los resultados.

Conclusiones

En este capítulo se ha presentado una implementación concreta del sistema diseñado en el capítulo anterior. El videojuego elegido ha sido uno desarrollado en el máster de videojuegos a partir de un esqueleto proporcionado denominado Galeón. La implementación mostrada debería ser válida para todos los videojuegos desarrollados a partir de Galeón, o necesitar una mínima modificación.

```
****************************************
Test 1: NO!
   Verificada condición de fallo número: 1
   ESTADO DEL TEST:
      Cond. éxito:
         cond 1: 0%
         cond 2: 0%
      Cond. fracaso:
         cond 1: 100%

****************************************
Test 2: NO!
   Verificada condición de fallo número: 1
   ESTADO DEL TEST:
      Cond. éxito:
         cond 1: 33.3333%
         cond 2: 50%
      Cond. fracaso:
         cond 1: 100%

****************************************
Test 3: NO!
   El tiempo máximo establecido ha sido superado.
   ESTADO DEL TEST:
      Cond. éxito:
         cond 1: 75%
         cond 2: 33.3333%
      Cond. fracaso:
         cond 1: 0%
```

Figura 5.5: Ejemplo del log después de ejecutar tres test consecutivos fallidos

Si se desea implementar este sistema en otros videojuegos con arquitectura basada en componentes con paso de mensajes, las ideas mostradas en este capítulo y en el apéndice no deberían variar mucho, aunque si la forma de integrarlo dentro del videojuego.

Con este capítulo, se ha querido mostrar lo sencillo que es incorporar este nuevo método de testing al desarrollo de un videojuego, y como se puede implementar de manera que no interfiera con el desarrollo del videojuego, al contrario de como se puede pensar en un principio.

Capítulo 6

Conclusiones y trabajo futuro

Los ordenadores son inútiles,
solo pueden darte respuestas.

Pablo Picasso

Resumen: En este capítulo se hace un mínimo recorrido a través de todos los capítulos anteriores, recalcando los detalles más importantes de cada uno de ellos. Se muestra como más allá de ser una idea teórica, el nuevo método de testing tiene una utilidad real. El capítulo finaliza comentando posibles ampliaciones del trabajo o la creación de nuevas herramientas que faciliten la tarea de crear nuevos test.

6.1. Conclusiones

Como se ha mostrado en el capítulo 2, aplicar herramientas de testing al desarrollo de un videojuego es una tarea importante. Sin embargo, los métodos tradicionales aplicados hasta la fecha poseen ciertas carencias haciendo que sea necesario la intervención de jugadores humanos para comprobar ciertos aspectos del juego.

Para intentar suplir estas carencias, en este trabajo se ha propuesto una alternativa basada en imitar los comportamientos que ha realizado un humano con anterioridad durante una partida. Con ello, se ha conseguido diseñar un sistema capaz de detectar errores en los distintos niveles y mapas que forman el juego.

La evolución que ha habido en el diseño de la arquitectura de un videojuego ha permitido idear un sistema capaz de capturar todos los eventos que se producen en el juego, pudiendo procesarlos y trabajar con ellos simultáneamente que el videojuego está en ejecución. Además, la arquitectura

permite que el sistema se integre perfectamente en el videojuego sin necesidad de que el programador adapte el código del juego específicamente para la herramienta de testing.

Este nuevo método, más allá de ser una idea teórica, ha sido implementado con éxito en un videojuego real, comprobando que es totalmente factible. Al implementarlo, se ha puesto de manifiesto la utilidad que tiene en el desarrollo del videojuego, ya que mapas especialmente modificados para que no puedan superarse han sido detectados como errores en los test. Así mismo, la utilidad de esta herramienta ha quedado patente al ser aceptada para su presentación en el II Congreso organizado por SECiVi[1]. El artículo presentado puede leerse en Hernández Bécares et al. (2015), o en el apéndice B.

6.2. Trabajo futuro

Aunque el sistema diseñado ha sido implementado correctamente en un videojuego real, aún existen muchos aspectos en los que se puede ampliar y mejorar. Entre todos estos aspectos destaca la necesidad de implementar esta técnica en otros videojuegos más complejos y de probarlo con mapas donde los cambios sean más rebuscados.

Así mismo, para diseñar los test se ha escogido un formato Json, pero podría crearse una herramienta que facilitara el proceso de creación de los test en vez de tener que escribir los ficheros a mano. De igual manera, se podría crear una herramienta que mostrase de forma más visual los resultados de los test, o incluso integrarlos en el software utilizado para programar el videojuego (como puede ser jUnit que se encuentra integrado dentro del IDE eclipse, ver figura 2.2).

Depende del juego, los mensajes intercambiados tendrán distintas estructuras, lo que supone que a la hora de escribir los mensajes que forman las condiciones de un test, el formato variará de un juego a otro. Para evitar este inconveniente, se puede crear una jerarquía de mensajes válida para todos los videojuegos, o en su defecto, una herramienta que permita al creador del test escribir los mensajes de manera sencilla.

Por último, este sistema de testing no solo sirve para ser lanzado una única vez, sino que podría incorporarse dentro de sistemas de integración continua, o lanzamiento múltiple en varios servidores para escenarios grandes con condiciones complejas.

[1]SECiVi: Sociedad Española para las Ciencias del Videojuego

Apéndices

Apéndice A

Modificaciones de Time & Space

A.1. Máquina de estados

Con el objetivo de tener una máquina de estados más sencilla, y dotarla de funcionalidad extra para implementar el sistema de testing, se han modificado los diferentes estados, comunicándolos con el sistema de testing. A continuación se muestran los métodos `activate()` modificados, que son llamados la primera vez que el estado se convierte en activo.

LoadingTestState.cpp:

```
1  void CLoadingTestState::activate()
2  {
3    // Init base state
4    CApplicationState::activate();
5
6    //Notify test system that level is loading
7    if (Tests::CTestManager::getSingletonPtr() != NULL){
8      Tests::CTestManager::getSingletonPtr()->levelLoaded();
9    }
10
11 } // activate(void)
```

GameTestState.cpp:

```
1  void CGameTestState::activate()
2  {
3    CApplicationState::activate();
4
5    GUI::CHudManager::getSingletonPtr()->setState(this);
6
7    //Activate the loaded map
8    _logicServer->activateMap();
9
```

```
10    //Reset player and copies
11    Logic::CPlayerManager::getSingletonPtr()->clearAll();
12
13    //GUI is going to manage the player
14    GUI::CServer::getSingletonPtr()->getPlayerController()
15                                                ->activate();
16
17
18    //Notify testing system that game starts
19    if (Tests::CTestManager::getSingletonPtr() != NULL){
20      Tests::CTestManager::getSingletonPtr()->gameStarts();
21    }
22
23  } // activate(void)
```

A.2. Nuevos componentes

Aprovechando la arquitectura basada en componentes que implementa Time & Space, y además la comunicación por paso de mensajes, se han creado dos componentes nuevos encargados de interceptar todos los mensajes de la aplicación para transmitirlos al sistema de testing, y para leer el reloj del servidor lógico del juego. La implementación de estos dos nuevos componentes, CRecorder y CTimer, se muestra a continuación:

CRecorder.cpp:

```
1  bool CRecorder::accept(CMessage *message)
2  {
3    if (!isActivated) return false;
4
5    //Aceptamos todos los mesajes. TODOS
6    return true;
7  }
8
9  void CRecorder::process(CMessage *message)
10 {
11   //Enviamos una copia de los mensajes al trace
12   //server para ser grabados
13   if (Trace::CServer::getSingletonPtr() != NULL)
14     Trace::CServer::getSingletonPtr()->manageMessage(
15                                 message, _entity, _clock);
16
17   //Enviamos una copia de los mensajes al Test Manager.
18   if (Tests::CTestManager::getSingletonPtr() != NULL)
19     Tests::CTestManager::getSingletonPtr()
20           ->manageMessage(message, _entity, _clock);
21 }
```

CTimer.cpp:

```
bool CTimer::accept(CMessage *message)
{
  //Este componente solo avisa al Trace::CServer del tick
  //No debe manejar ningún mensaje
  return false;
}

void CTimer::tick(unsigned int msecs)
{
  if (isActivated)
  {
    IComponent::tick(msecs);

    _clock += msecs;
    if (Trace::CServer::getSingletonPtr() != NULL)
      Trace::CServer::getSingletonPtr()->tick(msecs);

    if (Tests::CTestManager::getSingletonPtr() != NULL)
      Tests::CTestManager::getSingletonPtr()->tick(msecs);

  }
}
```

A.3. Gestor de datos

A continuación se muestra como se parsea el fichero con las condiciones del test y se inicializan los módulos encargados de comprobar las condiciones.

```
bool CTestManager::start(Json::Value & _root){
  std::string user_file, msg_file, test_type;

  ++_testNum;

  /* Prepare the system for the test*/
  _time2 = 0;
  _levelLoaded = false;
  _gameStarted = false;
  _success = false;
  _active = true;
  _maxtime = _root.get("max_time", 0).asInt();


  test_type = _root.get("type", "").asString();

  if (test_type == "msg"){
    user_file = _root.get("user_file", "").asString();
```

```
19      msg_file = _root.get("msg_file", "").asString();
20
21      /* Start to replay the traces */
22      Trace::CServer::Init(msg_file, user_file);
23    }
24    else if (test_type == "user"){
25      user_file = _root.get("user_file", "").asString();
26
27      Trace::CServer::InitUser(user_file);
28    }
29    else{
30      writeLog(LOG_TYPE::INFO,
31               "Archivo de confuguración incorrecto."
32               "Campo \"type\" desconocido");
33      return false;
34    }
35
36
37    //Comprobamos condiciones de éxito
38    Json::Value cond = _root.get("success_conditions",
39                                 NULL);
40    if (cond != NULL){
41      for (int i = 0; i < cond.size(); i++){
42        loadSuccConditions(cond[i]);
43      }
44    }
45
46    //Failure conditions
47    cond = _root.get("failure_conditions", NULL);
48    if (cond != NULL) {
49      for (int i = 0; i < cond.size(); i++){
50        loadFailureConditions(cond[i]);
51      }
52    }
53    return true;
54
55  } //start
```

A.4. Comparator

```
1  bool Comparator::compareTouched(Logic::CMessage* msg,
2                  Logic::CEntity* entity, Json::Value* json){
3
4    Logic::CTouchedMessage* tmsg =
5                   static_cast<Logic::CTouchedMessage*> (msg);
6
7
```

```
8   /* A touched message description can be composed by
9   two elements:
10   (a) Entity info
11   (b) Associated entity
12  We only have to compare the elements described in the
13  json.
14  If some element is not in the json, whatever value is
15  valid. */
16
17  // (a) Entity info
18  Json::Value entjs = json->get("entity", NULL);
19  std::string aux;
20  if (entjs != NULL){ //entity is described in the json
21    // entity name
22    aux = entjs.get("name", "").asString();
23    if (aux != "" && aux != entity->getName())
24    return false;
25
26    // entity position
27    aux = entjs.get("position", "").asString();
28    if (aux != ""){
29      std::stringstream ss;
30      ss << entity->getPosition();
31      if (aux != ss.str())
32      return false;
33    }
34
35    //entity type
36    aux = entjs.get("type", "").asString();
37    if (aux != "" && aux != entity->getType())
38    return false;
39
40  } // entity
41
42
43
44  // (b) Associated entity
45  Json::Value asocjs = json->get("associatedEntity", NULL);
46  if (asocjs != NULL){
47    // associated entity name
48    aux = asocjs.get("name", "").asString();
49    if (aux != "" && aux != tmsg->_entity->getName())
50    return false;
51
52    // associated entity position
53    aux = asocjs.get("position", "").asString();
54    if (aux != ""){
55      std::stringstream ss;
56      ss << tmsg->_entity->getPosition();
```

```
57          if (aux != ss.str())
58          return false;
59        }
60
61        // associated entity type
62        aux = asocjs.get("type", "").asString();
63        if (aux != "" && aux != tmsg->_entity->getType())
64        return false;
65
66      } // Associated entity
67
68
69      return true;
70
71  } // CompareTouched
```

Apéndice B

Artículo: Automatic Gameplay Testing for Message Passing Architectures

A continuación se muestra en artículo aceptado para su exposición en el II Congreso organizado por la Sociedad Española para las Ciencias del Videojuego (SECiVi). En él se muestra de manera resumida todo el sistema de testing propuesto, tanto la parte de detección de objetivos detallada en esta memoria como la parte de captura de trazas y su posterior reproducción y adaptación.

Automatic Gameplay Testing for Message Passing Architectures

Jennifer Hernández Bécares, Luis Costero Valero
and Pedro Pablo Gómez Martín

Facultad de Informática, Universidad Complutense de Madrid.
28040 Madrid, Spain
{jennhern,lcostero}@ucm.es
pedrop@fdi.ucm.es

Abstract. Videogames are highly technical software artifacts composed of a big amount of modules with complex relationships. Being interactive software, videogames are hard to test and QA becomes a nightmare. Even worst, correctness not only depends on *software* because *levels* must also fulfill the main goal: provide entertainment. This paper presents a way for automatic gameplay testing, and provides some insights into source code changes requirements and benefits obtained.

Keywords: gameplay testing, testing, automatisation

1 Introduction

Since their first appearance in the 1970s, videogames complexity has been continuously increasing. They are bigger and bigger, with more and more levels, and they tend to be non-deterministic, like Massively Multiplayer Online games where emergent situations arise due to players' interactions.

As any other software, videogames must be tested before their release date, in order to detect and prevent errors. Unfortunately, videogames suffer specific peculiarities that make classic testing tools hardly useful. For example, the final result depends on variable (nearly erratic) factors such as graphics hardware performance, timing or core/CPU availability. Even worst, correctness measure is complex because it should take into account graphic and sound quality, or AI reactivity and accuracy, features that cannot be easily compared.

On top of that, videogames are not just *software*. For example, it is not enough to test that physics is still working after a code change, but also that players can end the game even if a level designer has moved a *power up*. Videogames quality assurance becomes nearly an art, which must be manually carried out by skilled staff. Unfortunately, this manual testing does not scale up when videogames complexity grows and some kind of automatisation is needed.

This paper proposes a way for creating *automatic gameplay tests* in order to check that changes in both the source code *and levels* do not affect the global gameplay. Next section reviews the existing test techniques when developing

software. Section 3 describes the component-based architecture that has become the standard for videogames in the last decade and is used in section 4 for creating *logs* of game sessions that are replayed afterwards for testing. Section 5 puts into practise all these ideas in a small videogame, and checks that these tests are useful when levels are changed. The paper ends with some related work and conclusions.

2 Testing and Continuous Integration

Testing is a formal technique used to check and prove whether a certain developed software meets its quality, functional and reliability requirements and specifications. There are many testing approaches, each one designed for checking different aspects of the software. For example, a test can be done with the purpose of checking whether the software can run in machines with different hardware (*compatibility tests*), or whether it is still behaving properly after a big change in the implementation (*regression tests*). Alternatively, expert users can test the software in an early phase of the development (*alpha or beta tests*) to report further errors.

Unit testing is a particularly popular test type designed to test the functionality of specific sections of code, to ensure that they are working as expected. When certain software item or software feature fulfills the imposed requirements specified in the test plan, the associated unit test is passed. Pass or fail criteria are decision rules used to determine whether the software item or feature passes or fails a test. Passing a test not only leads to the correctness of the affected module, but it also provides remarkable benefits such as early detection of problems, easy refactoring of the code and simplicity of integration. Detecting problems and bugs early in the software development lifecycle translates in decreasing costs, while unit tests make possible to check individual parts of a program before blending all the modules into a bigger program.

Unit tests should have certain attributes in order to be good and maintainable. Here we list some of them, which are further explained in [1, Chapter 3]:

- **Tests should help to improve quality**.
- **Tests should be easy to run**: they must be fully automated, self-checking and repeatable, and also independent from other tests. Tests should be run with almost no additional effort and designed in a way that they can be repeated multiple times with the exact same results.
- **Tests should be easy to write and maintain**: test overlap must be reduced to a minimum. That way, if one test changes, the rest of them should not be affected.
- **Tests should require minimal maintenance as the system evolves around them**: automated tests should make change easier, not more difficult to achieve.

3 Game Architecture

The implementation of a game engine has some technological requirements that are normally associated to a particular game genre. This means that we need software elements specifically designed for dealing with different characteristics of a game like the physics system, the audio system, the rendering engine or the user input system. These software pieces are used by interactive elements of the game like the avatar, non-player characters (NPCs) or any other element of the game logic with a certain behaviour. The interactive elements of the game are called "entities", and they are organised inside game levels so that the user experience is entertaining and challenging but still doable. Entities are specified in external files that are processed during the runtime. This way, level designers can modify the playability of the game without involving programmers.

One of the most important tasks of a game engine is the management of the entities that are part of the game experience. An entity is characterised by a set of features represented by several methods and attributes. Following an object-oriented paradigm, each feature is represented by one or more classes containing all the necessary methods and attributes. Connecting the different classes that define the features of an entity leads to different game engine architectures, which can be implemented in very different ways.

The classical way of relating the different entity features is using inheritance. This way, an entity would be represented by a base class that will inherit from multiple classes, granting the entity different features. This method, in addition to the amount of time it requires to design a class structure and hierarchy, present certain additional problems described in [4]:

- **Difficult to understand:** The wider a class hierarchy is, the harder it is to understand how it behaves. The reason is that it is also necessary to understand the behaviour of all its parent classes as well.
- **Difficult to maintain:** A small change in a method's behaviour from any class can ruin the behaviour of its derived classes. That is because that change can modify the behaviour in such a way that it violates the assumptions made by any of the base classes, which leads to the appearance of difficult to find bugs.
- **Difficult to modify:** To avoid errors and bugs, modifying a class to add a method or change some other cannot be done without understanding all the class hierarchy.
- **Multiple inheritance:** It can lead to problems because of the diamond inheritance, that is, an object that contains multiple copies of its base class's members.
- **Bubble-up effect:** When trying to add new functionalities to the entities, it can be inevitable to move a method from one class to some of its predecessors. The purpose is to share code with some of the unrelated classes, which makes the common class big and overloaded.

The most usual way to solve these problems is to replace class inheritance by composition or aggregation associations. Thereby, an entity would be composed

by a set of classes connected between them through a main class that contains the rest of them. These classes are called *components*, and they form entities and define their features.

Creating entities from a set of components is called *Component-Based Architecture*. It solves all the problems mentioned before, but because it does not have a well-defined hierarchy, it is necessary to design a new mechanism of communication between components. The proposed mechanism is based on the use of a message hierarchy that contains useful information for the different components. Whenever a component wants to communicate with any other component, the first one generates a message and sends it to the entity that owns the receiver component. The entity will emit a message to all of its components, and each of them will accept or reject the message and act according to the supplied information. This technique used for communicating is called *Message Passing*.

Message passing is not only important for communicating between components, but also for sending messages from one entity to another. These messages between entities are essential when designing the playability of the game. The reason for using messages between entities is that they need to be aware of the events and changes in the game in order to respond accordingly.

4 Recording Games Sessions for Testing

Traditional software tests are not enough to check all the features that a game has. Things such as playability and user experience need to be checked by beta testers, who are human users that play the game levels over and over again, doing something slightly different each time. Their purpose is to find bugs, glitches in the images or incorrect and unexpected behaviours. They are part of the Software Quality Assurance Testing Phase, which is an important part of the entire software development process. Using beta testers requires a lot of time and effort, and increases development costs. In fact, testing is so important and expensive that has become a business by itself, with companies earning million of dollars each year and successfully trading on the stock market[1].

We propose an alternate form of testing, specifically designed for message-passing architectures with a component-based engine design. The objective is to have "high-level unit tests", based on the idea of reproducing actions to pass the test even when the level changes. To achieve that, we record game sessions and then execute the same actions again, adjusting them slightly if necessary so that the level can still be completed after being modified. Then, we check if the result of this new execution is the expected. Next sections give a detailed explanation on how to do this.

4.1 Using a New Component to Record Game Sessions

In order to record game sessions easily, having a component-based design is a great advantage. Our solution is based on the creation of a new component called

[1] `http://www.lionbridge.com/lionbridge-reports-first-quarter-2015-results/`, last visited May, 2015.

`CRecorder` added to any of the existing entities in the game. After that, when an entity sends a message to all of its listeners, the `CRecorder` component will also receive the message and act accordingly.

For example, a component aimed at recording actions in a game needs to be registered as a listener of the entity Player. Thus, whenever an action takes place in the game the component will be notified. Once the component has been created and the messages handled, saving all the actions to an external file is an easy task. Two different kind of files can be generated with this `CRecorder` component. One of them contains which keys have been pressed and the mouse movements, and the other one contains interesting events that happened during the gameplay, such as a switch being pressed by the player.

4.2 Loading Recorded Game Sessions and Replicating The State

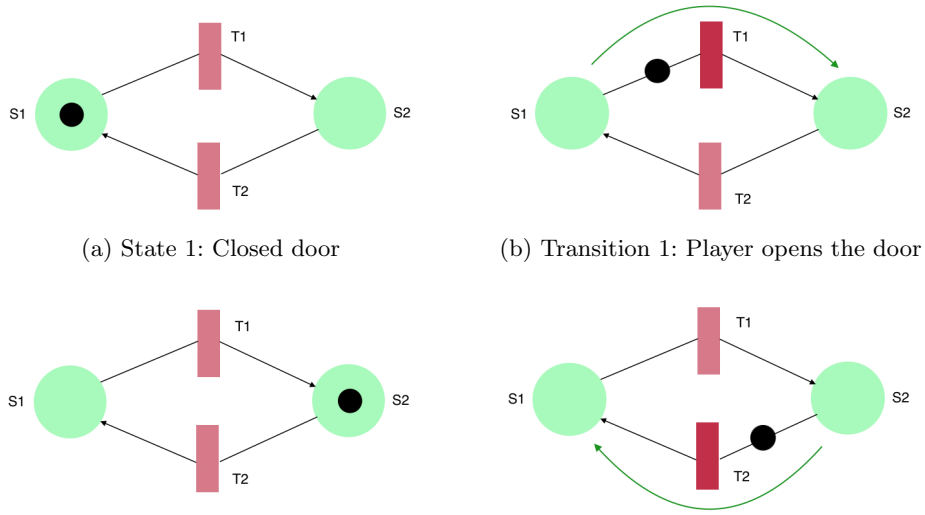
The goal of recording a previous sessions of the game is to avoid manually playing it several times if the changes are not significant. In this alternative, a file containing all the actions that the beta tester executed before is loaded into the game. These actions are used to replay the game in a test environment. Loading recorded game sessions and replaying them contributes towards having repeatable and automated tests, which were some of the advisable attributes of unit tests mentioned in section 2.

Some of the attributes of a game are the actions that can happen, the physical map or the state of the game. When the objective of recording a game is replaying it afterwards, it is necessary to think about what attributes need to be stored in the log. As an example, imagine an action in which a player picks up a weapon from the ground. To replay that, it is required to know which player (or entity) performs the action, what action is taking place and the associated entity (in this case, the weapon). Another important thing to take into account is the time frame for completing the action and the state of the game when it happens. A player cannot take a weapon if the weapon is not there or if he is not close enough to take it. Therefore, the state needs to be as close as possible when the time frame approaches so that replicating the action is feasible. On the other hand, storing the position of the weapon is not required, as using that position could lead to replaying a wrong action if the map changes. That information is stored in a different file (a map file), it is loaded into the game and it can be accessed during the run time.

With the purpose of modeling all these attributes, we use a powerful representations called Timed Petri nets, which can be very helpful to replay recorded games.

4.3 Modeling the Game With Timed Petri Nets

Petri nets [2, 3] are modeling languages that can be described both graphically and mathematically. The graphical description is represented as a directed graph composed by nodes, bars or squares, arcs and tokens. Elements of a Petri net model are the following:



(a) State 1: Closed door

(b) Transition 1: Player opens the door

(c) State 2: Opened door

(d) Transition 2: Player closes the door

Fig. 1. Modeling the actions of opening and closing a door with a Petri net.

- **Places**: They are symbolized by nodes. Places are passive elements in the Petri net and they represent conditions.
- **Transitions**: Bars or squares represent transitions, which are the actions or events that can cause a Petri net place to change. Thus, they are active elements. Transitions are enabled if there are enough tokens to consume when the transition fires.
- **Tokens**: Each of the elements that can fire a transition are called tokens. A discrete number of tokens (represented by marks) can be contained in each place. Together with places they model system states. Whenever a transition is fired, a token moves from one place to another.
- **Arcs**: Places and transitions are connected by arcs. An arc can connect a place to a transition or the other way round, but they can never go between places or between transitions.

Figure 1 shows an example on how to model the actions of opening and closing a door with a Petri net. 1a shows the initial state, in which a door is closed and a token is present. 1b shows the transition that takes place when a player opens the door. Then, the new state becomes 1c, making the token move from S1 to S2. If then the player performs another action and closes the door (showed in transition 1d), the token returns to the initial state again, 1a. Notice that in figures 1b and 1d the tokens are in the middle of one of the arcs. Petri Net models do not allow tokens to be out of places, but in this example they have been put there to highlight the movement of the token.

Classic Petri nets can be extended in order to introduce an associated time to each transition. When transitions last more than one time unit, they are called timed Petri nets. Introducing time in the model is essential for replaying games because actions are normally not immediate. For instance, if we want to replay an action such as "opening a door", firstly the player needs to be next to the door, and then perform the action of opening it. That means that the transition could be much longer than just a time unit, and other actions could be in progress at the same time. For that reason, modeling the game as a timed Petri net makes it easier than modeling it as a state machine.

After loading a recorded file to the game with the purpose of replaying it, actions need to be performed in a state as close as possible as the original state. Moreover, actions are normally ordered: a player cannot walk through a door if the door has not been opened before. In practice, some actions cannot be performed until some other actions have finished.

If the game has several entities capable of firing transitions, they can be represented as different tokens in the Petri net model. A token firing a transition may cause some other tokens to change their place, which is exactly what happens in games. Actions may affect several entities, not just one, so using Petri nets to model that behaviour seems to be reasonable.

When we detect that a player made an action in the first execution of the game and the corresponding Petri net transition is enabled (it is possible to complete the action), the appropriate messages have to be generated and injected to the application. There is no difference between messages generated when a real person is playing and the simulated messages injected. Components accept the messages, process them and respond accordingly. For that reason, the resulting file should be exactly the same and it is possible to see that the actions that are happening in the game have not changed.

4.4 Replaying Game Sessions and Running Tests

There are two possible ways of replaying a recorded game session: replicating the exact movements of the user or trying to reproduce specific actions using an artificial intelligence (AI). Replicating game sessions can be useful when we want to make compatibility tests (running tests using different hardware) or regression tests (introducing software improvements but not design changes). However, replicating game sessions consists on simulating the exact keys pressed by the user. These tests are very limited, since the slightest changes on the map make them unusable. Reproducing specific actions can solve this limitation. Saving detailed traces of the game sessions that we are recording gives us the chance to use that information to make intelligent tests using an AI. That way, we can still use the recorded game sessions to run tests even if the map also changes.

Once that replaying games is possible, it can be used to design and run tests. An input file with information for the tests can be written. In that file, the tester can define several parameters:

- **Objectives**: the tester can specify which messages should be generated again and if they should be generated in order or not. If those messages are generated, it means that the particular objective is fulfilled.
- **Maximum time**: sometimes it will not be possible to complete one of the tasks or objectives, so the tester can set a maximum time to indicate when the test will interrupt if the objectives are not completed by then.
- **User input file**: the name of the file containing all the keys pressed when the user was playing and the associated time.
- **Actions input file**: the name of the file with all the high level actions that the user performed, when he did them and the attributes of those actions.

Using those two ways of replaying the game can lead to the generation of very different tests. It is also possible to combine both ways and run a test that reproduces the actions in the input file and if and only if the AI does not know how to handle a situation it replicates the movements in the user file.

Several different tests can be run to check whether it is possible to complete a task. If any of them succeeds, then the objective is fulfilled. It is also possible to launch the game more than once in the same execution so that various tests are run and the objectives checked.

5 Example of Use: Time and Space

Time and Space is a game developed by a group of students of the *Master en Desarrollo de Videojuegos* from the Universidad Complutense de Madrid. This game consists on several levels with switches, triggers, doors and platforms. The player has to reach the end of each level with the help of copies from itself. These copies will step on the triggers to keep doors opened or platforms moving while the player goes through them. Sometimes clones will even have to act as barriers against enemies to keep them from shooting the player. Also, there are platforms in the game that cannot be traversed by the player, but only by his copies. These copies are not controlled by the person that is playing the game. Their movements are restricted so they just reproduce the actions they made in the previous execution of the level, before the player copied itself.[2]

When the tester activates the recording of traces, a file in json format is generated. This format was chosen because of its simplicity to read and write to a file using different styles (named objects or arrays). This file contains the information of the execution: what actions were performed, when they took place and the entities that were associated to that action (player, switch, enemy, etc). We also created a configuration file where the tester can choose if he wants to record the game or load previous executions that were recorded before. If he chooses to reproduce something recorded, then he can specify the name of the file that contains the actions that are going to be loaded and reproduced.

We have recorded game traces from a level that the user played and completed successfully, that is, reaching the end of it. To check that the programmed replay

[2] Full gameplay shown at `https://www.youtube.com/watch?v=GmxV_GNY72w`

system works with Time and Space, we reproduced those exact traces without any modification in the same level. However, the map was slightly changed in those new executions. Some of the tests that we carried out were the following:

- Firstly we tried to change the map and move the switches to reachable places from the position in which they were initially placed. We then repeated the same test but we also changed the end of level mark. Both of the tests were still feasible after all the changes, and by replaying the same traces it is still possible to complete the objectives and reach the end of the level. Another test we made consisted in placing one of the switches behind a closed door. In this case, we could see that the player detected that the new position was not reachable and therefore he did not move from the position he had before detecting he had to press the switch.
- After that, we recorded traces from a level in which the player needs three different copies to win the level. To reach the end of the level, the player has to go through a moving platform and some rays have to be deactivated. That is why the player needs his clones. If the player tries to go through the rays before deactivating them, he dies. To make the tests, several changes were added to the map. For example, we tried to pass the test after interchanging the switches between them. By doing that, they were closer or further away from the player. Running the test allows us to see that despite of all the changes, it is still possible to complete the level without difficulties. We recorded a video that shows the reproduction of two of these tests, and it can be seen at [5].

Because the game has been implemented following a standard component-based architecture, it was not necessary to make major changes in it. To record the game session we only added the `CRecorder` component as described in 4.1, which receives all the messages generated during the gameplay. The code for the `CRecorder` component and the required changes made in the original implementation are about 8 KB. Moreover, two new modules of about 127 KB were created for recording and replaying the messages. These modules were designed with a general purpose and only slightly modified to work with this game.

6 Related Work

With systems growth in size and complexity, tests are more difficult to design and develop. Testing all the functions of a program becomes a challenging task. One of the clearest examples of this is the development of online multiplayer games [6]. The massive number of players make it impossible to predict and detect all the bugs. Online games are also difficult to debug because of the non-determinism and multi-process. Errors are hard to reproduce, so automated testing is a strong tool which increases the chance of finding errors and also improves developers efficiency.

Monkey testing is a black-box testing aimed at applications with graphical user interfaces that has become popular due to its inclusion in the Android

Development Kit[3]. It is based on the theoretical idea that a monkey randomly using a typewriter would eventually type out all of the Shakespeare's writings. When applied to testing, it consists on a random stream of input events that are injected into the application in order to make it crash. Even though this testing technique blindly executes the game without any particular goals, it is useful for detecting hidden bugs.

Due to the enormous market segmentation, again specially in the Android market but more and more also in the iOS ecosystem, automated tests are essential in order to check the application in many different physical devices. In the cloud era, this has become a service provided by companies devoted to offer cloud-based development environments.

Unfortunately, all those testing approaches are aimed at *software*, ignoring the fact that games are also maps, levels and challenges. We are not aware of any approach for automatic gameplay testing as described in this paper.

7 Conclusions and Future Work

Although some methods for automatising gameplay tests exist, they are aimed at checking software aspects, not taking into account the necessity of checking that both the maps and levels are still correct. Because these levels and maps also evolve alongside software while developing games, finding a way to run automatic tests to check that all the modifications introduced into levels are consistent is a must.

In this paper we have introduced a proposal on how to carry out these tests. Taking advantage of the component-based architecture, we have analised the cost of introducing the recording and replaying of traces in games, which allow us to automatically repeat gameplays after modifying the levels. This proposal has been tested with a simple game, proving the viability of the idea.

Despite the promising results, the work we carried out is still on preliminary stages. It is still necessary to test this technique in more complex games, as well as proving its stability to more dramatic changes in them.

References

1. Meszaros, G: XUnit test patterns: refactoring test code. Addison-Wesley, (2007)
2. Popova-Zeugmann, L: Time and Petri Nets. Springer-Verlag Berlin Heidelberg (2013)
3. Estevão Araújo, M., Roque, L.: Modeling Games with Petri Nets. DIGRA2009 - Breaking New Ground: Innovation in Games, Play, Practice and Theory (2009)
4. Gregory, J.: Game Engine Architecture. A K Peters, Ltd. (2009)
5. Hernández Bécares, J., Costero Valero, L.: Time and Space: Replaying the Game. `https://www.youtube.com/watch?v=1OBlBKly1pk`
6. Mellon, L.: Automatic Testing for Online Games. Game Developers Conference, 2006.

[3] `http://developer.android.com/tools/help/monkey.html`

Bibliografía

Y así, del mucho leer y del poco dormir,
se le secó el celebro de manera que vino
a perder el juicio.

Miguel de Cervantes Saavedra

Blázquez, Á., Pérez, A., Cordón, A., Bergareche, I., Glaria, J., Pazos, F. y Rodríguez, L. I. Time and Space, The Game, `https://timeandspacethegame.wordpress.com/`. 2013-2014.

Boehm, B. A spiral model of software development and enhancement. *Software Engineering Notes, ACM*, 1986.

Gómez Martín, M. A. *Arquitectura y metodología para el desarrollo de sistemas educativos basados en videojuegos*. Tesis Doctoral, Facultad de informática. UCM, 2007.

Gregory, J. *Game Engine Architecture*. A K Peters, Ltd., 2009.

Hernández Bécares, J. *Ejecución y adaptación de trazas de juegos para la automatización de pruebas*. Proyecto Fin de Carrera, Facultad de informática, UCM, 2015.

Hernández Bécares, J., Costero Valero, L. y Gómez Martín, P. P. An approach to gameplay testing for message passing architectures. *CEUR-WS.org*, 2015.

id Software. Doom. 1993.

IEEE Std 829. Software engineering technical committee of the ieee computer society. Informe técnico, IEEE-SA Standard Board, 1998.

Lakos, J. *Large Scale C++ Software Design*. Addison Wesley, 1996. ISBN 0-201-63362-0.

McConnell, S. *Code Complete*, capítulo p. 29. Microsoft Press, 2004.

MEHTA, M., ONTAÑÓN, S., AMUNDSEN, T. y RAM, A. Authoring behaviors for games using learning from demostration. *Case-Based Reasoning for Computer Games worskshop*, 2009.

MELLON, L. Automatic Testing for Online Games. En *Game Developers Conference*. 2006.

MYERS, G. J. *The art of software testing*. Hoboken, N.J., 2004.

ONTAÑÓN, S., BONNETTE, K., MAHINDRAKAR, P., GÓMEZ-MARTÍN, M. A., LONG, K., RADHAKRISHNAN, J., SHAH, R. y RAM, A. Learning form human demonstrations for real-time case-based planning. *STRUCK-09 Workshop*, 2009.

ROLLINGS, A. y MORRIS, D. *Game Architecture and Design: A New Edition*. New Riders Publishing, 2004.

ROYCE, W. Managing the development of large software systems. *Proceedings of IEEE WESCON*, 1970.

VALVE SOFTWARE. Half life 1. 1998.

–¿Qué te parece desto, Sancho? – Dijo Don Quijote –
Bien podrán los encantadores quitarme la ventura,
pero el esfuerzo y el ánimo, será imposible.

Segunda parte del Ingenioso Caballero
Don Quijote de la Mancha
Miguel de Cervantes