UNIVERSIDAD AUTONOMA DE MADRID

Escuela Politécnica Superior

18|19

Trabajo fin de grado

Herramienta de diseño de juegos tipo mazmorra para Gamemaker Studio 2

Javier Gómez

Escuela Politécnica Superior
Universidad Autónoma de Madrid
C\Francisco Tomás y Valiente nº 11

excelencia Campus Internacional UAM CSIC

www.uam.es

UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Herramienta de diseño de juegos tipo mazmorra para Gamemaker Studio 2

Autor: Javier Gómez
Tutor: Carlos Aguirre

junio 2019

Javier Gómez
Herramienta de diseño de juegos tipo mazmorra para Gamemaker Studio 2

Javier Gómez
C\ Francisco Tomás y Valiente Nº 11

IMPRESO EN ESPAÑA – PRINTED IN SPAIN

A game isn't something that can be made by one person alone. Well, it's possible that a really hard-working game creator could finish a game all alone, but even then he would still need a player. Games grow and mature when they're created, played, and conveyed over and over.

Shigesato Itoi

Resumen

El objeto de este trabajo de fin de grado es el desarrollo de una aplicación de escritorio que facilite a los desarrolladores de videojuegos la tarea del diseño de niveles.

Palabras clave

Diseño de documento, LaTeX 2$_\varepsilon$, thesis, trabajo fin de grado, trabajo fin de master

Abstract

In our School a considerable number of documents are produced, as many aducational as research. Our students also contribute to this production through his final degree, master and thesis projects. The objective of this material is to facilitate the editing of all these documents and at the same time to promote our corporate image, facilitating the visibility and recognition of our center.

In this sense we have tried to design a style of LaTeX 2_ε that maintains a corporate image and with simple commands that allow to maintain the corporate image with the necessary quality without forgetting the needs of the author. For this, a set of simple commands have been created around complex packages. These commands allow you to perform most of the operations that a document of this type may need.

Likewise, you can control a little the design of the document through the options of the style but always maintaining the institutional image.

Keywords

Document design, LaTeX 2_ε, thesis, final degree project, final master project

Índice

Listas

Lista de algoritmos

Lista de códigos

Lista de cuadros

Lista de ecuaciones

Lista de figuras

Lista de tablas

Lista de cuadros

1

Introducción

Este Trabajo de Fin de Grado tiene como propósito el desarrollo de una aplicación de escritorio que servirá de herramienta para facilitar el desarrollo de videojuegos de tipo mazmorra.

La aplicación de escritorio *Maze Designer* se encargará de la tarea del diseño de niveles, permitiendo visualizarlos y editarlos como una gran estructura conjunta en lugar de como escenarios aislados y facilitando una interfaz lógica para la comprobación de la corrección de un diseño.

1.1. Motivación

En la actualidad, el género de videojuegos de tipo mazmorra ha adquirido bastante popularidad tanto entre grandes desarrolladoras como en el escenario independiente del desarrollo de videojuegos. Se han creado títulos como *Super Metroid (Nintendo, 1994)*, *Cave Story (Pixel, Nicalis, 2004)*, *Ori and the Blind Forest (Moon Studios, 2015)* y *Hollow Knight (Team Cherry, 2017)* entre muchos otros, que a día de hoy se consideran referentes e incluso juegos de culto.

Este tipo de juegos, centrados en la exploración y progreso a través de un entorno, presentan ciertas necesidades particulares a la hora de diseñar los espacios de juego al ser éstos un elemento tan importante para la experiencia del jugador. Además de deber cumplir con los requisitos visuales y estéticos del título en cuestión, los espacios de juego se diseñan de manera que el jugador necesite realizar ciertos logros, como avanzar en la trama u obtener un cierto objeto, para progresar y poder explorar nuevas zonas. De esta manera, los espacios de juego son diseñados para controlar el progreso del jugador, guiándole (sin que éste se de cuenta) hacia una experiencia determinada.
Para esto, se suelen emplear distintos patrones de bloqueo para dejar ciertas áreas inaccesibles hasta cumplir con unos requisitos determinados.
Estos bloqueos no son necesariamente bidireccionales, de manera que se le permite al jugador acceder a un área pero regresar a la zona anterior inmediatamente no es necesariamente posible. Éste es uno de los patrones más populares, empleado usualmente a modo de tutorial, pero empleado de forma errónea podría ocasionar que el jugador quedara atrapado en un área sin poder salir de la misma.

Además de esto, lo patrones para el desbloqueo de áreas más extendidos son el de power-ups y el de llaves consumibles. En particular este último también puede plantear situaciones en las que algunas áreas queden permanentemente inaccesibles para el jugador si los espacios de juego están diseñados erróneamente y éste hace una determinada exploración.

Debido los patrones problemáticos antes mencionados, el diseño de espacios de juego se vuelve muy costoso conforme éstos aumentan de tamaño, y más aún si el progreso del juego no es lineal. Por esto se plantea el desarrollo de una herramienta que facilite al diseñador la tarea de crear estos espacios de juego y, una vez acabados, los explore para asegurar el correcto diseño de los mismos, esto es, que el jugador no pueda realizar una exploración que como consecuencia le deje permanentemente atascado o que deje un área inaccesible sin remedio.

La intención de este trabajo de fin de grado es desarrollar tal herramienta para facilitar el desarrollo de videojuegos de tipo mazmorra.

1.2. Objetivos

El proyecto se centra en el desarrollo de una aplicación de escritorio que sirva como entorno de diseño de niveles de cualquier videojuego que, de una forma u otra, contenga una mazmorra.

La herramienta debe ser capaz de explorar el diseño creado por el usuario para asegurar que éste no presenta errores en el progreso del jugador y debe exportar los diseños a un motor de videojuegos escogido (*Game Maker Studio 2*) para continuar con el desarrollo del título en él.

Los objetivos principales a cumplir por la herramienta son:

1.– Una interfaz para editar el espacio jugable de forma visual.

2.– Una interfaz para delimitar las pantallas del juego que más tarde serán exportadas al motor de videojuegos.

3.– Una interfaz para colocar elementos del progreso lógico del jugador (llaves, bloqueos y punto de comienzo).

4.– La exportación del diseño elaborado por el usuario al motor Game Maker Studio 2 para continuar en éste el desarrollo del videojuego.

5.– La exploración del diseño para la detección de errores lógicos en el progreso del jugador diseñado implícitamente.

Para ello, se han desarrollado los principales módulos de la aplicación como proyectos de Qt independientes unos de otros en la medida de lo posible, que finalmente son integrados en una interfaz gráfica.

1.3. Estructura

Esta memoria del Trabajo de Fin de Grado está dividida en 6 capítulos además de esta introducción. Éstos son:

1.– **Estado del arte** donde detallaremos tecnología existente y herramientas utilizadas para tareas con cierta similitud a las partes de la aplicación a desarrollar, así como tecnologías utilizadas en el desarrollo de videojuegos que tienen directa relación con el producto que se construirá con esta aplicación. También se discutirá en este apartado ventajas e inconvenientes de cada una de las tecnologías que se mencionen.

2.– **Análisis de Requisitos** donde expondremos los requisitos mínimos, funcionales y no funcionales, que se han de cumplir por la aplicación, definiendo así el alcance de la misma.

3.– **Diseño** donde se planteará el diseño de la aplicación, el diagrama de clases y se discutirán las distintas decisiones de diseño tomadas en el desarrollo.

4.– **Desarrollo** donde se detallará el proceso seguido para el desarrollo, hablando de las metodologías empleadas, de dificultades encontradas y de las soluciones llevadas a cabo.

5.– **Integración, pruebas y resultados** donde se discutirá el resultado de la aplicación y se mostrarán las pruebas a las que ésta ha sido sometida para asegurar el correcto funcionamiento y el control de la calidad de la misma.

6.– **Conclusiones y trabajo futuro** donde se hablará del resultado obtenido, la utilidad de la herramienta desarrollada y de posibles mejoras y ampliaciones que podrían llevarse a cabo en el futuro.

2

Estado del Arte

Históricamente, el rol del diseñador de este tipo de juegos comenzaba dibujando en papel el diseño de los mismos, se empezaba por un esquema en forma de grafo en el que se describía el progreso lógico del jugador (qué requisitos debe cumplir para acceder al área siguiente) y más tarde se implementaban esta lógica del progreso en los niveles diseñados ya a nivel visual. Más adelante, se empezaron a utilizar herramientas software para sustituir el papel, pero las tareas permanecían divididas en un diseño lógico y de nivel, más cercano a lo que aparece en el producto final. Además de esto, la construcción de estos diseños en el juego, se ha llevado a cabo manualmente en distintos entornos de desarrollo y, en la actualidad, se utilizan los motores de videojuegos para este propósito.

2.1. Videojuegos de tipo Mazmorra

Los videojuegos de tipo mazmorra, englobando videojuegos que la industria ha bautizado como *metroidvania* y *zelda-like* entre otros, es un género de productos de ocio digital en los que el usuario controla a un avatar en un entorno que debe explorar, usualmente controlando la progresión del jugador a través de dicho entorno mediante puertas bloqueadas u obstáculos que debe sortear consiguiendo llaves o habilidades nuevas para su avatar. El foco central de estos juegos es la exploración y la progresión a través del escenario, a diferencia de otros géneros de videojuegos, donde el foco puede estar centrado en la trama, asumir el rol de un determinado personaje o en simular una experiencia como puede ser un deporte. El diseño de cómo es la exploración depende del desarrollador y de la visión que tenga sobre su producto. al hablar de la exploración de estos juegos, se suelen emplear términos como exploración lineal, exploración ramificada y backtracking para describirlos. Durante la exploración de los escenarios del juego, el jugador puede recolectar diversos objetos que, dependiendo del juego, tienen un uso u otro, sin embargo se repiten mucho los patrones de objeto consumible y power-up ya que estos en particular pueden utilizarse para el control del progreso del jugador.

Algunos de los títulos de éste género más notables de la industria son:

2.2. Diseño Gráfico

Las herramientas de diseño gráfico, que no son capaces de realizar ningún tipo de abstracción lógica, sino que son empleadas sola y únicamente para representar el aspecto del espacio de juego de forma gráfica, con el nivel de detalle que el usuario decida. La actual tecnología empleada para este tipo de diseño, varía desde el diseño en papel, hasta herramientas software como:

1.– **Photoshop** La herramienta de diseño gráfico profesional de *Adobe*, permite detalladas manipulaciones de imágenes para diseñar el espacio de juego e incluso permite el uso de capas para mayor utilidad.

2.– **Gimp** La herramienta open source de diseño gráfico que cumple, a los efectos del diseño de niveles de un videojuego, los mismos propósitos que *photoshop*

3.– **Paint** La herramienta de dibujo de windows por defecto, con las limitaciones que presenta como herramienta primitiva, pero que puede servir para el diseño de espacios jugables.

4.– **Tiled** https://www.mapeditor.org/ Es un editor de mapas orientado específicamente al diseño de videojuegos, pero su función es únicamente la colocación de elementos gráficos para la construcción de los niveles.

Todas las herramientas mencionadas presentan el problema en común de que no aplican ningún tipo de abstracción lógica al diseño, de manera que la comprobación de que un éste sea correcto queda a discreción del usuario. Además de esto, la integración de los diseños con un motor de videojuegos en específico, a excepción de *Tiled* es íntegramente manual.

2.3. Diseño Lógico

Las herramientas del diseño lógico, se basan en la abstracción del progreso del jugador en forma de grafo. Dado que el progreso se modela como un grafo con aristas unidireccionales, la tecnología orientada a cubrir estos requerimientos consiste en software de visualización y manipulación de grafos. Para estas tareas específicas, existen las siguientes herramientas:

1.– **graphviz** http://www.graphviz.org/ Se trata de una aplicación para diseñar y visualizar información abstracta estructurada en grafos.

2.– **librería diagrams de Haskell** http://hackage.haskell.org/package/diagrams Se trata de una librería de haskell con la que se pueden declarar y visualizar grafos. Es una potente librería de fácil utilización, sin embargo tiene el defecto ser necesario programar en haskell, lo cual es una tarea que puede escaparse de las competencias de un diseñador o puede llegar a ser demasiado complejo para la tarea en cuestión.

3.– **Finite State Machine Designer** http://madebyevan.com/fsm/ Se trata de una aplicación web para diseñar grafos de forma visual cuyo objetivo principal es el diseño de autómatas finitos. Sin embargo, debido a la naturaleza de grafo de los diseños realizados y a la sencillez de utilización, puede cumplir con los requisitos de esta categoría de diseño, aunque su naturaleza de aplicación web puede resultar incómoda en especial para diseños muy complejos.

Todas las herramientas mencionadas comparten el defecto de ser herramientas puramente abstractas para representar el progreso en forma de grafo, sin preocuparse de la visualización de los niveles y, por tanto, la implementación de la lógica diseñada con estas herramientas a niveles de un producto es una tarea que el diseñador debe realizar manualmente.

2.4. Motores y Entornos

En la actualidad, la producción de videojuegos se lleva a cabo en motores de videojuegos, que conforman un entorno de desarrollo. Algunos de estos motores son de propósito general y permiten el desarrollo de cualquier tipo de juego.

1.– **Construct** Es un motor de videojuegos creado por Scirra Ltd. Se trata de un motor de propósito general orientado a un público con pocos conocimientos técnicos en el desarrollo de videojuegos. Permite crear videojuegos en dos dimensiones.

2.– **MonoGame** Es una implementación del framework XNA4 de Microsoft, desarrollada por MonoGame Team. Se trata de una herramienta open-source que permite la creación de videojuegos tanto en dos como en tres dimensiones.

3.– **Unity** Es un motor de videojuegos creado por Unity technologies. Se trata de un motor de propósito general que permite la creación de videojuegos tanto en tres como en dos dimensiones.

4.– **Unreal Engine** Es un motor de videojuegos creado por Epic Games. Es también un motor de propósito general para creación de juegos tanto en tres como en dos dimensiones.

5.– **Game Maker Studio 2** Es un motor de videojuegos creado por YoYo Games. También de propósito general pero limitado a la creación de juegos en 2D.

De entre los motores enumerados, escogeremos *Game Maker Studio 2* por ser de propósito general, suficientemente completo y, al limitar su ámbito a los juegos en dos dimensiones, se simplifican muchas de las complicaciones que presentan los juegos en tres dimensiones, entre las que se encuentra la creación de escenarios, que es el propósito principal que abarca la herramienta Maze Designer.

2.5. C++ y Qt

C++ es un lenguaje de programación multiparadigma cuyo origen es la extensión del lenguaje de programación C al paradigma de la programación orientada a objetos. Tras esto, conforme C++ ha ido evolucionando, se han incorporado facultades propias de otros paradigmas, de manera que ahora engloba a la programación estructurada, a la programación orientada a objetos, a la programación genérica y en las versiones más actualizadas contiene ciertos conceptos de la programación funcional. Como características distintivas, C++ permite al programador el uso de punteros, gestión de la memoria y también permite la sobrecarga de operadores. Actualmente, C++ se encuentra en su versión C++17, destacando la versión C++11 que realizó mejoras a nivel del núcleo del lenguaje. C++ es un lenguaje compilado que destaca por su alto rendimiento, sus capacidades para paralelizar tareas y por la gestión de la memoria por parte del programador.

Junto a C++, se encuentra Qt, un framework que no pertenece al estándar pero de uso muy extendido para la tarea del desarrollo de aplicaciones de escritorio. Qt incluye librerías y APIs escritas en C++ que abarca ámbitos desde renderización, cálculos geométricos, multithreading, control del input de dispositivos hardware y hasta realización de peticiones http. Actualmente, Qt está en su versión

5.12 y ofrece un lenguaje propio para declaración de componentes gráficos (QML) y un binding con python.

2.6. Conclusiones

Las herramientas orientadas al diseño lógico no cubren los requisitos estéticos y visuales que sí cumplen aquellas orientadas al diseño gráfico, sin embargo estas últimas carecen de la abstracción lógica que aportan las primeras. En cuanto a los motores, al ser de propósito general y no existir ninguno para un género tan concreto como los juegos de tipo mazmorra, no cubren las necesidades de diseño de éstos juegos tan específicamente, dejando la tarea del diseño a los desarrolladores.

Es por esto que la herramienta *Maze Designer* se plantea como una opción que integra la abstracción lógica de las herramientas de manipulación de grafos (de forma transparente para el diseñador) con el cumplimiento de ciertos requisitos visuales que cumplen las herramientas de diseño gráfico, ofreciendo además una conexión a un motor de videojuegos donde se puede continuar con el desarrollo una vez terminado el diseño de los espacios de juego.

3

Definición del proyecto

3.1. Alcance

La aplicación será una herramienta de diseño de niveles, por tanto abarcará las tareas de construcción de espacios, la instanciación solamente en el diseño de agentes lógicos como llaves y puertas y la posterior exploración del diseño para la comprobación de que éste es correcto, es decir, no existe una cadena de acciones o exploración que tiene como consecuencia que el jugador no pueda acceder a algún área del juego. De la misma manera, la aplicación deberá poder exportar estos diseños a un proyecto de *Game Maker Studio 2*. Esta exportación creará tantos ficheros .yy como habitaciones hayan sido definidas en el diseño, y al contenido de dichos ficheros .yy, solo se exportarán los objetos sólidos que conforman las paredes, suelos y techos de la habitación en cuestión. Esto es, las instancias lógicas antes mencionadas que se encuentren dentro de una habitación no serán exportadas al fichero .yy dado que estas instancias lógicas, en la implementación del juego, serán objetos cuyo funcionamiento deberá implementar el desarrollador dentro del propio motor, ajustándose a los requisitos del juego como producto final. Es decir, esta herramienta no es un editor de niveles, pues no cubre la funcionalidad de colocar instancias en los espacios de juego.

Además de esto, a la hora de realizar la exploración, la herramienta solamente tendrá en cuenta la vecindad de componentes conexas del diseño a través de las instancias lógicas de las puertas. Es decir, la herramienta no tendrá en cuenta mecánicas de juego como “el jugador no puede saltar más de una altura X y por tanto no puede llegar a determinados lugares” a la hora de determinar qué áreas son accesibles o no. Para modelar algo así, el desarrollador debería partir el espacio de juego con una puerta cuya condición de apertura sea “poder saltar más de una altura X”, es decir, “el jugador debe poseer un power-up que permitiría saltar la altura X”.

3.2. Requisitos funcionales

3.2.1. Requisitos de persistencia

RF-1.– Guardar el diseño

El usuario debe poder guardar los datos del diseño sobre el que está trabajando, especificando el nombre del fichero. Dicho fichero se guardará con la extensión. *.maze*

RF-2.– Cargar el diseño

El usuario debe poder cargar los datos de un diseño previamente guardado.

RF-3.– Crear nuevo diseño

El usuario debe poder crear un nuevo diseño vacío sobre el que trabajar.

3.2.2. Requisitos de contexto

RF-4.– cambio de contexto del canvas

El usuario podrá cambiar el contexto actual del canvas para realizar tareas diferentes:

RF-4.1.– Contexto de edición de *espacios jugables*

En este contexto el usuario podrá realizar todas las operaciones que tengan que ver con la creación, edición, selección y eliminación de los *espacios jugables*.

RF-4.2.– Contexto de edición de pantallas

En este contexto el usuario podrá realizar todas las operaciones que tengan que ver con la creación, selección y eliminación de las regiones denominadas pantallas.

RF-4.3.– Contexto de edición de elementos llave y de elementos bloqueadores

En este contexto el usuario podrá realizar todas las operaciones que tengan que ver con la creación, selección y eliminación tanto de elementos llave como de elementos bloqueadores.

3.2.3. Requisitos del canvas

RF-5.– Dibujar *espacios jugables*

El usuario, en el contexto de *diseño* debe poder dibujar *espacios jugables* en el canvas haciendo click en un punto del canvas y soltando el click en otro.

RF-5.1.– Anexión de *espacios jugables*

Si el conjunto de los *espacios jugables* que intersecan con el recién creado es no vacío, entonces se realizará la unión de los *espacios jugables* preexistentes y el nuevo *espacio jugable*

RF-5.2.– Indexación de *espacios jugables*

Todo *espacio jugable* debe ser indexado por una estructura para su futura transformación en grafo.

RF-5.3.– Vértices de un *espacio jugable*

Para todo *espacio jugable* deben definirse sus vértices en el canvas.

RF-5.4.– Grid de coordenadas Los vértices de todo *espacio jugable* deben pertenecer a un *grid* de coordenadas.

RF-6.– Eliminación de *espacios jugables*

El usuario debe poder extraer áreas rectangulares de un *espacio jugable*, de manera que el resultado respete la definición de *espacio jugable*.

RF-7.– Crear un elemento bloqueador

El usuario debe poder crear un elemento bloqueador dibujando una línea en el canvas.

RF-7.1.– Especificación del tipo de elemento bloqueador

El elemento bloqueador creado por defecto será de tipo bidireccional, con ambas condiciones siendo la condición vacía.

RF-7.2.– Cálculo de vecindad al crear un elemento bloqueador

En el caso de que la creación de un elemento bloqueador divida un *espacio jugable* en dos, se han de indexar estos dos nuevos *espacios jugables* y se ha de definir su vecindad teniendo en cuenta el tipo de elemento bloqueador y sus condiciones de apertura.

RF-8.– Seleccionar un elemento bloqueador

El usuario debe poder seleccionar un elemento bloqueador preexistente del canvas.

RF-9.– Eliminar un elemento bloqueador

El usuario debe poder eliminar un elemento bloqueador preexistente del canvas.

RF-10.– Crear un elemento llave

El usuario debe poder crear un elemento llave, instanciando un tipo de llave en el canvas.

RF-11.– Seleccionar un elemento llave

El usuario debe poder seleccionar una instancia de llave del canvas.

RF-12.– Eliminar un elemento llave

El usuario debe poder eliminar una instancia de llave del canvas.

RF-13.– Zoom

El usuario debe poder acercar y alejar la vista que tiene del canvas.

RF-14.– Desplazar vista

El usuario debe poder desplazar la vista que tiene del canvas.

3.2.4. Requisitos de llaves

RF-15.– Crear tipos de llaves

El usuario debe poder crear tipos de llaves especificando su nombre y su clase (si es un power-up o no).

RF-16.– Eliminar tipos de llaves

El usuario debe poder eliminar tipos de llaves previamente creados.

RF-17.– Modificar el nombre de un tipo de llave

El usuario debe poder modificar el nombre que se le hubiera asignado a un tipo de llave.

RF-18.– Modificar la clase de un tipo de llave

El usuario debe poder modificar la clase de un tipo de llave, es decir, modificar si es o no un power-up

RF-19.– Seleccionar un tipo de llave

El usuario debe poder seleccionar un tipo de llave de entre la lista de todos los tipos de llaves.

3.2.5. Requisitos de condiciones de apertura

RF-20.– Edición de condiciones de apertura

Cuando el usuario haya seleccionado un elemento bloqueador, debe ser capaz de describir las condiciones bajo las cuales este se abrirá. Dicha descripción debe ser escrita siguiendo la sintaxis de un lenguaje.

RF-20.1.– sintaxis del lenguaje

el lenguaje en el que se especifican las condiciones debe seguir la sintaxis definida en el apéndice A

RF-20.2.– Condición vacía

La condición vacía del lenguaje se interpreta como una satisfacible siempre.

RF-20.3.– Orden de precedencia

A la hora de evaluar una expresión del lenguaje, se debe respetar el orden de precedencia habitual: resolución del contenido de los paréntesis primero y conexión de izquierda a derecha.

RF-21.– Sintetización de condiciones

El sistema debe ser capaz de sintetizar las condiciones escritas por el usuario, siguiendo la sintaxis del lenguaje de descripción de condiciones, en clases compatibles con las aristas de un grafo, para inducir en estos una exploración condicionada.

3.2.6. Requisitos de comprobación de la corrección

RF-22.– Comprobación

El usuario debe poder comprobar que su diseño no contenga errores de tipo *dead lock*.

RF-22.1.– Conversión a grafo

El diseño se debe poder convertir a una estructura de tipo grafo para poder realizar la exploración del mismo

RF-22.2.– Notificación de errores

Si el diseño comprobado contiene un error de tipo *dead lock* se ha de notificar al usuario del camino escogido para encontrar dicho error.

3.2.7. Requisitos de conexión con Game Maker Studio 2

RF-23.– Creación de pantallas

El usuario debe poder seleccionar regiones rectangulares del canvas que representarán las pantallas del juego que está diseñando. Al crearse, las pantallas tendrán un nombre asignado por defecto.

RF-24.– Seleccionar una pantalla

El usuario debe poder seleccionar una pantalla.

RF-25.– Cambiar el nombre de una pantalla

El usuario debe poder cambiar el nombre de una pantalla.

RF-26.– Eliminar una pantalla

El usuario debe poder eliminar un región que representa una pantalla de su juego

RF-27.– Conversión a ficheros .yy.room

El sistema debe ser capaz de transformar las pantallas a ficheros .yy.room de *Game Maker Studio 2*, con las dimensiones y nombre determinados por el diseño del usuario. En estos ficheros figurarán solamente los bloques sólidos que cubrirán los *espacios no jugables* del diseño que haya dentro de cada una de las pantallas.

RF-27.1.– Construcción de un área con el mínimo número de rectángulos

Los *espacios no jugables*, que tendrán forma poligonal cuyos ángulos serán múltiplos de 90 grados, han de cubrirse del menor número de rectángulos posible. Estos rectángulos, en la traducción a ficheros .yy.room, se representarán como bloques sólidos del juego.

3.3. Requisitos no funcionales

4

Diseño

4.1. Tecnologías y Estándares

Para el desarrollo de esta aplicación, ha sido necesario el uso de varios estándares y tecnologías que interactuasen entre si. La codificación se desarrolla en C++ junto con Qt, pero se debía atender a necesidades de compatibilidad con Game Maker Studio 2, mediante el estándar Json, así como respetar un formato de persistencia que fuera amigable con un sistema de control de versiones como git. Así pues, el conjunto de las tecnologías empleadas y cómo han sido relacionadas se explica en el siguiente gráfico y en las siguientes subsecciones:

4.1.1. Qt

Qt es el framework de C++ empleado para la implementación, tanto lógica como de interfaz de usuario, de esta aplicación.

Se han empleado diversas clases de Qt para la codificación de la aplicación, desde clases generales como QString, QList, etc. hasta clases más específicas de ciertos ámbitos como QRectF, QPolygonF, QPainterPath para las operaciones geométricas, QJsonObject, QJsonDocument para las operaciones de persistencia y generación de ficheros .yy y distintas clases que heredan de QWidget y QLayout para la construcción de la interfaz gráfica de usuario. También se ha hecho uso del sistema de slots y señales que proporciona Qt para la comunicación entre las distintas componentes de la aplicación y para la interacción con el usuario. Este sistema sustituye a los callbacks usuales en el desarrollo de aplicaciones de escritorio.

4.1.2. JSON

JSON es un estándar de formato de descripción de objetos para el intercambio de datos. se ha empleado este estándar en los módulos de persistencia, para guardar los datos en ficheros de manera que luego puedan ser leídos por el software para reconstruir los datos de una sesión anterior. *Game*

Maker Studio 2 también utiliza este formato para mantener, en ficheros separados, la persistencia de los proyectos desarrollados con dicho motor, por lo que el módulo de esta aplicación que se encarga de exportar datos del diseño, debe hacer uso del estándar JSON.

Adicionalmente, debido a cómo se generan los datos, el formato JSON debidamente escrito en ficheros es amigable tanto para los desarrolladores, que pueden leer ficheros escritos en dicho formato, como con tecnologías como git para el control de versiones.

4.1.3. git

Git es una herramienta open source de control de versiones diseñado para manejar todo tipo de proyectos software de forma rápida y eficiente. Junto con git, existen múltiples plataformas de repositorios que pueden emplearse para guardar los avances del desarrollo de forma remota. Tales como *github* (la plataforma escogida para esta aplicación) o *bitbucket*. Durante el desarrollo de esta aplicación, se ha utilizado para el control de versiones de la misma, pero también se ha tenido en cuenta a la hora de construir un sistema de persistencia que fuera amigable con git, dado que la aplicación desarrollada es una herramienta para desarrollar otros productos, es deseable que el desarrollo de estos productos últimos pudiera también manejarse con una herramienta de control de versiones.

4.1.4. Game Maker Studio 2

Game Maker Studio 2 es un motor de videojuegos desarrollado por *YoYo Games* de propósito general y en el cual se pueden desarrollar videojuegos en dos dimensiones. En el desarrollo de esta aplicación, se ha tenido en cuenta como el eslabón siguiente a *Maze Designer* en el pipeline del desarollo de un videojuego. Es decir, que los productos creados utilizando la herramienta que se desarrolla en este Trabajo de Fin de Grado están preparados para formar parte de un proyecto de *Game Maker Studio 2* Los proyectos de *Game Maker Studio 2* son guardados en disco como un conjunto de directorios y ficheros, cuyo contenido sigue el formato *JSON* anteriormente descrito.

4.2. Módulos y Clases

5

Desarrollo

Para el desarrollo de esta aplicación se ha seguido un modelo incremental iterativo, dividido en 4 etapas con una etapa previa del análisis de requisitos. Cada una de ellas subdividida en una fase de diseño, una fase de codificación, una fase de pruebas y finalmente una fase de integración.

Las cuatro etapas han sido divididas atendiendo a funcionalidad y son:

1.– **Canvas** En esta etapa se cubren los requisitos relativos a la componente de la aplicación que el usuario utilizaría para la construcción de los espacios jugables, así como todos los submódulos requeridos.

2.– **Llaves, habitaciones y persistencia** En esta etapa se cubren los requisitos relativos a la definición de modelos de llaves, la selección de regiones en el canvas para la declaración de habitaciones, el submódulo del canvas para la instanciación de llaves y puertas y la presistencia de todo lo desarrollado hasta el momento.

3.– **Interfaz gráfica** En esta etapa se han integrado todas las componentes hasta ahora desarrolladas y se ha extendido su funcionalidad para la comunicación entre las componentes y la representación de la información de los elementos del diseño.

4.– **Condiciones y exploración del diseño** En esta etapa se cubren los requisitos relativos a la definición de las clases que gestionan las condiciones de apertura de las puertas así como la exploración del diseño para la comprobación de su correctitud.

5.1. Entornos de desarrollo

Para la creación de la aplicación se han empleado distintos entornos de desarrollo. Para la implementación de la aplicación en si, se ha utilizado *qtCreator*, que es un IDE preparado específicamente para el desarrollo de aplicaciones utilizando C++ y Qt. Se ha investigado el motor de videojuegos *Game Maker Studio 2* tanto a nivel de usuario como su funcionamiento interno en lo que a persistencia y gestión de los recursos de un proyecto se refiere. Para la elaboración de esta memoria, se ha utilizado *TeXstudio*, que se trata de un entorno que facilita la escritura de documentos LaTeX.

5.2. Desarrollo de los módulos

5.2.1. Canvas

En esta etapa se desarrolla gran parte del módulo del Canvas, que será el núcleo de los módulos de la aplicación destinados al diseño de los espacios jugables. Las clases más destacables en este módulo son Canvas y Grid.

La implementación de la clase Canvas atiende a las siguientes funcionalidades:

1.– detección e interpretación del input de un usuario para añadir o eliminar rectángulos del espacio jugable.

2.– uniones, intersecciones y substracciones de rectángulos con elementos del espacio jugable.

3.– almacenamiento de los espacios jugables descritos por el usuario.

4.– conversión de los elementos del espacio jugable a regiones, para la posterior conversión del diseño a un grafo.

La primera de estas funcionalidades se lleva a cabo detectando inputs del ratón sobre el Canvas. Un input válido se interpreta como la acción de pulsar una tecla del ratón, a la que llamamos comienzo, el movimiento del ratón y la acción de soltar dicha tecla del ratón, a la que llamaremos final. Las coordenadas del comienzo y del final determinan las coordenadas de las dos esquinas opuestas del rectángulo que describe el usuario con el input, siempre y cuando dicho rectángulo tenga área mayor que cero. Estas coordenadas, sin embargo, se deben rectificar para que el rectángulo se ajuste a la rejilla. La clase Grid representa dicha rejilla y, mediante métodos como *nearestPoint*, permite, dado un punto, determinar las coordenadas del punto de la rejilla. Haciendo uso de esto, la clase Canvas puede ajustar los rectángulos descritos por el usuario a la rejilla. La distinción de adición o eliminación de dicho rectángulo se llevaba a cabo teniendo en cuenta qué tecla del ratón fuera pulsada.

La segunda de estas funcionalidades va intrínsecamente ligada a la tercera. Como decisión de diseño respecto a los espacios jugables, deben ser un conjunto de polígonos, con ángulos de 90 o 270 grados, que puedan tener agujeros (es decir, cuyo grupo fundamental no sea necesariamente el trivial). Nótese que, debido a la construcción de los espacios jugables, son uniones finitas de rectángulos. Para almacenar un objeto así en memoria, se plantearon iterativamente distintas implementaciones atendiendo a los problemas que planteaban las iteraciones previas:

Lista de objetos de clase QRect en la que Canvas contendría una lista de rectángulos. Los problemas de esta implementación son:

- A la hora de almacenar, o bien la información podía resultar redundante (si un vértice de un rectángulo quedaba contenido en otro) o bien las operaciones a realizar para evitar esto cada vez que se añadía o eliminaba un rectángulo acababan siendo demasiado costosas computacionalmente.
- A la hora de renderizar el espacio jugable, los rectángulos podían solaparse unos con otros. Además que la renderización de tantos rectángulos por separado podía llegar a ser demasiado pesada y en definitiva, visualmente quedaba una representación que podía llegar a ser confusa para diseños con geometrías más complejas.
- atendiendo a la cuarta funcionalidad, la operación para extraer las regiones de esta lista de rectángulos independientes era excesivamente costosa.

Lista de objetos de clase QPolygon en la que Canvas contendría una lista de polígonos, construidos mediante adición (unión) o substracción de rectángulos. La principal ventaja de esta implementación era que las áreas

conexas del espacio jugable quedaban representadas por un único polígono. Los problemas de esta implementación son:

- A la hora de añadir un nuevo rectángulo al espacio jugable, había que realizar tantas uniones entre polígonos como el número de polígonos del espacio jugable intersecaran con el nuevo rectángulo. Lo cual resultaba computacionalmente costoso.
- debido a la implementación de la clase QPolygon, los polígonos con agujeros no eran representados de forma correcta, pues todos los polígonos representados por dicha clase, por construcción, necesariamente tienen el grupo fundamental trivial (es decir, no pueden tener agujeros o en otras palabras, son convexos en el plano). Esto representaría un problema más adelante con respecto a la funcionalidad 4.

Lista de objetos de clase Shape en la que Canvas contendría una lista de objetos de clase Shape, que heredara de la clase QPolygon, construidos mediante adición o substracción de rectángulos. Dicha clase modelaría también polígonos con agujeros, solucionando el principal problema de la implementación anterior. El problema con ésta es que la algoritmia era demasiado costosa y compleja.

Objeto de clase QPainterPath la implementación final que utilizaba una clase de QT que, con cierto esfuerzo, resolvía todos los problemas anteriores.

5.2.2. Llaves, habitaciones y persistencia

En esta etapa se desarrollaron los módulos de Llaves, de Habitaciones atendiendo a las necesidades de las funcionalidades de persistencia y se extiende el módulo del Canvas para que integre la instanciación de llaves, la creación de habitaciones y la colocación de puertas.

El módulo de llaves consta de una clase KeyRepository que se encarga de la gestión de las llaves (objetos de la clase Key) a nivel lógico y de la clase KeyListWidget que es una interfaz gráfica a través de la cual el usuario interactua con el KeyRepository creando nuevas llaves, eliminando llaves anteriormente creadas y modificando llaves ya existentes. De manera similar se contruye el módulo de Habitaciones, sin embargo, la creación y eliminación de éstas se hace a través de una interfaz que se construye de manera similar al Canvas. Para la edición del nombre de las habitaciones y de una propiedad que determinará si la habitación va a ser exportada o no, se ha desarrollado una interfaz gráfica a parte similar a la de las llaves.

El módulo del Canvas se extiende atendiendo a los dos nuevos módulos creados. Con una nueva clase, que depende del Canvas y del Grid, se permite la definición de habitaciones seleccionando una región del diseño. Con otra clase también que depende también del Canvas y del Grid, se permite la instanciación de los modelos de llaves (que se definen con el módulo de llaves) en el diseño, así como la colocación de un token de comienzo y las puertas, que ambos elementos se utilizarán más adelante en el módulo de exploración.

También durante esta etapa, en cada una de las clases creadas y modificadas se implementan módulos para la propia representación en JSON y la interpretación de un objeto JSON para extraer sus datos, logrando así métodos necesarios y suficientes para las funcionalidades de persistencia.

5.2.3. Interfaz gráfica

En esta etapa se han integrado todas las componentes las unas con las otras, dejando componentes vacías que serían sustituídas en la etapa siguiente.

En la interfaz gráfica, se siguieron las maquetas planteadas, dividiendo el área de trabajo en 3 zonas, con tamaños diferentes atendiendo a su función e importancia a la hora de trabajar en el diseño.

Se construyó el sistema de los cambios de contexto mediante el uso de pestañas como elemento gráfico. Los elementos renderizados en el área que se correspondería con el canvas deben gestionar su propia renderización, aunque esta sea dentro de otra componente.

Además de la creación de la propia interfaz gráfica del usuario, se han implementado las acciones generales de la aplicación como el guardar los datos de todos los elementos en un fichero, cargar un diseño desde fichero y la creación de un nuevo diseño sin contenido sobre el que poder trabajar.

Se han proporcionado interfaces de comunicación entre los distintos módulos y componentes gráficas a través del mecanismo de huecos y señales (*slots and signals*) de Qt.

También en esta etapa se ha desarrollado el módulo que se encarga de exportar el diseño al motor *Game Maker Studio 2* para continuar con el desarrollo del juego en esa herramienta.

Finalmente, se han realizado pruebas de consistencia entre las distintas componentes, que involucraban, por ejemplo, que cambios ocasionados en una componente afectaran a otra. Por ejemplo, al eliminar espacio de juego del canvas, si hubiera alguna llave en dicho espacio eliminado, esta tendría que ser eliminada también. Otro ejemplo es que al eliminar un modelo de llave, todas las llaves instanciadas con ese modelo deberían ser eliminadas a su vez.

5.2.4. Condiciones y exploración del diseño

Durante esta etapa se han desarrollado los módulos de condiciones y de exploración del diseño.

El módulo de condiciones tiene como objetivo leer una cadena de caracteres y, si sigue la sintaxis definida en el apéndice A, entonces se creará un objeto de tipo condición. Este objeto se utilizará más adelante en el módulo de exploración y debe ser capaz de comprobar si la condición que representa puede ser satisfecha dado un inventario, imponiendo una lista de objetos consumidos al ser satisfecha. Nótese que el conjunto de conectivos lógicos admitidos por el lenguaje (*or* y *and*) no es un conjunto funcionalmente completo, esto es, utilizando solamente esos dos conectivos, no se pueden replicar otros como un conectivo *not*. Es decir, en la sintaxis no es posible poner como condición que no se tenga un objeto para poder ser satisfecha. En otras palabras, durante toda una exploración, el número de condiciones satisfechas es no decreciente.

En cuanto al módulo de exploración, se plantearon los siguientes problemas a resolver:

1.– construcción del grafo dirigido que representa al diseño, en el que las puertas son aristas y las regiones conexas del diseño son nodos.

2.– definir las acciones que se deben poder tomar para explorar dicho grafo.

3.– definir las condiciones de finalización de una exploración.

4.– definir una heurística para poder explorar el grafo de manera inteligente y reducir así el tiempo de exploración en los mejores casos.

5.– en caso de que el diseño no fuera correcto, devolver de alguna forma la cadena de acciones que el jugador podría tomar para acabar atascado.

El primero de estos problemas se resuelve obteniendo un objeto poligonal que es el diseño habiendo sustraido las puertas. Desde este punto podemos extraer las regiones conexas que quedan, que serán los nodos. Y para determinar las aristas del grafo dirigido, se han de emplear de nuevo las locaclizaciones de las puertas: para cada una de ellas, se toma un punto *A* que se encuentre inmediatamente en la dirección y sentido del vector normal de la puerta y otro punto *B* que se encuentre en la misma dirección pero de sentido opuesto. El nodo creado a partir de la región que contenga al punto *A* quedará unido por una arista al nodo creado a partir de la región que contenga al punto *B* y viceversa (recordemos que las aristas son direccionales). Dichas aristas tienen las condiciones definidas en la puerta para ser atravesadas en una exploración. Nótese que de esta forma se crean siempre dos aristas entre nodos adyacentes pero no tienen por qué compartir condiciones para ser atravesadas. Además, es posible que un nodo sea adyacente de sí mismo.

El segundo de estos problemas queda determinado por la lista de objetos consumidos para satisfacer una condición que imponen las aristas. La propia exploración, en lugar de simular un agente que simplemente se traslada de nodo a nodo, hace simplificaciones del grafo a explorar cuando las condiciones de las puertas a atravesar son satisfechas: si las condiciones para viajar de un nodo a otro han sido ambas satisfacidas (se ha recolectado un nuevo power-up o dicha puerta ya ha sido atravesada) entonces ambos nodos se unen en uno solo.

La tercera condición se deduce de esto último: un diseño no es correcto si llegado un punto, no se pueden tomar acciones que no hubieran sido intentadas ya y el grafo que se ha ido reduciendo consta de más de un nodo. Por contra, un diseño es correcto si cualquier exploración del mismo resulta en que el grafo se simplifica hasta ser un único nodo.

Para el cuarto problema, buscamos una heurística que reduzca el tiempo del mejor caso, en el que el diseño no es correcto. Ya que el peor caso es aquel en el que sí es correcto y por tanto se realizan todas las exploraciones posibles. Para esta heurística, entonces, buscamos una según la cual el explorador sea un derrochador: que malgaste el mayor número de llaves de manera que se desbloqueen el menor número de nodos nuevos y que no priorice la recolección de objetos, en especial la de aquellos que son de tipo power-up

Finalmente, el último problema se reduce a deshacer los pasos tomados por una exploración con la que se concluye que el diseño es incorrecto, listando los objetos recolectados y las puertas abiertas,

en el orden en el que se hicieron dichas acciones.

6

Integración, pruebas y resultados

Para comprobar el correcto funcionamiento de los módulos se han llevado a cabo una serie de pruebas, tanto a nivel individual como a nivel colectivo, con el objetivo adicional de asegurar el cumplimiento de los requisitos establecidos para el producto software.

6.1. Pruebas unitarias

Con cada módulo, se han realizado pruebas unitarias que aseguran la correcta ejecución del código, no solo a nivel íntegro de la aplicación, en tanto a que no haya errores técnicos que cierren el programa, sino también a nivel de usuario, en tanto a que cada módulo hace las operaciones que se esperan de él y cómo se esperan de él.

Para el módulo del Canvas, se han realizado pruebas de añadir y eliminar espacios tanto del interior como de los bordes del diseño, a distintas ampliaciones del Grid y habiéndolo trasladado tanto hacia coordenadas negativas como a positivas. También se ha intentado eliminar espacios donde no había antes, se ha intentado añadir y eliminar espacios con área cero (puntos y líneas) y se ha comprobado que el funcionamiento de la creación y eliminación de espacios es el esperado. Con respecto a las adiciones siguientes a este módulo, se ha comprobado que no se pueden definir habitaciones con intersección no vacía y que se eliminan como cabe esperar. la creación de las mismas funciona también correctamente tal y como sucede con el Canvas. A la hora de instanciar llaves, puertas y el token de comienzo, se ha comprobado que solamente se pueden instanciar en el interior de un espacio jugable, que las instancias creadas pueden ser efectivamente eliminadas y que no se pueden colocar dos instancias en la misma posición.

En el módulo de las llaves, se ha comprobado que se pueden crear y eliminar modelos de llaves correctamente y que estos cambios en la interfaz se ven reflejados en los modelos que hay en memoria. Se ha probado a eliminar una llave sin seleccionar ninguna, a eliminar varias a la vez, a crear llaves con el mismo nombre y a realizar cambios sobre ellas.

En el módulo de condiciones se ha comprobado que solamente se admiten condiciones que cumplen con la sintaxis definida en el apéndice A y que estas, sean simples o compuestas, pueden ser

validadas, y pueden comprobar si son satisfacibles dado un inventario, así como devuelven correctamente tantas lista con los objetos a consumir en caso de poder ser satisfacibles como posibilidades hay para ser satisfechas.

En el módulo de persistencia se ha comprobado que los datos guardados y esos mismos datos posteriormente cargados de fichero coinciden.

En la exportación se ha comprobado que se generan ficheros .yy de acuerdo a la estructura de proyecto de *Game Maker Studio 2*, creándolos cuando el proyecto en cuestión no los tiene y editándolos en caso contrario, sin sobreescribir ningún dato innecesariamente del proyecto.

En el módulo de exploración se ha comprobado con diseños incorrectos que el programa puede detectar los fallos y devuelve la secuencia de acciones a tomar para replicar el problema. También se ha comprobado que el programa detecta cuando un diseño es correcto.

6.2. Pruebas de integración

La integración de los módulos se ha realizado de forma iterativa en el desarrollo del proyecto. Con cada integración se han realizado pruebas para asegurar que el sistema funciona en conjunto. La mayoría de estas pruebas han consistido en comprobar la correcta interacción entre distintos componentes de la aplicación.

Se ha comprobado que seleccionando elementos del Canvas, se cargan los detalles de los mismos en la consola de la aplicación. Que al cambiar al contexto de la definición de las habitaciones, se muestra la lista de habitaciones con sus datos en lugar de la lista de los modelos de llaves y que en la pestaña de edición de las condiciones de las puertas efectivamente se pueden definir las condiciones y son aplicadas a la puerta seleccionada. También se ha comprobado que la eliminación de un modelo de llave supone la eliminación de todas sus instancias en el Canvas y que, en una condición que requiere un modelo de llave inexistente, se reporta un error y la condición se convierte en insatisfacible.

6.3. Resultados

El resultado tras el periodo de desarrollo es un producto software que cumple con los requisitos establecidos y que puede ser utilizado en el desarrollo de videojuegos.

7

Conclusiones y trabajo futuro

Una vez finalizadas las etapas del desarrollo, se ha construido un producto software funcional que cumple con los requisitos. Sin embargo, el ciclo de vida del desarrollo no concluye. En esta sección se habla del producto logrado, la utilidad que puede tener y apartados del mismo que quedan para ser ampliados, añadidos o mejorados en el futuro.

7.1. Conclusiones

Maze Designer es un producto software que cumple con los requisitos establecidos para ser utilizada como una herramienta de diseño de niveles en juegos de tipo mazmorra tal y como se ha explicado a lo largo de esta memoria. Cubre un área de trabajo bastante específica en la industria del videojuego pero sin embargo, el uso de esta herramienta facilita mucho tareas que no son triviales en el diseño de los niveles. A la vez, estas facilidades pueden dar lugar a un nuevo estilo de diseño de los niveles de un juego de tipo mazmorra, permitiendo que se exploren las posibilidades de un diseño orientado a una exploración ramificada. Además, al estar preparada para exportar a *Game Maker Studio 2*, puede ser incorporada en el pipeline del desarrollo sin mayor esfuerzo por parte del equipo de desarrollo. Aunque esto no significa que los diseños elaborados con esta herramienta sean específicos para *Game Maker Studio 2*. Así mismo, *Maze Designer* es una herramienta que puede ser utilizada para experimentar con patrones de diseño de forma poco costosa. Acelerando el aprendizaje de nuevos diseñadores de niveles.

7.2. Trabajo futuro

Con la finalización del desarrollo, se plantean las siguientes tareas que podrían realizarse en el futuro para mejorar o ampliar las funcionalidades de *Maze Designer*:

1.– **Exportar a otros motores:** se ha desarrollado la aplicación con *Game Maker Studio 2* como motor de videojuegos al que exportar, sin embargo, debido a que los diseños son genéricos, se podría explorar la posibilidad de exportarlos a otros motores, abarcando un mayor número de usuarios.

2.– **Incluir un sistema de notas:** a la hora de desarrollar esta *Maze Designer*, no se tuvo en cuenta una funcionalidad que podría ser útil, como es la de poder apuntar notas en cualquier parte del diseño.

3.– **Mantenimiento:** como todo proyecto software, es inevitable la presencia de bugs en un código. Se delega al trabajo futuro la corrección de estos errores según sean reportados.

4.– **Realización de pruebas por parte de usuarios:** en el desarrollo de este TFG no se han podido realizar pruebas de la aplicación por parte de usuarios reales más allá del desarrollador. Convendría realizar una serie de pruebas para el control de la calidad del producto, de manera que se obtuviera retroalientación para poder seguir mejorando la aplicación.

5.– **Mejorar el rendimiento de la aplicación:** debido al tiempo de desarrollo, no se han podido optimizar todos los módulos al máximo, utilizando el potencial que ofrece C++. Una optimización de la herramienta podría significar una mejor experiencia del usuario y, en particular, en las tareas de exploración, podría reducirse el tiempo de cómputo.

Definiciones

backtracking En una exploración, la acción de regresar a localizaciones anteriormente exploradas.

editor de niveles Herramienta utilizada por los motores de videojuegos para colocar las instancias que constituyen un escenario del juego.

espacio de juego ver espacios jugables.

espacio jugable regiones o terreno virtuales de un juego por el cual el jugador podrá mover a su avatar.

exploración lineal Tipo de exploración caracterizada por carecer de múltiples caminos por los que progresar.

exploración ramificada Tipo de exploración caracterizada por presentar múltiples caminos por los que progresar.

inventario conjunto de instancias lógicas, como llaves o power-ups, recolectadas durante una exploración.

objeto consumible Categoría de objetos del videojuego que tras ser utilizados desaparecen.

pipeline Conjunto de fases que componen el desarrollo de un producto software. Dichas fases se disponen de manera secuencial, estableciendo una cadena de dependencias.

power-up Categoría de objetos del videojuego que tras ser recolectados el jugador puede utilizarlos tantas veces como quiera.

Apéndices

A

Definición de la sintaxis de condiciones

Código A.1: a definición de la sintaxis seguida para las condiciones, escrita en el lenguaje prolog. Nótese que la cláusula *item ->[key]* debería admitir cualquier cadena de caracteres alfanuméricos que no contenga espacios, pero se deja así por simplificar el código debido a que esta especificación es puramente formal y no se emplea en el desarrollo como tal.

```
1  startCondition --> condition.
2  startCondition --> [].
3  condition --> simpleCondition.
4  condition --> compositeCondition.
5  simpleCondition --> have, number, item.
6  have --> [have].
7  have --> [].
8  number --> numberConstant.
9  number --> [].
10 numberConstant --> nonZeroDigit.
11 numberConstant --> nonZeroDigit, digits.
12 nonZeroDigit --> [1].
13 nonZeroDigit --> [2].
14 nonZeroDigit --> [3].
15 nonZeroDigit --> [4].
16 nonZeroDigit --> [5].
17 nonZeroDigit --> [6].
18 nonZeroDigit --> [7].
19 nonZeroDigit --> [8].
20 nonZeroDigit --> [9].
21 digits --> digit.
22 digits --> digit, digits.
23 digit --> nonZeroDigit.
24 digit --> [0].
25 item --> [key].
26 compositeCondition --> conditionList.
27 compositeCondition --> conditionList, connector, compositeCondition.
28 compositeCondition --> ["("], conditionList, [")"], connector, compositeCondition.
29 compositeCondition --> ["("], compositeCondition, [")"].
30 conditionList --> simpleCondition.
31 conditionList --> simpleCondition, connector, conditionList.
32 connector --> [or].
33 connector --> [and].
```


UAM
UNIVERSIDAD AUTONOMA
DE MADRID