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

mayo 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

Introducciónintroduccion/introduccion

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 (1994)*, *Cave Story (2004)*, *Ori and the Blind Forest (2015)* y *Hollow Knight (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 detalles de la implementación.

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. 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.

2.2. 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.

2.3. 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.– **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 3D como en 2D.

2.– **Unreal Engine** Es un motor de videojuegos creado por Epic Games. Es también un motor de propósito general para creación de juegos tanto en 3D como en 2D.

3.– **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 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.

2.4. 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 Análisis de Requisitos

3.1. Requisitos funcionales

3.1.1. 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.

3.1.2. 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.

3.1.3. 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.

3.1.4. 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.

3.1.5. 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 siguiente sintaxis:

```
startCondition --> condition.
startCondition --> [].

condition --> simpleCondition.
condition --> compositeCondition.

simpleCondition --> have, number, item.

have --> [have].
have --> [].

number --> numberConstant.
number --> [].

numberConstant --> nonZeroDigit.
numberConstant --> nonZeroDigit, digit.

nonZeroDigit --> [1].
nonZeroDigit --> [2].
nonZeroDigit --> [3].
nonZeroDigit --> [4].
nonZeroDigit --> [5].
nonZeroDigit --> [6].
nonZeroDigit --> [7].
nonZeroDigit --> [8].
nonZeroDigit --> [9].

digit --> nonZeroDigit.
digit --> [0].

item --> [key].

compositeCondition --> conditionList.
compositeCondition --> conditionList, connector, compositeCondition.
compositeCondition --> ["("], conditionList, [")"], connector, compositeCondition.
compositeCondition --> ["("], compositeCondition, [")"].

conditionList --> simpleCondition.
```

```
conditionList --> simpleCondition, connector, conditionList.

connector --> [or].
connector --> [and].
```

donde la cláusula item puede ser cualquier cadena de caracteres del espacio de nombres de llaves.

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.

3.1.6. Requisitos de comprobación de la completitud

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.

3.1.7. 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.

3.2. Requisitos no funcionales

4

Diseño

4.1. MazeDesigner GUI

5 Desarrollo

5.1. Canvas

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 el método *nearestPoint*, permite, dado un punto, determinar las coordenadas del punto de la rejilla más próximo a este. 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. Atendiendo a la definición de un espacio jugable, éste debe ser un conjunto de polígonos 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 (de hecho, pertenecen a una sigma-álgebra) Para almacenar un objeto así en memoria, se plantearon iterativamente distintas implementaciones atendiendo a los problemas emergentes:

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 rec-

tángulo 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

- 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 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. 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. Llaves

6

Integración, pruebas y resultados

7

Conclusiones y trabajo futuro

7.1. Conclusiones

7.2. Trabajo futuro

Bibliografía

[1] K.S.Narendra and K.Parthsarathy, *Identification and control of dynamical system using neural networks*, IEENN, 1 (1990), pp. 4–27. Download.

[2] R. W. Zurek and L. J. Martin, *Interannual variability OF planet-encircling dust activity on Mars*, J. Geophys. Res., 98 (1993), pp. 3247–3259. (Descargar).

Definiciones

acrónimo Sigla cuya configuración permite su pronunciación como una palabra; por ejemplo, ovni: objeto volador no identificado; TIC, tecnologías de la información y la comunicación.

definición Proposición que expone con claridad y exactitud los caracteres genéricos y diferenciales de algo material o inmaterial.

espacio jugable regiones de un nivel por las que el jugador podrá mover a su avatar.

Acrónimos

IEEE Institute of Electrical and Electronics Engineers.

Índice terminológico


UAM
UNIVERSIDAD AUTONOMA
DE MADRID