Generation of Random mazes / Maze like Levels and Creation of a Game Using them

Project Final Submission

De Montfort University

Computer Games Programming

Jake Coffey – P15187559

Contents

[Abstract 1](#_Toc511058813)

[Introduction 2](#_Toc511058814)

[Analysis of Requirments 3](#_Toc511058815)

[Level generation 3](#_Toc511058816)

[Core Game Components 4](#_Toc511058817)

[Enemies 4](#_Toc511058818)

[Player Character 5](#_Toc511058819)

[UI 5](#_Toc511058820)

[Analysis Conclusion 5](#_Toc511058821)

[Bibliography 6](#_Toc511058822)

# Abstract

Random generation of various content in games has become commonplace in a subset of games that use some level and style of random generation to design the layout of the game level and what content from a pool is used in each level. The level of random generation in each game varies, with some using pseudo random generation that takes pre-built level portions and content spawn points and combines them in a random way, and others using true random generation with some limiters applied to prevent problems such as content being created within each other. Generation of levels can be a complex matter, with various and varied methods exiting that provide a wide range of different outputs and levels of complexity. The majority of these can and are applied in creating maze or maze-like levels, with random generated dungeons in games typically resembling mazes. The purpose of this project is to take a method a random generation and apply it to creating maze like levels to be used in a game. The game style used in combination with the generated levels is a “rogue-like” game, where the level and content (enemies, items, traps) are randomly generated and placed through out the levels to allow the player to progress. This project shows various generated levels that, while not resembling the typical perfect maze (hedge maze style), resemble a non-typical clear maze-like structure, as well as a rogue like game that utilises these levels appropriately. The process and reasons for the generation and game creation choices will be discussed in this paper.

# Introduction

The program created consists of a fully-fledged game that fully utilises random generation to create maze like levels, not using any pre-set combinations of level designs to achieve “random levels” and instead using an algorithm that randomly iterates through “cells” of the level and chooses which cells should be open walls or open, with some limits set in place, discussed in the generation segment of the report, to ensure a navigable and maze like structure is generated. The algorithm used is similar to a style of generation know as recursive backtracking, that chooses a random direction each iteration and creates a path or wall in that direction if it can, continuing until it can no longer do this and then going back until it reaches a cell where it can continue this process. The differences between this algorithm and the method implemented will be discussed in the generation segment of the report. Creating levels that both resemble a maze-like structure and are suitable for a game level is a challenge that required good programming and debugging skills as well as creativity in the approach used in the implementation. The game implementation also uses random generation to some extent to populate the generated levels with content from the pool of enemies, items and traps created as part of the project in order to give the structure of a “rogue like” game that allows the player to progress and improve in each run through the game.

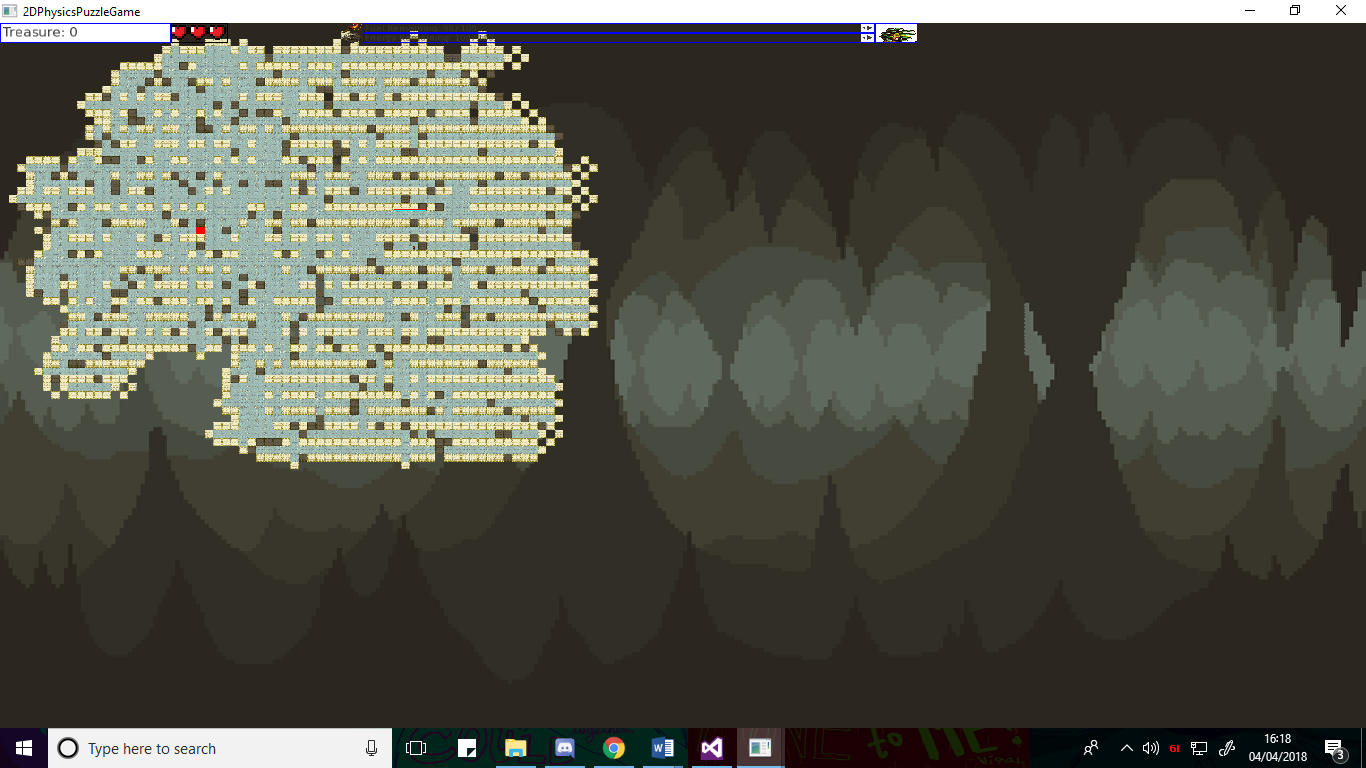


Fig.1 – Left shows the normal view of the game. Right shows a zoomed out shot of a generated level.

Various libraries are used to create the appropriate implementation. The super-fast media library ([4] SFML) is utilised for managing parts of the project that improve the overall quality, including sound, window management, textures and sprites, the game loop, and keyboard input. The game is rendered and designed in 2D to properly utilise the functionality provided by SFML. To properly implement a rogue like game that utilises the generated levels, a GUI is required for a variety of purposes. The GUI will be required for menu systems for a main menu, upgrades and progression, and representing information important to the player during actual gameplay. To implement a professional looking and functioning GUI and menu system, an extension of the SFML library called [6]TGUI was used, that focuses on and provides methods for GUI functionality. [1] Box2D was utilised to provide physics and collision management for the project, including special interactions between different entities in the game and filtering of collisions that were unwanted.

While various rogue like games and level generation algorithms and methods exist, the two in combination are typically not done with true randomness, instead using the randomness algorithms to piece together procreated level portions and content in a random manner. As such, a portion of the reasoning behind the project was to see how effective the level generation would be for a rogue-like game when not using pre-set portions of levels and instead generating the level randomly through a set of iterations, limitations and an algorithm, and then attempting to create a fully functioning game.

Portions of the project are similar or utilise functionality that I have experimented with previously, where as other sections of the game created are completely unlike anything I have undertaken before. As such, this, in combination with the previously mentioned need for creativity in the implementation as well as good programming skills, knowledge, debugging skills and research skills, forms further reasoning for undertaking this project, as it expands upon existing knowledge, as well as providing experience in new areas, allowing for a development of further knowledge and experience in programming various parts of games, and algorithms. Likewise, the amount of content need for a good project as well as for a good “rogue-like” game allows for creativity and experimentation with programming that also leads to further expansion of personal experience and knowledge. For example, I have created games before, however I have never developed items in a game that can be acquired and used to change how the game plays, and this project provided the opportunity to experiment and create a variety of these, developing my skills in game creation.

Further reasoning behind the project is the previously mentioned libraries being used. Undertaking this project allows for further development of skills utilising external libraries, something that is typically required when creating ambitious and complex games, as well as developing skills in the specific areas each of the libraries covered, these being, physics and collision form Box2D, game management, sound and sprite creation from SFML, and GUI creation and management form TGUI.

# Analysis of Requirments

The project can be split down into a group of two different major components that themselves can then be split down into smaller, important for them, components. The two major components of the project were the maze / level generation and mechanics related to that, and the core rogue-like game component.

## Level generation

Level generation was one of the key points of the project proposal, the proposal being to create a game in levels that were randomly generated each time and resembled a maze-like structure upon creation. As such, it was obvious that one of the major components that would need to be focused on was the level generation. Due to the required randomness of the levels, this component of the project would require an algorithm of some kind to be implemented to create the levels on the projects being run, rather than the typical pre-set level design coded into either the project source code or loaded from an external file that the majority of games utilise. Fortunately, research into this area had already been undertaken as part of the first hand in literature review, which specifically researched and discussed various algorithms or methods of level generation. The requirement to keep the levels either as mazes or resembling maze like structure ruled out some of the researched generation methods and algorithms, such as cellular automata, due to them producing results that resembled other structures, such as cave systems, more than they did a maze-like structure. This culling of certain algorithms led to the conclusion that the algorithm utilised would need to be an algorithm that has limits placed on how it created the level, for example making sure the cells are connected before creating a piece of the level there and directions limits, to ensure a coherent maze-like structure where each segment was accessible and connected.

The level generation would also require some method of taking the result of the algorithm and actually making the level from it. This was required, since the algorithms typically only created a template to be used such as an output of different numbers for each type of cell there could be, and the algorithm itself didn’t have any way to create a visual representation of the output. The visual and physical representation of the maze would need to be created from the algorithm output by utilising a method to create an object in the project. In the case of this project, the visual would need to be created from SFML and the physical representation from Box2D, as these were the libraries Utilised to create visuals and any physics and collision.

## Core Game Components

To produce the game element of the project, various components were required, with some being general game requirements, and others being requirements needed specifically because of the rogue-like genre aimed for. The majority of the below are also listed as goals in the project proposal and contract, and as such were required to properly fulfil the contract and complete the project.

### Enemies

A key component of the project was enemies. Enemies are an engaging element in a majority of games that allows the player to interact with something other than the environment and any environmental objects such as puzzles. Enemies were viewed as a core component for the game element of the project because the project intended was not an exploration style game and needed to be more engaging and complex than simply finding the exit of the maze and proceeding to the next level. Enemies provided both a hinderance to the player that they would have to overcome, providing an extra element of challenge, as well as another element for the player to engage with, since having a number of different enemies would require the player to consider what they were doing and what they were facing before taking action, rather than blindly exploring. Aside from my reasoning, rogue-like games in general require enemies to properly function, as one of the key elements of a rogue-like game is a dungeon style game and progression, both of which would need enemies, either to populate the dungeon or to provide some method of progressing and something to progress against [7].

The requirements for enemies are shared between the different types of enemies. Each type of enemy would require a visual and physics element, from SFML and Box2D respectively, so that they can be seen and interact/ collide with other elements of the game. All the enemies would require a method of attacking as well, since they need to be able to hinder the player in some way, however this method of attack can vary. Movement would also be required, as the enemy being completely stationary would be both boring and not much of a challenge.

### Player Character

As the game section of the project involved moving around the maze, another requirement was an avatar or character of some kind for the player to control and use to interact with objects and entities in the maze. This was a requirement, since without this, the game would literally be just moving a camera around to look at a maze, with no method, or at least limited methods of interacting with anything else in the game. While there are some types of games that do not require a character for the player to control, these are typically genres such as puzzle games or strategy games, where the player doesn’t require a single character to interact with, something the project didn’t fit into.

A player character, like the enemies, required a visual and physics element to be displayed and interact and collide. Likewise, the player would also need a method of attack to combat the enemies in some way and provide a way to defeat them and thus progress. A method of moving the player would be required to move them around the maze and progress through it. However, unlike the enemies, the player character required other elements to interact with the various items in the game, as well as objects such as a camera, so that the game view would follow the player rather than being stationary, and a torch to fit with one of the mechanics of the game to provide illumination that slowly got darker the longer the player spent in the maze levels.

### UI

Due to the variety of items intended to be included in the game and because the game would have a variety of data that would need to be presented to the player in some manner, a UI element was required to visually represent this information for the player, preferably a graphic interface of some kind for the more professional and eye pleasing look compared to a purely text based interface. The UI was required to display a variety of data important to the player, from their health, to the amount of treasure they had and fuel for the torch mechanic. To properly display this information, a variety of different types of interface were required form the UI. For example, displaying the treasure required the amount of treasure acquired as a number, and so a text box with background was required. Whereas something like the health or the fuel could be displayed purely visually by a bar and image to show what the bar was for or a set of hearts to represent the current amount of health. While all of this could be done with textboxes, it was “required” to have these different UI elements to present a visually pleasing and more professional appearance.

The UI was also required for the “progression” portion of the rouge-like game, as some method, such as buttons, was required to both show and allow the player to choose various upgrades to improve the character and allow them to progress further the next time they played.

### Items

Items are a common feature in rogue-like and rogue-lite games, commonly changing how the game works in some way or providing the player with more methods to interact with their surroundings. These items are sometime bought as parts of upgrades (as mentioned above in the UI), whereas other times they are found during actual gameplay and only last until that bout of gameplay is concluded. Either way, they feature in the vast majority of chosen genre of game, and as such are practically a requirement for the project. They are also required simply because they were listed as goal for the project as part of the contract and the goals set forth in the weekly plan.

The requirements for items are similar to all other physical components of the game, needing a visual and a physics element to be properly displayed and interacted with in the game. Items would also require a way to interact with the player, since they typically manipulate the way the player can act and interact with other elements of the game, as well as typically being activated by the player. Each item would also require a different effect that they create or do so that they have some reason for existing and being used by the player.

### GameStates

Because of the need to have an upgrade menu, as mentioned in the UI analysis above, we would also require some way to control what state the game is currently in, so that it knows when to the Upgrade menu UI code, and when to run the normal gameplay code alongside the in-game UI. Likewise, we would also need to have a method to know what state the game is in to facilitate having multiple levels that the player can progress through, since we would need to know when to delete the level and then create the next one, which while these are being done, the normal in-game code can’t be run and the other state can’t be done. This is because while we are deleting a level, we can’t run the level and load a new one at the same time, since the deleted level can’t be run when pieces of it are being deleted and loading a new level while deleting one would cause a mass of conflicts.

To solve this, we need “Game States” that tell us what state the game is in so that we can do the code and processes in the correct order without them conflicting with each other.

### Analysis Conclusion

The above-mentioned components were required to create the basis for the game, without them, the game wouldn’t have been functional, at least not to a decent and proper capacity. Further elements could be and were included to improve the game, but the above were required.

# Design Considerations

Some elements of the project and game design went through changes and additions to their original design concept or intended method of design. These changes came from a change of direction in the project development due to either additional information, a change of mind, or possible errors and problems that prevented the original design. These Are discussed in this section.

## UI Considerations

The original design of the UI intended to be used [see appendix 1] contained far fewer visual elements than the final design and was mostly designed to display the information as text within a border or shape of some kind. For example, the health bar was originally intended to just be a text box that read out the number of the player’s health. This was redesigned to contain much more of a visual element when compared to the original proposed design, changing the health textbox to be intended as a set of sprites (cartoon hearts) that show how much health is remaining visually, as well as changing the current item to have a sprite of the item behind it for a visual representation [see appendix 2]. This design choice was done mostly because of the introduction of the TGUI GUI library, which allowed for a much more visual UI to be considered and designed through the use of its dedicated GUI methods when compared to the original intention of using SFML text to display the information and sf rectangle shapes for the background and fuel bar. The second intended design also had some other non-visual changes, such as removing the information for the number of ropes and bombs held. This was done because how Items were implemented into the project no longer required a bomb or rope to be held and instead had them be reusable, meaning the number held value was pointless and therefor removed from the design. The mini-map in this design was also moved from the bottom of the screen to the top to fill in the free space that had been created from the changes and make the UI look neater. The final design implemented [see appendix 3] was mostly the same as the intended design with only images next to the fuel bars to represent what each bar was for visually. Other choices considered during the designs included having all of the information on one bar that was then split up into segments to have a solid single GUI that also wrapped around the Minimap. This was considered for the design, since the initial idea seemed like it would be far more professional looking and neater. This design wasn’t used eventually and the above discussed was used, due to the methods believed to allow this from TGUI, proved to not look or function as imagined, spoiling the design, as well as the design obscuring too much of the screen to be of practical use.

TALK ABOUT MAIN MENU MAYBE OR DO IT IN PERMA-DEATH

## TExture Loader

To construct a game with a professional look and sense of high-quality, a large number of textures and sprite sheets were required to provide all of the game elements with a good looking visual representation that wasn’t just a coloured shape of some kind. Unfortunately, this led to some problems, since the initial two methods of loading textures both had flaws. Initially the textures were loaded all in the main file of the project, and while this worked perfectly fine, it meant passing the textures to any of the created objects that needed them, leading to excessively long and untidy method calls and object creation. This became even more a problem when introducing a level manager, since the textures would fist have to be passed to the level manager and then distributed to the objects, leading to passing a significant amount of textures to the level manager and then re-passing them to the correct object methods. To fix this, loading the relevant textures in each object was considered so that passing them would not be required, however this would have lead to loading the texture multiple times when creating multiple instances of the objects, with the enemies for example, leading to a waste of memory and decrease in performance.

To solve the above, the decision to create a texture loader was made instead. This texture loader would load all of the textures in one file, much like with creating them in the main, but instead of having to pass them to objects, the texture loader was designed in a manner that allowed any object to access it so long as they were part of the same projects. This allowed for both neater and more efficient texture loading, since all of the textures could be loaded once upon the game starting and then provided to each class without the long untidy method calls. The textures could be neatly loaded in the texturelaoder, and so long as the textureloader.h file was included, any other file could access and utilise the loaded textures as if they had been loaded in their own file. The design of the texture loader was fairly simple, providing all classes with the textures by simply declaring the texture loader as an extern class, allowing it to be accessed from anywhere in the project.

# Implementation

This section will discuss how various sections of the project were implemented in regard to how they were coded and how the desired effects were achieved.

## Texture Loader

The intent and design decisions behind the texture loader have been explained in the design considerations segment above. Much as discussed above, the texture loader simply created and loaded all of the various textures for the game in one file upon starting the game so that they could be accessed from this file. This was done using the in built SFML texture objects and their load from file method to take a file path to the relevant image wanted as a texture and then load it and apply it to the texture object. These texture objects could then be used to apply the texture to a SFML rectangle to draw the texture instead of the rectangle shape or used as a sprite by creating a SFML sprite object from them. The loading method can be seen in appendix 4.

However, this wouldn’t allow other files to access the objects. To achieve this, we had to declare the textureloader.h as an external object,[see appendix 4], and create a single instance of it within the main.cpp file of the project. Doing this made it so that the methods and variables, i.e. the texture objects, could be accessed by any file in the project so long as the textureloader.h file was included and used to set the texture over the top of the objects rectangle to use the texture instead. The reasoning behind doing this was discussed above in the design choices segment. The reason that declaring the texturelaoder as an extern allowed for this is because doing this sets the project to refer to the same instance of the object throughout the entire project, and so once the single instance was created in main.cpp, all references to the texture loader in the project were set to check that instance, allowing for any file to access the loaded textures.

# Bibliography

[1] Catto, E. (n.d.). *Box2D | A 2D Physics Engine for Games*. [online] Box2d.org. Available at: http://box2d.org/.

[2] Sfml-dev.org. (n.d.). *Controlling the 2D camera with views (SFML / Learn / 2.4 Tutorials)*. [online] Available at: https://www.sfml-dev.org/tutorials/2.4/graphics-view.php.

[3] Sfml-dev.org. (n.d.). *Playing sounds and music (SFML / Learn / 2.4 Tutorials)*. [online] Available at: https://www.sfml-dev.org/tutorials/2.4/audio-sounds.php.

[4] Sfml-dev.org. (n.d.). *SFML*. [online] Available at: https://www.sfml-dev.org/.

[5] Tgui.eu. (n.d.). *TGUI: 0.8 Tutorials*. [online] Available at: https://tgui.eu/tutorials/0.8/.

[6] Tgui.eu. (n.d.). *TGUI: Texus' Graphical User Interface*. [online] Available at: https://tgui.eu/.

[7] King, A. (n.d.). *The Key Design Elements of Roguelikes*. [online] Game Development Envato Tuts+. Available at: https://gamedevelopment.tutsplus.com/articles/the-key-design-elements-of-roguelikes--cms-23510 [Accessed 9 Apr. 2018].

# Appendix

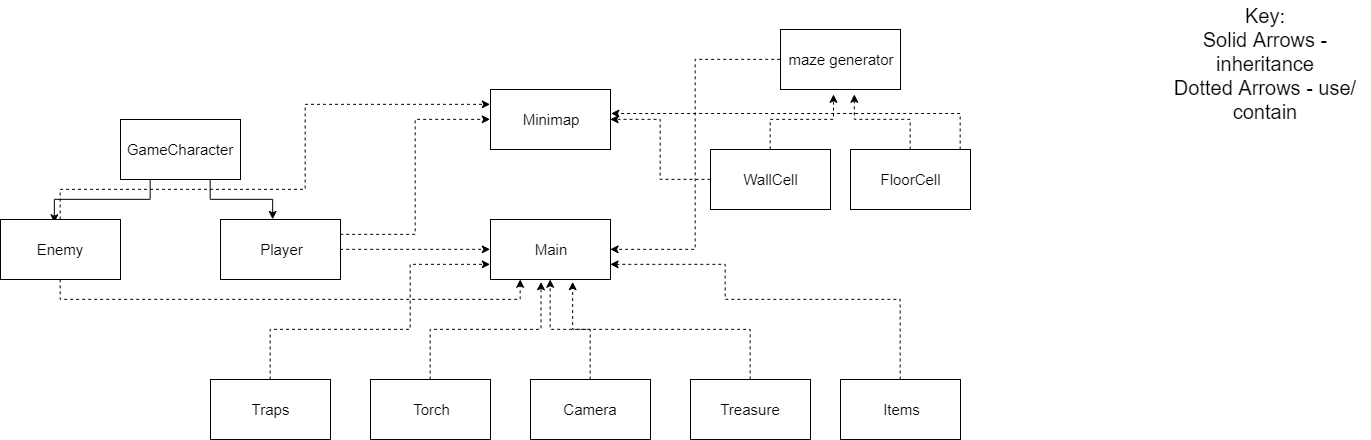
## [1] UI Design original

## [2] UI DESIGN EDITED

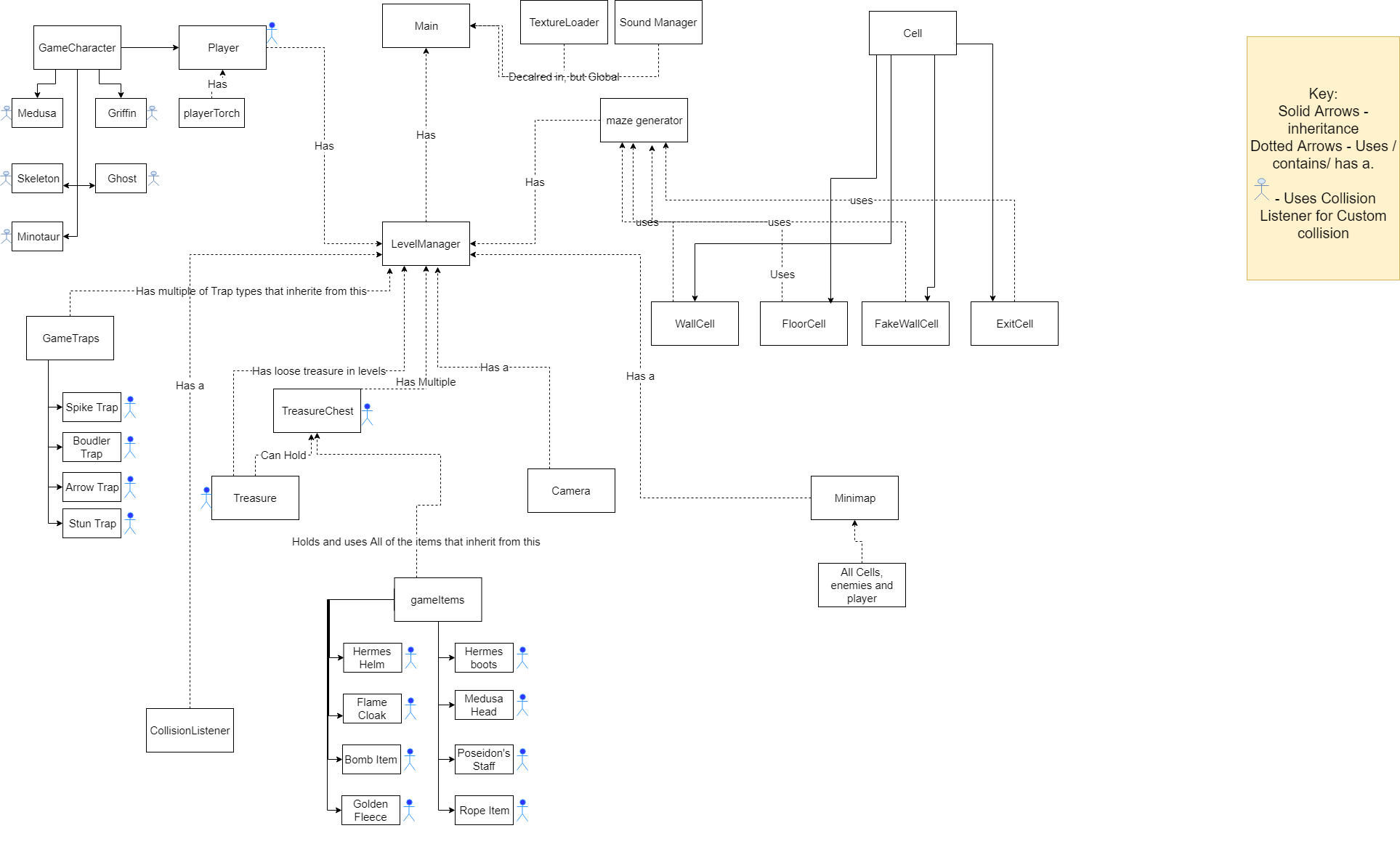
## [3] Actual UI Design

## [4] Left texturelaoder being made and extern and sf Texture objects being created. Right, examples of loading textures

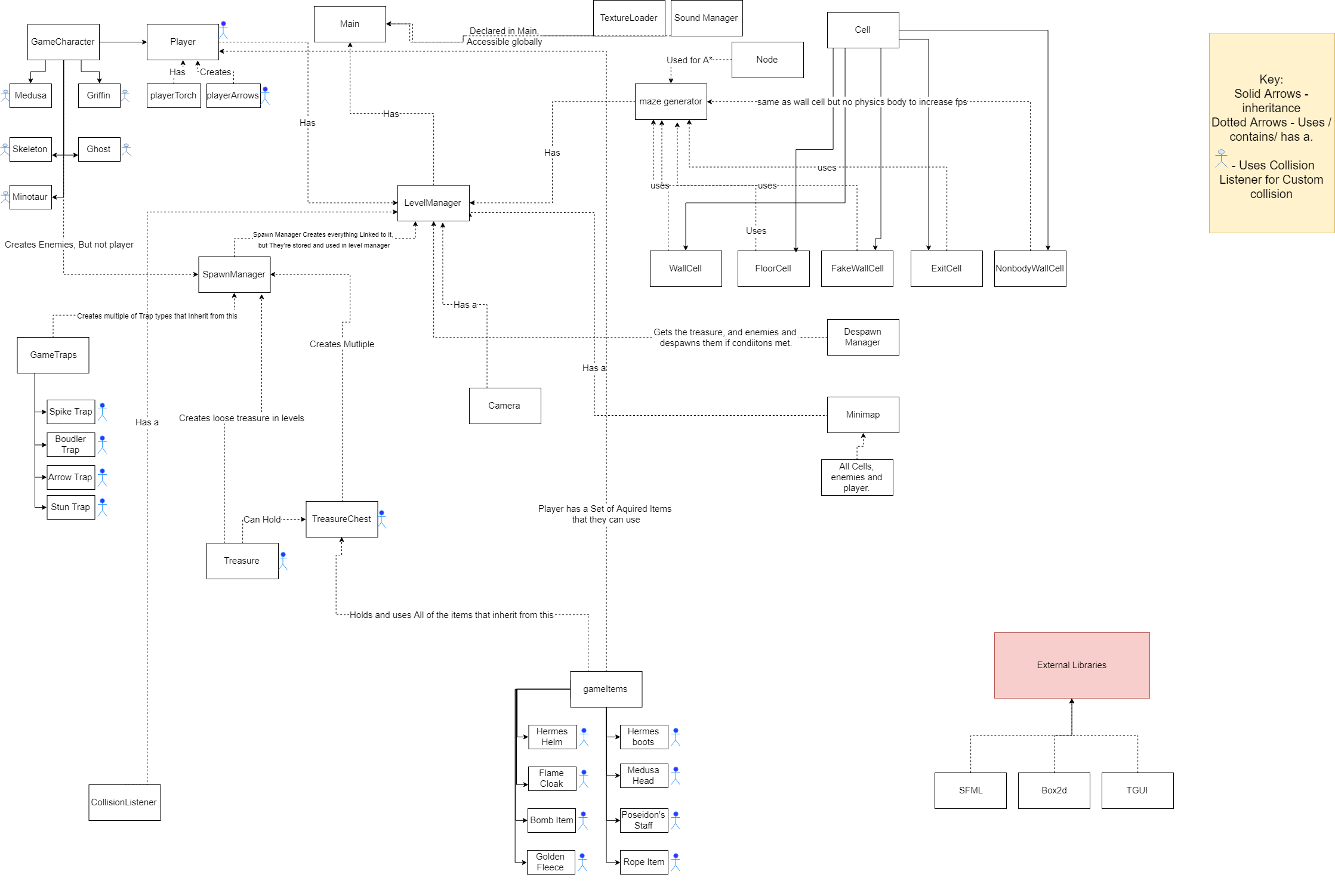
## [5] UML Original Idea Diagram

Initial idea of what I believed the basic classes would be

## [6] UML Diagram Updated

UML diagram created after coming up with the idea for the majority of the content and implementing some of it. General Relations ships (i.e. passing some of the data from one class to another) excluded for clarity fo diagram and readability. Functions and variables Excluded for the same reasoning.

## [7] UML Final

UML containing the inheritance and ownership/ usage between all of the classes. Again, doesn’t show the general relationships and functions/ variables to maintain clarity and readability.

## [8]Class Diagram Showing Functions and Variables – Also Availble To view in Visual Studio and As Image in project Directory

