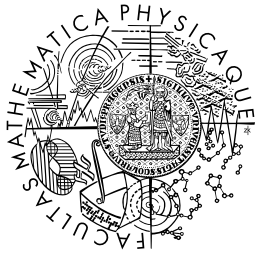

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Erdi Izgi

Framework for Roguelike Video Games Development

Department of Software and Computer Science Education

Supervisor of the master thesis:	Mgr. Jakub Gemrot, Ph.D.
Study programme:	Computer Science
Study branch:	Computer Graphics and Game Development

Prague 2018

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

Title: Framework for Roguelike Video Games Development

Author: Erdi Izgi

Department: Department of Software and Computer Science Education

Supervisor: Mgr. Jakub Gemrot, Ph.D., Department of Software and Computer Science Education

Abstract: While the video game development industry has had big success and increases in the amount of competition, using new tools which accelerate and improve the process is inevitable. Especially, domain-specific tools prevent game developers from performing unnecessary effort by providing reusable components. Partial automatization of the games that fall under the same game genre significantly decreases the development time. In this thesis, we propose an extensible framework architecture for roguelike video games development with a visual node-based user interface which is also artist and designer-friendly. The architecture aims to provide a faster game development process by wrapping common patterns in the roguelike games into simple node representations.

Keywords: game development framework video games roguelike

I would like to thank my supervisor Mgr. Jakub Gemrot, Ph.D. for valuable consultations and passionate discussions during the whole time I have been working on this thesis. I also would like to thank my family for supporting me from a thousand miles away.

Contents

1. Introduction

The process of a game creation is hard and challenging from the first sketched idea to release the game. It is also really hard to make it in the gaming industry due to problems which come after the release such as insufficient or wrong marketing. Despite the fact that these results are reversible, the developers may not afford it because of the prolonged development time.

Especially for the new video game corporations, keeping the game creation time as short as possible is quite a vital requirement to spend the resources for marketing after the release. That is why all the developers in the game field have to create reusable components or tools which automatize a decent amount part of the game development process.

Today, there are hundreds of game engines which are serving different platform requirements and designed with various approaches. These utilizing tools try to collect all the good practices in the Computer Graphics and Software Engineering areas to make the games more efficient and help developers not to spend time on developing the primitive structures such as the game loop or the event system.

However, the competition among the game development studios has never been higher than what we can observe in the present state of the market based on the report of McDonald [2017]. This phenomenon also means that the necessity for the game automation tools is significantly apparent. Even though the support that comes from the game engines are quite substantial, unfortunately, this is not adequate to meet the today's game development standards. Therefore, many game engine companies have opened their virtual shops to support developers with the plug-ins and extensions. These extra tools give a better chance to their user base for not creating the same components repeatedly.

One can imagine that the possibilities become endless when the developers can create assets for a specific goal and submit them to the asset stores. Game development studios can purchase an asset for an affordable cost, and save months of development in return. Apparently, this ecosystem is quite valuable and reputable for small or big corporations that aim to decrease the development time and enter the market rapidly with higher promotion chances.

The number of assets in the stores increases rapidly since the game development studios also became asset development studios. This inevitable result of using game engines also has created an alternative income source for many studios. Therefore, developers came to a point which they search for an asset first before developing it. There are plenty types of assets such as 3D models, complete game projects, editor extensions, particle systems, scripts, services and even shaders and graphical assets.

In the scope of this thesis, editor extensions will be the primary focus among all the asset categories, and a framework for automatizing the game development for roguelike game genre is created, and its architecture is deeply investigated. Indicators as mentioned earlier show that assets are worth to develop and think as a software product.

1.1 Motivation

Computer code is much like ordered building blocks. Most of the compilers transform the human-readable code into Parse Trees or Abstract Syntax Trees (AST). Therefore a code block can be represented in a tree structure as it is in the Figure 1.1. Most of the visual scripting tools wrap the multiple nodes in the AST and makes them one node. Their aim is achieving the most efficient node order to create the same code block with a visual interface. The same process is applicable top of the idea of visual scripting tree by wrapping the frequently used nodes since developers use the same or similar node orders to create a game in the same or similar genre.

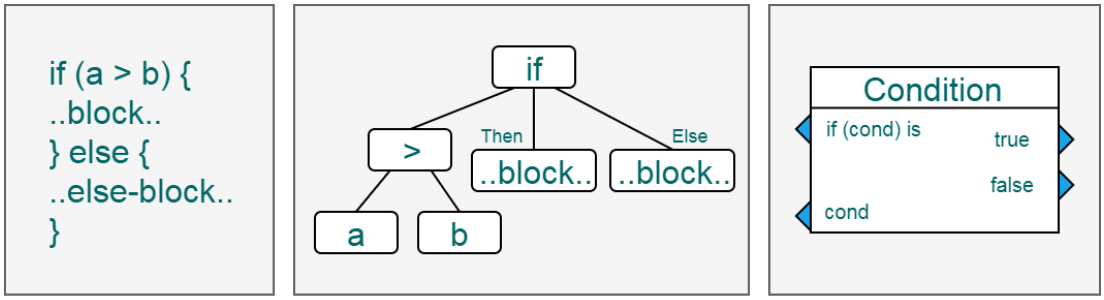

Figure 1.1: The transition among human-readable code (a), AST(b) and visual scripting node (c)

Roguelike luckily has been one of the oldest types, and millions of gamers have been playing it. Accordingly, it will not be wrong to say that this game genre is quite mature and the features of it easily extractable with its consistent gameplay patterns. Therefore it should be possible to create a framework for roguelike genre with the support of visual editor to meet today's requirements.

In the game industry, using assets becomes more reputable than ever. Therefore, building a framework on top of a game engine can show significant results for the roguelike genre and be a showcase that improves the efficient and quality game development process.

Our framework is designed to achieve the usage not only by developers but also with artists. One of the gray lines in game development is separating the roles of developers and artist during the process. Most of the times, one may have to wait for another. A visual editor can unblock this problem by providing a common language between developers and artists over the same logical components which also have the power of manipulating the graphical artifacts.

Besides all, the proposed architecture which is the outcome of this thesis can even be used as an approach for the other game genres or extended for the different aims such as shader editing and level design.

1.2 Overview of the Area

Visual Scripting software products are being used to help non-programmers for creating a logical complement. One of the oldest examples by Ellis et al. [1969] - GRAIL - can be traced back to 1968. This marvelous tool was used to program the man-machine communication with a node-based editor on CRT monitor as it can be seen in Figure 1.2. Today, we use this kind of software products in many areas such as architectural simulation, civil engineering, material engineering, and artificial intelligence.

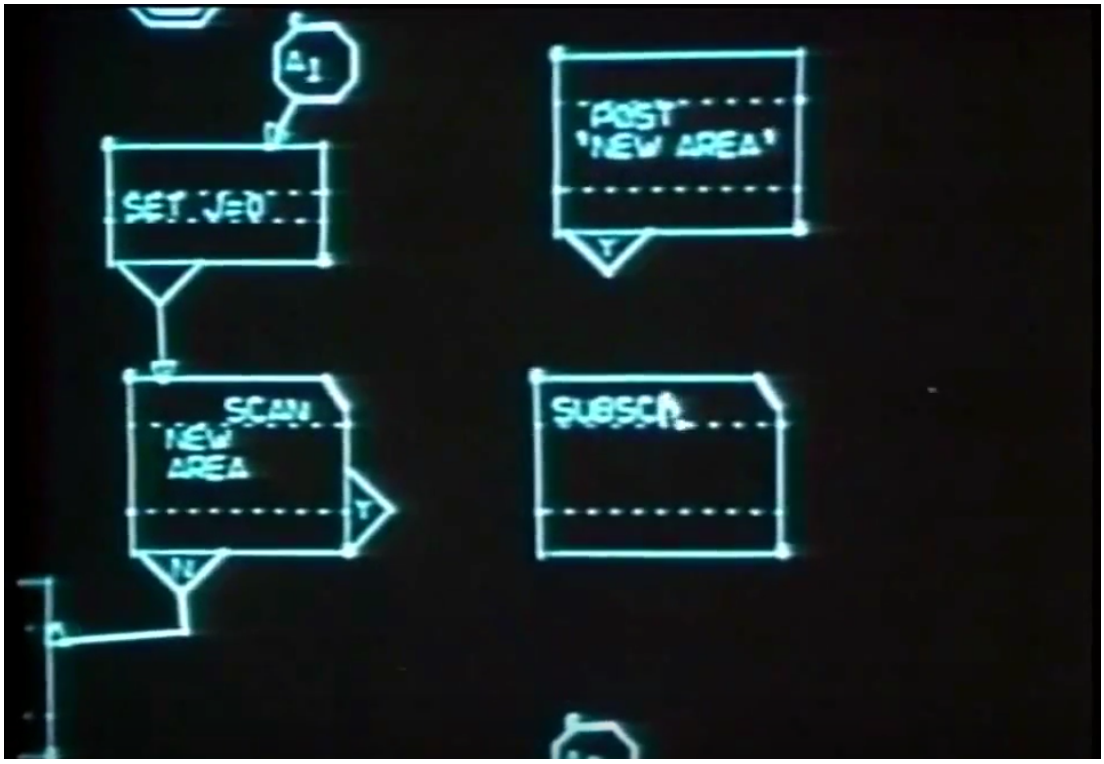

Figure 1.2: A screenshot from the GRAIL's interface

One of the most magnificent examples is the Unreal Engine's Blueprint Visual Scripting system which allows users to create gameplay elements using a node-based editor. It is quite famous among the user's of the engine, and it gives quite a low-level API to reach engine's core functionalities with its complete node set.

Another essential software is Unity3D Game Engine on the market today. It does not give a visual scripting support as default, but it has an enormous asset market which you can find a couple of visual scripting editors for different aims such as creating behavior trees (see ch6 in Rabin [2014]) for artificial intelligence, editing shaders, and building materials which keeps the information needed to simulate the surface visually and physically. The Unity Asset store is a free market. Therefore everyone can develop all kinds of assets and put there for free.

Even though there are some tools to support roguelike game development, there is not a visual framework as it is intended in the scope of this thesis. More

detailed information regarding this can be found in Related Work section. Before arriving that point, we should have a clear interpretation of what roguelike is to implement a generic editor. The definition of roguelike will be examined detailly by giving historical examples from the genre in the following section.

1.3 Roguelike Game Genre

Roguelike is one of the oldest game genres. Even in the times without graphical user interface (GUI), the versions with ASCII characters were quite famous back in the days and these games inspired and triggered a lot of games (see **?**). It will not be wrong to say that this genre is also inspired by J. R. R. Tolkien's books because it is common to see races like elves, dwarfs or elements like the spell (see **?**).

What we can do with computers has changed a lot during the time. However, the roguelikes did not change a lot even though we can do more with the current computation power and graphics cards. There is a subbranch which they call themselves modern roguelike which might appear as 3D or decorated with Role Playing Game (RPG) elements but we can still see games which follow the traditional roguelike every day. Despite the fact that roguelikes have a visible effect on the other games today, it mostly remained unknown tot he mainstream gamers.

1.3.1 What is a Roguelike?

"What is a roguelike?" is an endless question, and there is not only one answer to this question. Various perspectives can be applied to understand what "roguelike" means by evaluating its historical propagation and focusing its features.

Eryk Kopczyński and Čtrnáct [2017] defined the roguelike as a type that is often seen as a subgenre of RPG, but whereas RPG becomes the mainstream, roguelike moved into the other way by protecting its traditions and staying in the non-mainstream area.

This nature of the genre gives us to judge the other games by their roguelikeness. Despite the fact that there are a lot of standard features which can affect roguelikeness, design features of the genre are the most apparent one with the tile-based or ASCII based game world.

1.3.1.1 Berlin Interpretation

This definition of "Roguelike" was created at the International Roguelike Development Conference [2008]. It took place in Berlin. Therefore it is called as Berlin Interpretation. Most factors what makes a game roguelike is renewed, and some new elements have been added whereas some factors have been removed. Lacking some factors does not mean the game is not under the roguelike genre. Likewise, possessing some factors of the kind does not say the game is a roguelike. The aim of the Berlin Interpretation is not constraining the developers. The sole purpose

of finding a definition for the community was to have a better understanding what they are studying.

According to the Berlin Interpretation, there are high-value factors and low-value factors, and these factors are used as criteria when a game is evaluated as a roguelike or not.

High Factors

- **Procedural World Generation:** Major parts of the world in which the game is played are generated using a random maze/dungeon generation algorithm. Thus, every game is different than the others, and this feature makes the level playable many times.

- **Permanent Death:** In most of the games, the players can save their progress. However, this is different in roguelike games. When the main character dies, the player loses all the progress and starts over. This feature is not evaluated as a punishment since the newly generated level is procedurally created and different than the last one.

- **Turn-based:** The time is not an actor in the roguelikes. Every command of the player is accepted as one turn. For instance, the main character moves, then the enemies move. These game elements cannot move simultaneously like it is in the Chess game.

- **Grid-based World:** The game world is a rectangle and formed of tiles. The main character moves through the tiles, and a one-game element can fit in only one tile.

- **Non-modal (Freeform):** Everything is up to the player. There is no linear story-line. Choosing what to do, how and when does not have to affect the game result in between defined goals. The progress is part of the player's gameplay strategy since there are many ways to achieve the goals.

- **Complexity:** The game complexity should allow several solutions to reach the end depending on the goals of the game. Therefore, the game should contain enough interactions between items, monsters and the player. For example, the level should provide enough health potions to survive until the end of the level.

- **Resource Management:** The game should push the player to manage the limited amount of resources such as food or potions. These items have to be logically useful at the level. The player should not possess an item which is trivial to use in the aspect of gameplay.

- **Player vs. World:** The game is against the world. Thus the main point here is killing as many monsters as possible. It is also worth to mention that there is no interactions or communications between monsters.

- **Discovery Mechanics:** The game should allow the player to wander around the world and find ways to discover the usage of unknown items with some little cues.

Low Factors

- **Single player character:** The player controls a single entity. The character properties may change, but the game always puts the player in the center. The camera follows the main character, and whenever he or she dies, the game ends.
- **Player-like Monsters:** Enemies should have the similar properties to the main character. Thus, they can equip as the player does or have stats. They can be static and activate when the character attacks them, or a monster can have an artificial intelligence which fits the turn-based gameplay.
- **Strategical challenge:** The player has to define the strategies before making meaningful progress. Because the game can get challenging quickly and the player will not be able to make any progress until acquiring enough experience. The game should provide these challenges to create the real fun factor.
- **ASCII display:** Early roguelikes are all released with the ASCII display. Therefore it is quite common to see the games with ASCII display. However, today since the roguelikes with ASCII display may not attract the new players, the number of releases with this form is scarce.
- **Others:** Roguelikes often take the dungeon as the game world. However, this is not a significant requirement. The game world can be a cave, maze or another imaginary place composed of rooms and corridors. Another point is showing the stats such as health points, attack power or defending attributes.

Controversy: Berlin Interpretation declared in 2008, and the game industry has changed a lot since then. Some part of the community claims that Berlin Interpretation is not valid anymore since it is out-dated and restrictive for such an open genre. The game aesthetics should be significant as much as gameplay mechanics are. Especially some features like ASCII display should remain respectable, however, applying this kind of old features to the game does not make it more roguelike.

1.3.2 Roguelike Examples

Significant roguelikes from the past are listed below. The list follows the historical development of them.

Rogue: Rogue was written in 1980 by Michael Toy, Glenn Wichman and Ken Arnold for Unix. A plenty of ported versions have been created for several platforms. The most important fact regarding this game is that it established the genre and inspired all other Roguelikes.

One of the distinctive features that leave this game out from the RPGs at that time, all levels were randomly generated. Since it is designed to be played on Unix terminals, Figure 1.3 shows that the dungeon was displayed in ASCII

mode. The main character, monsters, and other game elements are represented by letters. The in-game actions were commanded by single keystrokes. It gives several features to the genre such as random world generation, tile-based world, ASCII based display.

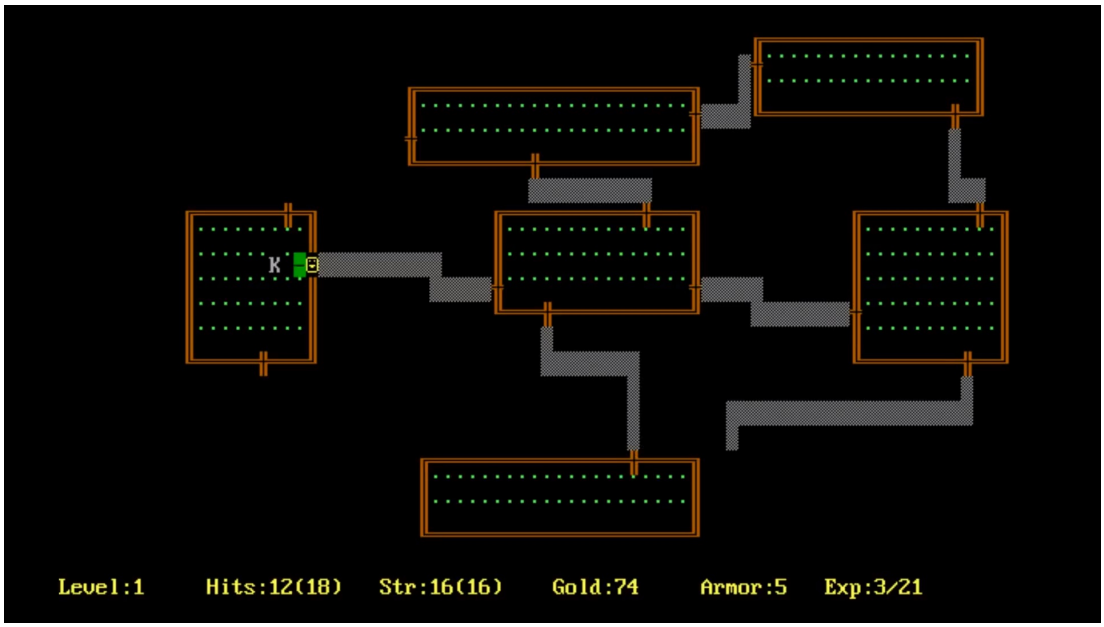

Figure 1.3: The Rogue port to IBM PC(1984) (Source: sourceforge.net)

Hack: Hack was written by Jay Fenlason in 1985 for Unix. It is one of the successful successors of Rogue. Several variants were released for DOS and the Atari ST. There are even current modern versions of the Hack, and some called hacklike.

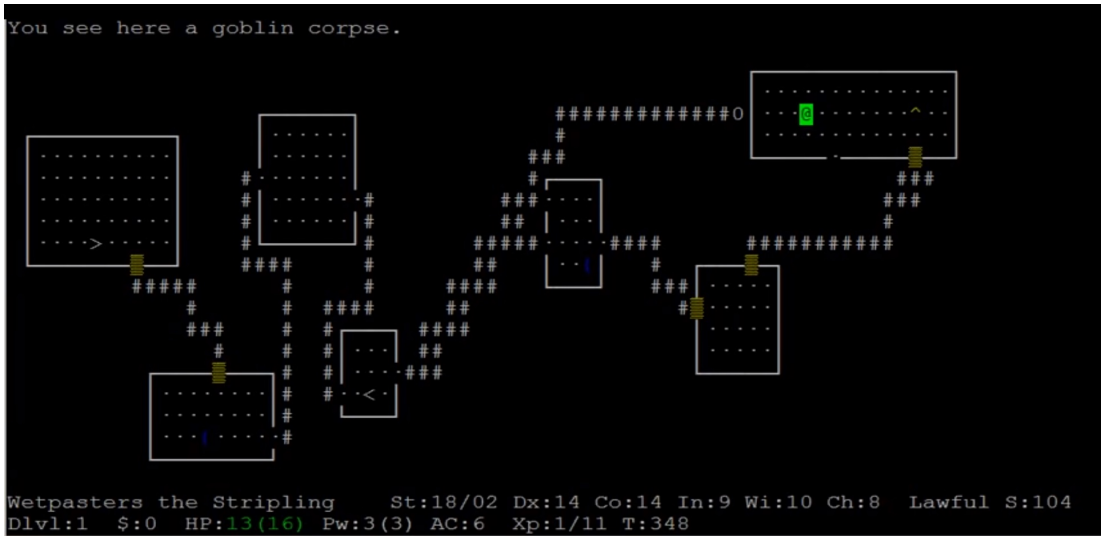

Figure 1.4: The NetHack: a port of Hack(1987) (Source: sourceforge.net)

Hack introduced significant new features, such as a dog following the character, several new races, even some shops where the player can acquire items. The game focuses on resource management. Thus, the player should fulfill this vital requirement, and feed the character correctly. Otherwise, the game ends due to starving.

One of the most prominent Hack variant was NetHack in the Figure 1.4. It has stayed in active development for 15 years. There are a few notable features of the game besides the resource management. Monsters and items can interact with each other in many ways, and this interaction can cause unpredictable consequences. The character has to define a strategy to collect the items.

Larn: Noah Morgan released the Larn for the first time in 1986. As the other predecessors, Larn is released to be played on Unix terminal. Many ports have been developed for the other platforms as well. An AMIGA port can be seen in Figure 1.5.

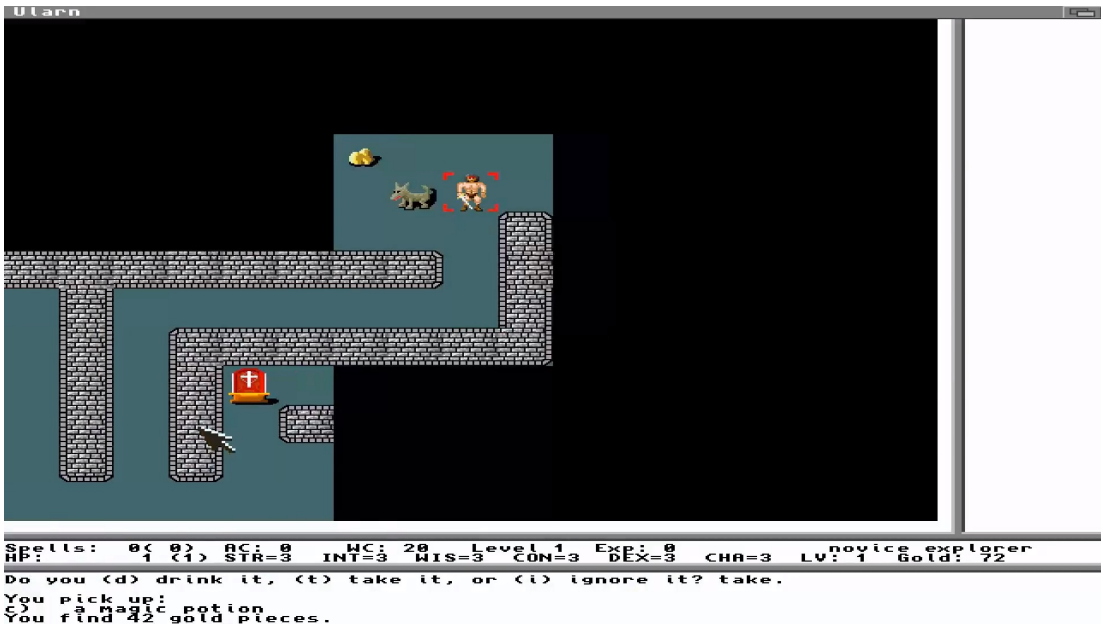

Figure 1.5: The Larn's AMIGA port: ULarn (Source: sourceforge.net)

Larn uses shops to exchange items, the character's experience matters more than the items the player is equipped, the depth of the dungeon weights the item generation, and there's a town that must repeatedly be returned. Resource management is also critical here as it is for Hack.

Field of vision is handled here by providing two tiles vision in every direction. Contrary to the other roguelike, time is an effective factor in Larn to push the player to wander around more levels and find more food to survive.

ADOM: Thomas Biskup released the ADOM in 1994. The storyline is quite rich, and it is one of the most successful roguelikes in the Steam. Early versions used ASCII style display, but it is evolved in time as it is seen from Figure 1.6.

Unlike the other predecessors, the source code is closed. The primary objective of the game is stopping the chaos and disorder. There are also multiple endings to the game even though it is a plot-driven game.

Figure 1.6: Stunning UI design of the ADOM (Source: steam.com)

We can count ADOM as one of the hacklike games. Some parts of the levels are persistent, and rest is non-persistent. Additional to the other roguelike, it provides a significant number of classes and races. Magic and items are available to every race without restriction.

The user interface is relatively more modern, and acceptable for the users even from out of the genre's patronizers.

Dwarf Fortress: Bay 12 Games started to develop in October 2002, and the first version of the game was released in August 2006 with a conventional ASCII style display. The game was combining both the elements from roguelike and strategy genres. The game took a considerable amount of interest from the community with its unique gameplay style.

Even though the game is released with the ASCII display, there are many modes which players can enjoy this game even in 3D mode as it is on Figure 1.7.

Before the gameplay, a world is procedurally generated as it is done in the other games. However, each world is constructed differently; a terrain is generated fractally, then an erosion applied. Towns and wildlife creatures are placed. The world creation was taking 15 minutes back in the days. There are two modes as Adventurer and Dwarf Fortress. Former is created to be played in a roguelike manner, whereas the latter is more strategy genre.

Figure 1.7: A 3D mode of Dwarf Fortress

The Greedy Cave: The Greedy Cave: Avalon-Games released The Greed Cave for the mobile platforms, and it has been played more than a million players based on the market data. It apparently successfully combined traditional roguelike features with the mobile platform. It is easy to control, and the game uses A* algorithms for the player navigation. Monsters keep their static state until the player attacks them.

Figure 1.8: The Greedy Cave: A mobile adaptation of roguelike genre (Source: avalon-games.com)

There are 80 levels, and it is possible to move between levels. Every ten levels, the world appearance, and the monster types are changing. Items that can be

found in the higher levels give more power than the items in the prior levels. Figure 1.8 shows that the game graphics are inviting for all type of gamers. Experience is one of the essential properties, and it matters to survive more at the higher levels.

Unlike the other roguelikes, the greedy cave gives a way to carry the items earned during the game. There is a specific item which gets the character out of the game session and all the golden items carried out and usable in another gameplay. However, the player has to start the first level even after using this powerful item.

1.3.3 Common Features of the Genre

Roguelike is an old genre and evolution of it is continuing and keep player base active. Since the first release of Rogue in 1980, a new feature is added to the next roguelikes. Some of these features accepted as a roguelike feature and some did not.

Even though trying to define what is roguelike can be controversial, the research which takes place in this thesis require some set of features to provide an architecture which supports or may support the features.After a detailed examination of the roguelikes and definitions, the common elements of the genre can be listed as follows.

- **Procedural content creation:** Even though some of the games are semi-persistent like ADOM, all the roguelikes contains procedural content creation. This feature is not even limited to world creation. Also, the story can be created as procedurally.
- **Permanent death:** This feature and procedural world generation create a coherence and eliminate the frustration that might appear due to permanent death. Some recent examples provide ways to keep the items during one game session, but it is not allowed to start the game from the place where the character is dead.
- **Turn-based game in the grid-based world:** The game level can be represented as ASCII display, tile-based graphics, isometric representation or 3D models however the game world is always grid-based. The character and the enemies have a turn-based gameplay mechanism. This rule is also valid for attacking. The game elements cannot attack simultaneously.
- **Single Character with an Inventory:** Players can control only a single character. Some games like Hack provides a pet dog which follows the main character, but still, the player controls the individual entity. Characters have an inventory which might contain cloth, magical items or weapons which does not have a minor use in the session.
- **Discoverability:** The game should allow the wander around the level and discover the world. Field of Vision algorithms and in-game maps are used to make this point strong.

1.4 Aims and Scope

The primary goal of the thesis is providing an architecture for a framework that provides a visual node editor to aid the creation of roguelike games with requiring minimum code. The architecture should be designed in a way that the outcome is open to both developers and designers to use and tweak the game.

After a detailed search Unity game engine is decided as the development environment. Since the framework is supposed to be developed as a plug-in for this environment, the framework should be compatible with the engine requirements.

The node editor shall support;

- a simple project persistence to save the project files and open them,
- subgraphs to reuse them by providing sub persistence form them,
- a sidebar which keeps the nodes,
- a menu to navigate,
- built-in documentation for nodes,
- an abstract structure so that it can be specialized to create roguelike games.
- example node editor specialization tailored for roguelikes

An interpretation module should be provided to interpret this node graph with the compatibility of Unity3D. The prototype should be adequate to prove the provided architecture can create roguelike games encapsulating the game-related code.

Therefore the scope of the thesis will be limited to;

1. the framework architecture,
2. a prototype of the framework,
3. an experiment to attempt creating a roguelike game,
4. the game which is the outcome of the experiment.

1.5 Methodology

Before the software design, a detailed requirement analysis should be done. The framework will be a plug-in for the Unity3D game engine and support roguelike game creation. Therefore, this project requires an analysis both on roguelike games and Unity3D game engine. The features of the roguelike genre is already extracted in the previous sections. Related works and libraries will be examined as the next. Since the Unity3D game engine is a commercial and closed-source software, a model that can be reliable will be deducted from the documentation as well.

Software design will be divided into modules, and every module will be designed inter-dependently considering the Unity's architecture and roguelike game requirements.

A prototype will be developed around a node editor which is flexible to support the meaningful game tree using the architecture and software design. The interpreter module will be developed to translate the node graph into a Unity3D game project.

A demo game that has 3-5 levels will be created using the prototype to show that framework supports or may support the games which can be evaluated as a roguelike.

1.6 Outline of the Thesis

The chapters are placed in a way that a chapter will require knowledge from the previous one. Therefore, reading it from further chapters may harm the understanding.

The rest of the thesis is organized as follows. Chapter 2 provides some decent related work with a detailed examination of them. Chapter 3 explains how the underlying infrastructure is and Unity game engine works. The chapter tries to touch the vital points for this thesis.

Chapter 4 gives a detailed explanation of the proposed architecture for the roguelike game framework. It is divided into modules, and every module has the same structure to ease the following for the reader. Chapter 5 tries to create a sample roguelike game empowered by the framework. The last chapter checks if the thesis met with its goals, and recommends some future improvements for the framework.

2. Related Work

In this chapter, several libraries that are already exist for roguelike genre will be examined and show which features they support. These libraries are still valid for several platforms. It is worth to mention that these libraries are all independent and they ease the development, and none of them is created by considering to support a game engine.

2.1 T-Engine

This engine is one of the alive middlewares in the roguelike world. It is powering Tales of Maj'Eyal (ToME) starting from earlier versions to now. ToME and the engine were together before the last release T-Engine. They saw that the engine improves in accordance with the ToME and this was a considerable drawback which restricts the usage. It is entirely rewritten in Lua for the last version and separated from the ToME.

T-Engine 4 provides interchangeable module-based architecture, and one really can create original content. The engine is flexible and comfortable enough to moderate for the experienced developers.

In the backend, they use OpenGL, and the creators claim that the rendering is "fast enough". The engine supports ASCII or tile-based display. Since it has to operate as an engine, it gives a standard definition which is called as the entity. This definition covers objects, the character, items, monsters and even the terrain.

The engine tries to give quite a flexible customization environment for the in-game actors via integrated interfaces to the game. It comes with a lot of utility modules which support character generation, particles, map handling, sound, and music.

Inputs are handled with a key binding system which the most engines support out there. It is a very convenient approach because the user can assign a key or multiple keys to an action and then use in the game.

Unfortunately, there are not enough sources online, and since it is an individual engine, it is not more popular than the ones used by the developer communities like Unreal Engine and Unity3D. However, it has a dedicated documentation which tries to explain how to use it.

The engine lacks an editor, but its modular design helps to increase ease of usage. Even though, it gives an adequate support to create a sophisticated roguelike game, not so many games use this engine.

2.2 Doryen Library

Jice released the Doryen Library in python language in 2008. It won't be wrong the say that Doryen is a toolkit which contains a bunch of small libraries to support roguelike game creation such as map generation, the field of view and display handling.

The core library is written in C language, but there is python wrapper used. It is not intended as a library, but it is created during a roguelike game development process by one developer. Then the library is released by him upon the request of the community. From that day to now, it became a community-driven library. It has a quite active forum which developers help each other.

"Doryen is not a roguelike framework. It is just a toolkit. So you really should use the part you needed. It's not something that builds the game for you. You just customize what you want for your gameplay extras. It's really small simple tool." said Jice during an episode of the roguelike radio. This aspect of the Doryen is a significant and considerable tool because the library does not push the developers in a single direction. Developers use this library by adjusting it following their needs.

There is a firm support for randomness. The library has wrapped algorithms such as Mersenne Twister, Complementary Multiply, Perlin, Simplex and Wavelet noise. There is even a name generator attached to the library. There are plenty of field of vision algorithms are available to use. The time in which the library is released, it was excellent to have a windowed or full-screen console with the true-color display.

The library is easy to use and a combination of tools. Dozens of games are created and released empowered by this toolkit. Some of the users of the library abandoned to use it due to aesthetical reasons. It is also worth to mention that the library has ported to C, C++ and C#. Also it is possible to get builds for Android.

2.3 Carceri

Carceri is one of the tools which is abandoned without any public release. However, the approach of the instrument can make a significant effect on this thesis since it is using data-based game configuration. Therefore, it is worth to examine it.

Kornel Kisielewicz declared the details about Carceri for the first time in 2005. The development took three years with FreePascal language, and development details are shared with the community frequently. However, the author announced that the project was canceled and there will be no longer any releases.

Carceri's approach was a bit different than its predecessors. It intended to be an engine based on the data files. Therefore the audience for the tool wasn't only developers. The typical flow is designed as follows. The user creates or edits some human-readable text files, then these text files interpreted into an executable roguelike game.

Roguelike game development does not involve any scripting. Everything is supposed to be entirely data-driven with some exceptions on event system. Having all those triggers based on text files is not an easy task to do. Therefore, the fighting simulator, inventory, the magic system will all be hard-coded in the tool. However, the items in the stock could be defined by the user.

A roguelike game called Angband which is one of the most successful roguelikes allows to mod the game through data files. Carceri wanted to take this feature one step ahead and create an entirely different roguelike game based on text files.

Carceri has never focused on huge roguelikes, on the contrary, it focused on more smaller levels with a simple plot support. The debatable thing was that game levels are meant to be persistent without procedural world generation involved.

2.4 The Ng Java Roguelike Engine

Thomas Seufert released the Ng java Roguelike Engine (JRLE), and it is still in beta version. The project contains three significant modules; the core, the variant manager, and the prototype.

The core module is written in service-based standards. Therefore, developers can use some part of the library and ignore the rest. It mainly divides into two submodules; event system, utility tools for roguelikes. The event system is based on the game board on which all the events occur. So there is a BoardListener which is notified of creature moves. Separating the event system from the rest of the core methods is significantly effective.

The Variant Manager creates the project files. A project is a directory structure containing XML files. The variant manager is provided with an editor, so it is easy to edit project's high-level data. One of the essential features of the tool is providing the multiple language support for the roguelike games. Therefore language and word management are done separately. This core feature also prevents user to hardcode words in the code.

Despite the fact that the tool is providing quite exciting features, there is no user base or games which are empowered by the engine.

2.5 H-World 2004

The real intention of the H-World engine is providing a tool for RPG games. However, the abstraction in the tool is made in a way which creates an opportunity for roguelikes as well. The design features of the engine might have an impact on the framework in the scope of this thesis.

The engine bundles a simple game with its distribution which the user edits and turns it into an entirely different game. The level editor screen is given in Figure 2.1. The last update was made in 2004, and it works on Windows and Linux platforms.

The engine allows the user to use tile-based or isometric display. Lua scripting language is preferred to extend the abilities of the engine. The turn-based game is supported by the engine for a single character. Multiple entities are also supported as it was in the Hack style roguelike. A built-in artificial engine support comes with the distribution. Entity actions are managed elegantly during the combat or non-combat.

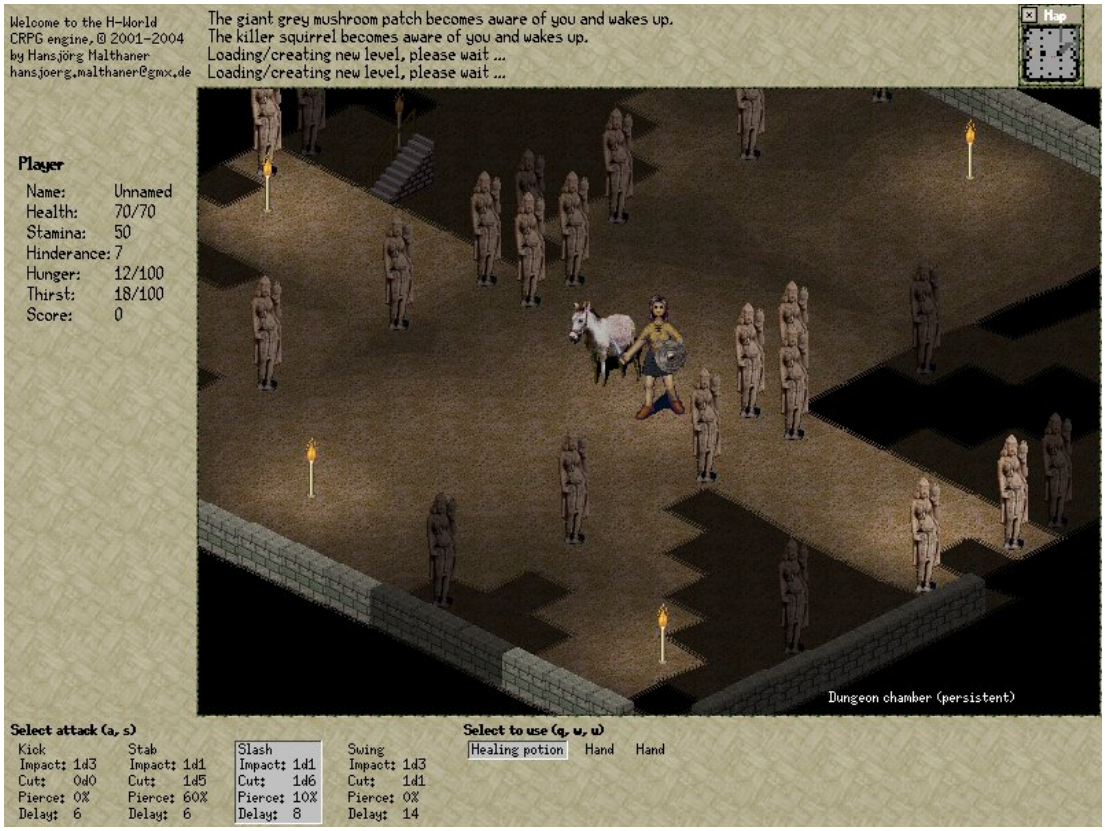

Figure 2.1: Level editor screen of the H-World

There is a trading system for items. Most of the engines might have skipped this feature because this is primarily an RPG feature. However, it can be observed that the latest roguelike also provide some item exchange systems. This item trading also gives some alternatives to the user for defining a strategy based on already acquired items. However, H-world is doing it in a barter base trading system. Therefore, it is quite a high-level component which user may not customize as it is intended.

As a conclusion, H-World tool is not intended to support roguelike game genre. It does not even support non-persistent game world as default. However, it is possible to create roguelikes, and it gives a unique way to create a game with its in-game editor.

2.6 A Brief Comparison

Five tools are described above to analyze extractable features of the roguelikes. These tools might be outdated or abandoned projects. Most of the developers claim that they use only their libraries since these tools are all platform specific. They might not even fully support all the characteristics of the roguelikes as it is seen in Figure 2.2 however it is worth to get inspired by these tools.

Unfortunately, there is no tool to make a roguelike game development which fits today's industry standards. These tools are all outdated or require individual effort to learn how to use them. Even though the major game engines do not

Comparison between Roguelike Libraries					
Feature	T-Engine	Doryen	Carceri	JRLE	H-World
Procedural World Creation	YES	YES	NO	YES	NO
Permadeath	YES	YES	YES	YES	WITH SCRIPT
Turn-based	YES	YES	YES	YES	YES
Grid-based	YES	YES	YES	YES	YES
Single-character	YES	YES	YES	YES	YES
Inventory	YES	NO	YES	YES	YES
Discover-ability	YES	YES	NO	YES	WITH SCRIPT

Figure 2.2: Comparison table for Roguelike Middlewares

provide extra features to support roguelike, they are the best tools and promise the fastest development time. This implies that there is a room for the implementation of a new roguelike framework.

2.7 Visual Node-based Code Editors

Programming with node-based editors is not a new thing. However, text-based editors are more preferred option since there is a huge IDE support for them. There have been some studies (e.g. Asenov and Muller [2014]) which try to prove visual programming (VP) might be as fast as text-based programming and the results show that visual programming can be quite fast depending on the tool.

In this section, five different VP tools are described. There are also other software or research results. However, these tools are selected to represent various

approaches to VP. Moreover, these tools also prove this type of programming can be used for reaching different goals and can be created for domain specifically.

2.7.1 Envision

Envision is an independent tool which works for Java language developed by Asenov and Muller [2014]. The approach focuses on fluid Interactions. Experienced developers feel more comfortable with text-based programming, and the reason is keyboard interactions make them fast on coding. The authors observe that most of the other tools depend on mouse interactions, but they claim that it is a problem due to mouse movement's sluggish interactivity.

Envision's second concern is the performance and scalability. The chance to have success on a VP tool depend on how it handles the big projects. A project can contain millions of lines of code, therefore, handling this size visually might be unrealistic. However, Envision takes this issue and tries to solve it elegantly. Most of the tools out there provide only one-way conversion which is node-graph to computer code. Envision can be used in both ways since it is using the AST tree as a transition concept. Therefore, it can be used ever for the past java projects.

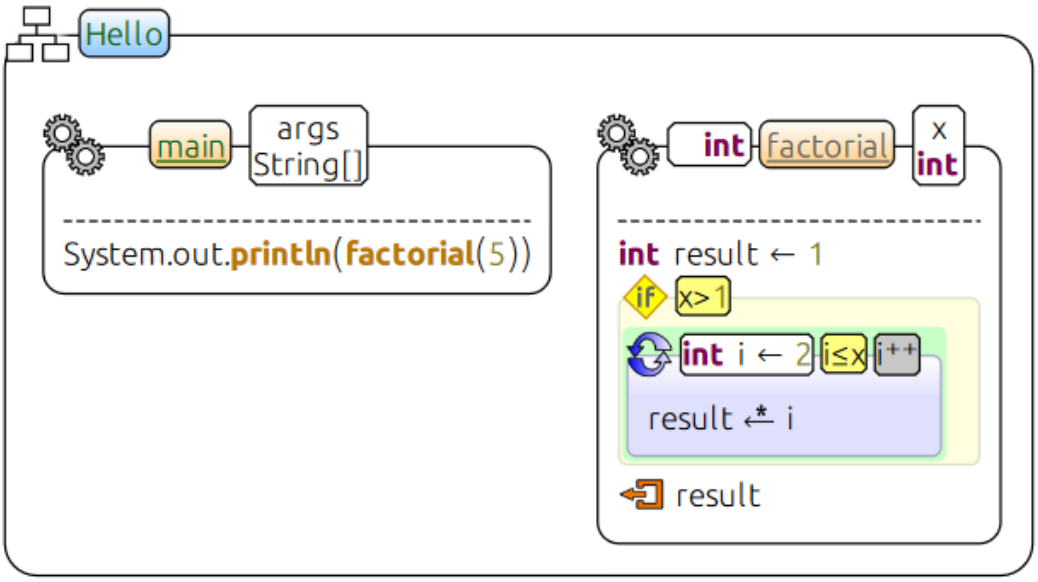

Figure 2.3: A java class that prints the factorial of 5 (Source: Asenov and Muller [2014])

The tool works like an IDE which uses a visual display instead of text and interactions are done by the keyboard controller. Please see Figure 2.3 There is a real-time code to AST conversion, and AST is used for the construction of the visual representation. It is also worth to mention that it supports a visual way for code commenting as well. The user can put comments with shapes. Therefore, this will make this tool quite strong for documenting the code.

Key points of the tool:

- Keyboard interactions over mouse interactions
- Two-ways conversion between the code and node-graph
- Quality commenting support

2.7.2 Unreal Engine Blueprints Visual Scripting

Unreal Engine's Blueprints is one of the most robust pioneers in the industry. It provides a complete solution for Unreal Engine, and it is possible to create a game only using the blueprints. Even though the tool is sophisticated, the approach is simple. There is always a node which corresponds to classes or objects in the engine. Linking these entities creates logical functions and users can design their game logic using the blueprints efficiently.

A mouse is enough to operate the blueprint. However, the keyboard and mouse give more efficient workflow when they are used together. The company provides a decent amount of tutorials. Therefore it is easy to grasp the system even for the designers. As it is mentioned earlier, developers adapt text-based conditions easier than VP environments. Therefore it is also possible to write the code in Blueprint-specific markup which is possible by Unreal Engine's C++ implementation. Then this code can be transformed into blueprint which is available to manipulate by the designer.

Even though using the blueprints is optional, this feature comes with the engine distribution package. Therefore this component is developed by the engine's developers. The feature has a high compatibility with the Unreal Engine API. The engine's community is enormous and it has a giant user base.

Key points of the tool:

- Bundled with the engine
- Support for designers
- It can be combined with the text-based code
- Extensibility

2.7.3 Bolt

Bolt is a visual scripting plug-in which is targeted to the Unity3D game engine and used by many users. Unfortunately, Unity3D does not have a native VP support. Therefore, there is a plenty of number extension available for the users. The approach of Bolt is quite specific and uses reflection which is known as its slowness. However, Bolt claims that their reflection library provides six times faster functionality than the other ones in the industry.

Both flow graphs and state graphs are supported instead of choosing one of them. Mouse and keyboard commands are used together during the construction the graph. It uses icons for the nodes which gives an idea of the type, and the

general user interface design is quality. Since it uses reflection, it supports not only Unity API but also the third party libraries.

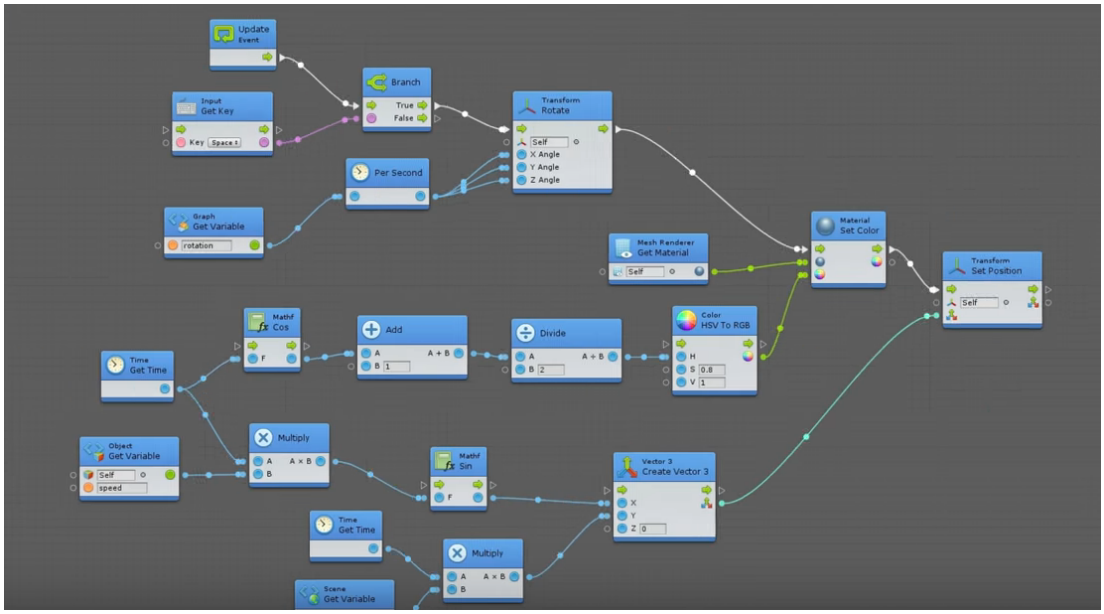

Figure 2.4: Bolt scripting with node editor. (Source: assetstore.unity3d.com)

Bolt provides debugging, and Figure 2.4 shows the code flow visually as well. Therefore following the state or the flow is almost effortless. The documentation quality is adequate, and there is a decent number of tutorials for the users who want to learn the tool. There is a large community since this tool is the most used VP tool on Unity3D.

Key points of the tool:

- Fast reflection
- Visual Debugger
- Support for third-party libraries

2.7.4 Amplify Shader

Applify Shader is a real-time node-based shader editor plug-in which is released for Unity3D. It is awarded as the best tool in the store in 2017. This award not only proves the success of the tool, but it also shows that node-based editor can have a meaning other than scripting. Especially, shader creation is not an area which developers can do easily due to infrastructure. Even though there are plenty of tools to support shaders, the quality of those tools may not be satisfying. Therefore, Amplify Shader is an excellent example of all the shader tools.

Every graph defines only one shader which is compatible with the Unity3D. The user interface is kept minimalistic, and the tool comes with many templates. Repetitive actions which are done during a shader development became nodes

such as lerp or color blend. There are plenty of shader functions available for the developers. It is possible to see how a slow and hard to debug process can turn into five-minute work with node-based editors.

Applify Shader supports many platforms such as mobile, Xbox and PlayStation. The tool is updated regularly and allow material editing as well. The node API is provided to extend the tool, and an adequate documentation is available for the users.

Key points of the tool:

- Works in real-time
- Provide templates
- Domain specific and handles every area of the domain

2.7.5 Behavior Designer

Behavior Designer is an example of a Behavior Tree implementation for the Unity3D game engine. The tool is using a visual editor to implement AI for the game entities. Figure 2.5 can give a idea how the tree looks. There is a broad support with hundreds of built-in actions, and it works even for the multiplayer games.

The API is open to the developers. Therefore new actions can be easily added. The users can preview the behavior state and given decisions in real time. Thus, it is easy to detect the errors in many circumstances. The tool also comes with an integrated debugger which users can add breakpoints and track the state step by step.

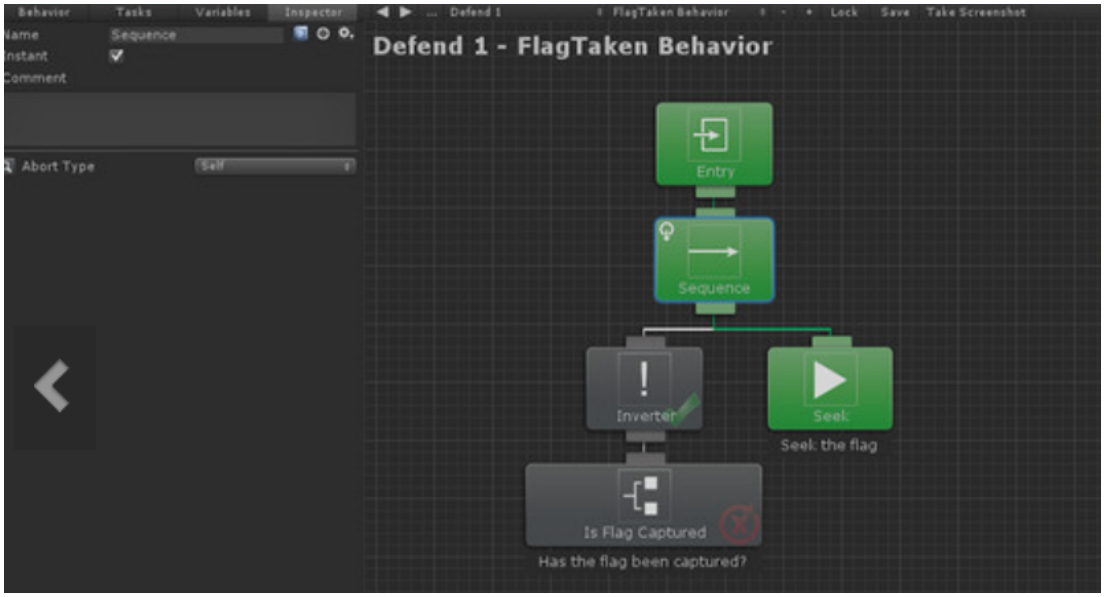

Figure 2.5: Behavior Tree demonstration (Source: assetstore.unity3d.com)

After the initialization of the related component, there is no memory allocation applied thanks to its data-oriented design. The tool also comes with a local event system. Therefore the behavior tree can be triggered by an event, or it can send an event as an action.

Key points of the tool:

- Data-oriented design
- Built-in event system
- Flexible API

2.8 Conclusion

The engines mentioned in the previous sections could not have a significant impact or could not protect their user base due to modern game engines and training requirements. However, the tools that have been integrated into those modern game engines can be even sold to thousands of users which creates a community. Accordingly, developing a roguelike framework with this approach will have a better chance to survive.

Today's biggest asset store is provided by the Unity, and the support and documentation are in a quality state (see Messaoudi et al. [2015]). Therefore, the roguelike framework targeted in the scope of this thesis will be proposed as a Unity plugin since there are already examples of visual programming tools. Our tool's functional requirements should be as follows;

- The framework should have a visual generic node editor which carries a node graph
- The node graph should be persistable on the disk
- The nodes can carry images, object or prefabs.
- Code and data-file generation should be applied by interpreting the nodes.
- The output of the framework should comply with the Unity's game project format and its architecture.

In the next chapter, significant features of the Unity will be discussed for the goals of the framework.

3. Introduction to Unity3D Asset Development

Unity3D allows developers to create an asset and use them in the game projects. The asset development can be straightforward for small assets. However, the assets like in the scope of this thesis pushes the limits of the engine. Therefore a detailed analysis should be done to adapt the engine efficiently. For instance, the functionalities like reflection and serialization are handled by Unity3D differently with several restrictions. Since the engine is flexible, these obstacles can be eliminated.

This chapter will try to elaborate if Unity3D is feasible to create a framework for roguelike genre with a visual editor. Architecture details and capabilities of the engine will be shown in the following sections.

3.1 Fundamentals of the Unity3D Game Engine

Unity3D Game Engine is one of the commonly used game engines. The number of registered developers is more than 4,5 million. Every month more than a million users logins to the engine. Almost half of the mobile game developers published games with Unity (see Public Relations Report [2017]).

After launching the Unity Asset Store, users started to sell assets to each other. According to some estimates, Asset Store saved game developers about $1 billion only in 2013 since there are many tools and libraries published there. Therefore, developers do now have to write the same functionalities repeatedly.

There is a big community behind, an unlimited amount of tutorials and blog posts can be found to get help. The company provides video lectures and live session given by professional game developers. The engine is updated based on the latest developments in computer graphics field. The high-level features of the Unity3D are listed below.

- More than 25 supported platforms including web
- Proven learning curve
- Community
- Native output
- 2D and 3D support
- Extensible API

There are many low-level details of the engine. However, this chapter mainly investigates the points that are significant for the scope of the thesis.

3.2 Architecture

Unity3D is a closed-source engine since it is a commercial tool. However, it has quite extensive documentation (see Unity User Documentation [2017]) which might give a chance to deduct the architecture of the engine. However, the results shown in this section may not be correct, but it shall be reliable since the information is based on the official documentation.

First of all, a Unity project follows a model that is the same for all the game projects. Every Unity game contains at least one scene. The scene covers scene tree, therefore, game objects. Please see Figure 3.1. Every game object has a component-based structure. These components can be predefined structures or scripts which are handled by the engine based on the game settings. Besides those, a game can contain third-party libraries or toolkits which are located in the assets folder.

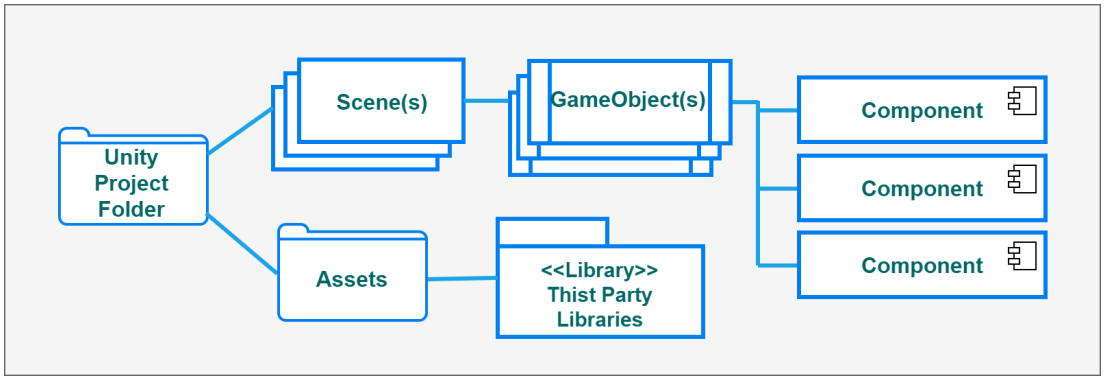

Figure 3.1: Unity Project Structure

Besise the model above, there is no project definition file for a unity project. A folder represents a project, and it is recognized by the editor. Every script is accompanied by a metafile and if a change made an impact on that file metadata is updated. Unity keeps track of these metadata. All these flow is happening automatically by the Unity.

Component-based design is supported by Game Loop pattern and Update Method patters. Every cycle of the game loop, update method of every active game object's components are called (See Game Loop Design Pattern and Update Method Design Pattern in Nystrom [2014]). Therefore one game object might be affected by multiple update methods since unity using a property-centric method instead of a game entity-centric model. So it will not be wrong to say a game object is a container for components.

3.2.1 How to build a game in Unity3D?

For the sake of the framework's architecture in the scope of the thesis, the list given should be followed to create a game using Unity game engine.

- Create a new game using the wizard (this will create the game folder).

- Drag and drop the assets which are going to be used in the game into the assets folder.
- Create a new scene file which may represent a level in the game.
- Decorate every scene with the assets in the assets folder
- Adjust the camera settings
- Add the built-in components to the game object in an intended way
- Customize the behavior of the game objects by adding scripts
- Test the game using the play pause functionality in the editor
- Build the game for the desired platform

The workflow given is done by the developers manually, and the rest is handled by the system. The aim of the thesis is automating some of those steps with the information acquired from the nodes.

3.2.2 High-Level Architecture

Unity engine is written in C/C++. However, the logic is managed using C# or UnityScript (a typesafe derivation of javascript). Developers can use C# and UnityScript freely. If this is the case, the engine creates two different assemblies. The only drawback here is that one cannot refer the other one. (see Public Relations Report [2017].) Therefore, a single language is more appropriate to use and have a consistent workflow.

There is another condition in which unity creates a separate assembly. If the developer intends extending the editor, then unity will handle this situation separately. The most significant impact of this condition for this thesis is when code generation involves from the editor extension, the code will be generated in the game assembly. Therefore the editor cannot reach the newly generated code.

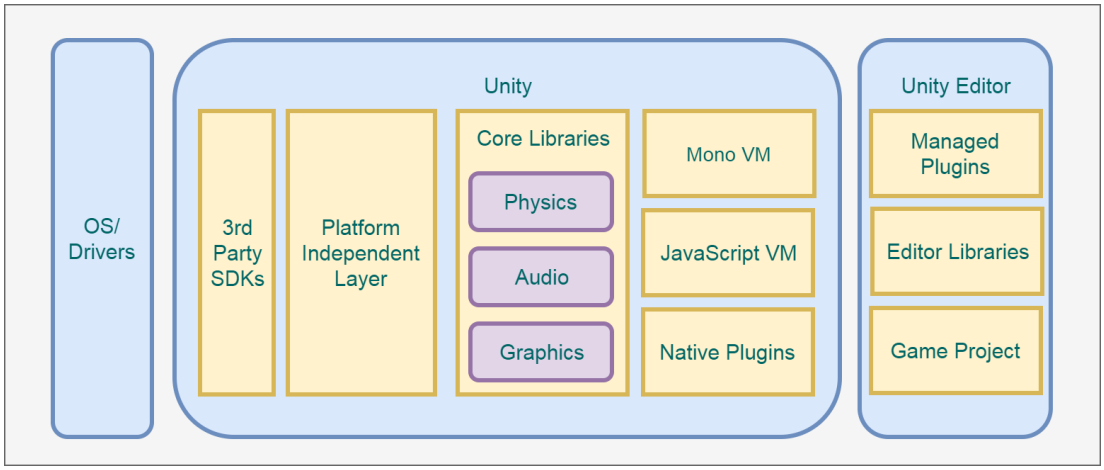

Figure 3.2: Unity High Level Architecture

This amount of analysis regarding the Unity's high-level architecture will be enough here to proceed with the thesis's goals. As a remark, the architecture of Unity is much complex than Figure 3.2, but this is the level of detail we need for the rest of the thesis.

3.2.3 Design of the Engine

The game object in the scene forms a tree structure, and there is a parent-child relationship between them. Children are affected by the manipulations of parents. Parent has an interpreter role even though the child object seems to be operable independently.

All the things in the engine are extending one class: Object. This class is the root of the class diagram and helps to share some similar features among all the objects in the engine. A GameObject class is accompanied by every object in the scene tree. A game object has the one-to-many relationship with the components (see Component Design Pattern in Nystrom [2014]). However, an entity might exist without components except Transform. This component is compulsory for every game object. Please see Figure 3.3.

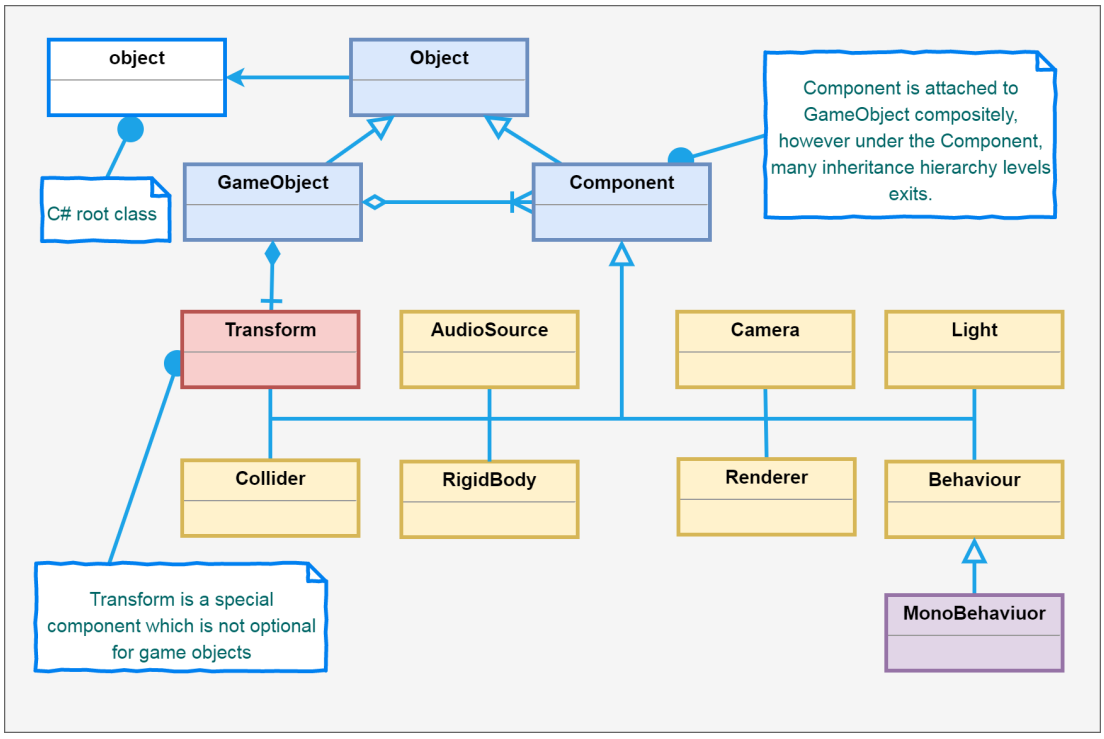

Figure 3.3: Unity Domain Modal Diagram

As it is mentioned earlier, Unity is a component-based game engine. Therefore during a component creation, the user can reach the GameObject and Transform objects. This design decision creates double referencing between GameObject and Components as it is seen in Figure 3.4. Even though Component is the parent of the Transform, it contains a reference to the transform object due to convenience.

These points can seem like a design flaw. However, it eases the scripting for the developers.

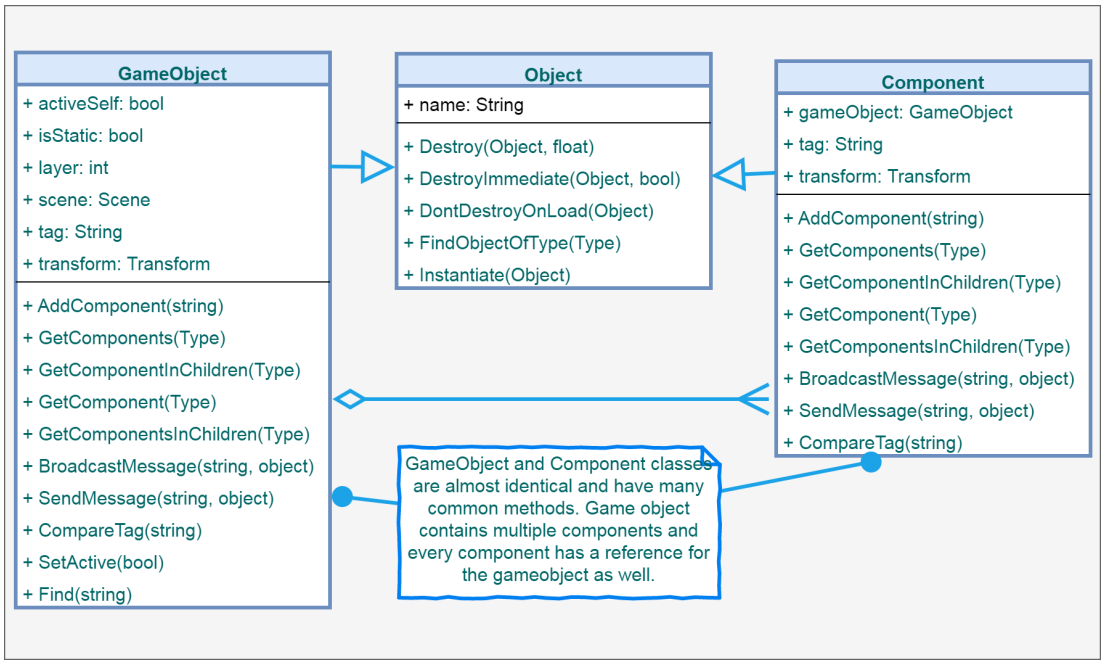

Figure 3.4: GameObject vs. Component in Unity

Components are the extension point of the Unity. There are a couple of components provided by Unity like Colliders. However, GameObject is sealed and wraps static methods which may manipulate the scene tree.

3.2.4 MonoBehaviour Class

All the structure of Unity is significant, but MonoBehaviour Class has a vital role in customization. All the scripts which are attachable to the game objects extend this class and Unity has a running pipeline for the MonoBehaviour.

MonoBehaviour extends Behaviour class, and Behavior extends the Component class as it is seen in Figure 3.5. Therefore, it is possible to see its details on the inspector window.

There are a couple of methods which involved in the Lifecycle. The list is given in their call order by the engine

- Awake (): It is called when the script is attached or reset.
- OnEnable(): It is called after the awake method or when the script is enabled at an arbitrary time.
- Start (): It is called during the initialization and called only once during the lifecycle.
- FixedUpdate (): Called once and stable during the physics cycle.

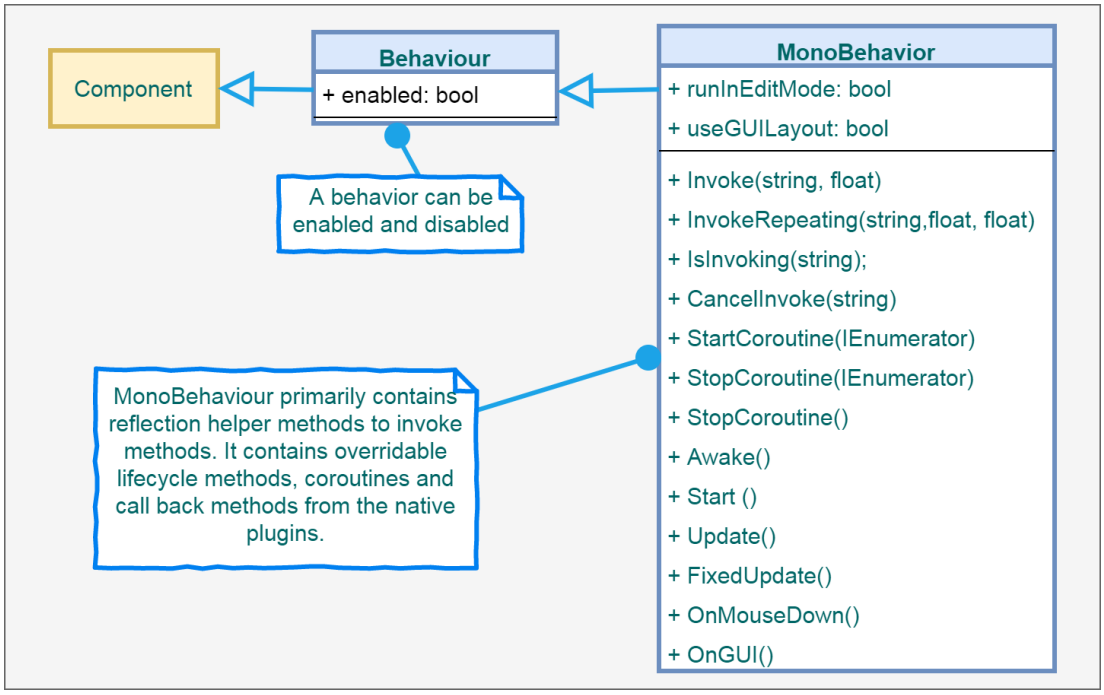

Figure 3.5: Design of the Monobehaviour Class

- Update (): Regular update method of the game logic which can be called more than once per cycle.
- LateUpdate (): Called at the end of the game logic cycle.
- OnGUI (): Called for GUI rendering. Can be called multiple times.
- OnDisable(): Called whenever the script is disabled.
- OnDestroy(): Called before the script is destroyed.

It is worth to mention that MonoBehaviour also contains callback methods to the other components such as Collider. When the object collides with another object which contains a collider, this callback method will be fired. As a side note, the code generated by the roguelike framework in the scope of this thesis should extend this class and attach it to a game object to make it functional.

3.3 Component-Based Design over Inheritance

It is mentioned that Unity3D preferred a component-based design approach. In this section, it will be explained why this choice has a significant impact and why the plugin development should be orientated in this way as well.

First of all, Component Pattern is a decoupling pattern (see Decoupling Patterns in Nystrom [2014]). It is applied when there is a class which touches different areas which are supposed to stay decoupled.

As it can be seen in the example below, inheritance creates an unnecessary complexity. The knight with the sword is duplicating the code from the other

branch. Inheritance is a quite general feature which fits in many situations. However, when it comes to game objects, inheritance makes the design overcomplicated.

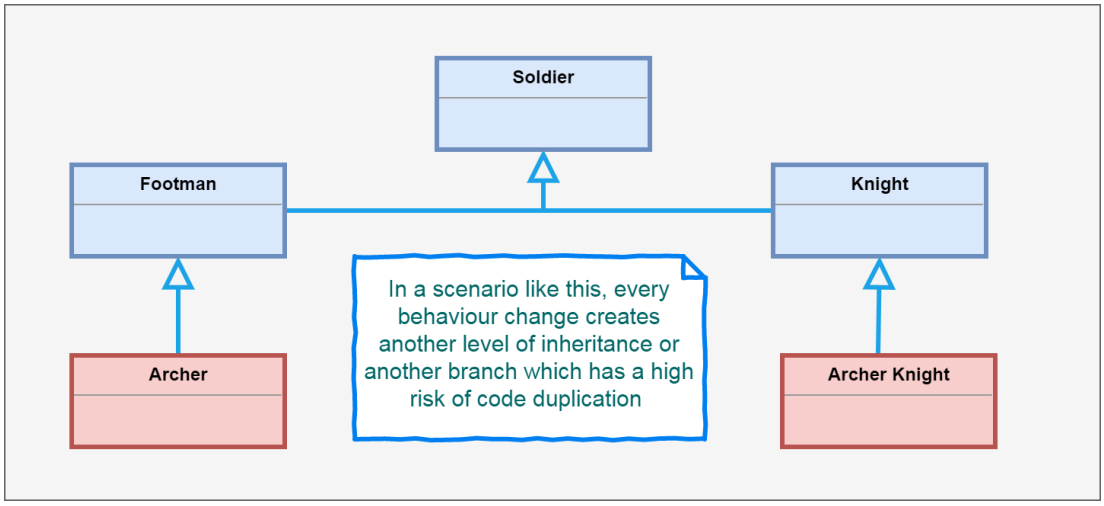

Figure 3.6: Inheritance example

When it is time to define a variety of objects which share various capabilities, inheritance will not allow selecting the parts which might be reused. Please see the example in Figure 3.6

The component-based version of the sample case explains how code duplication is prevented. Whatever feature required can be easily added to any game object. This design decision would keep the classes clean and shorter. Besides, this design will stay more robust when the code base becomes large.

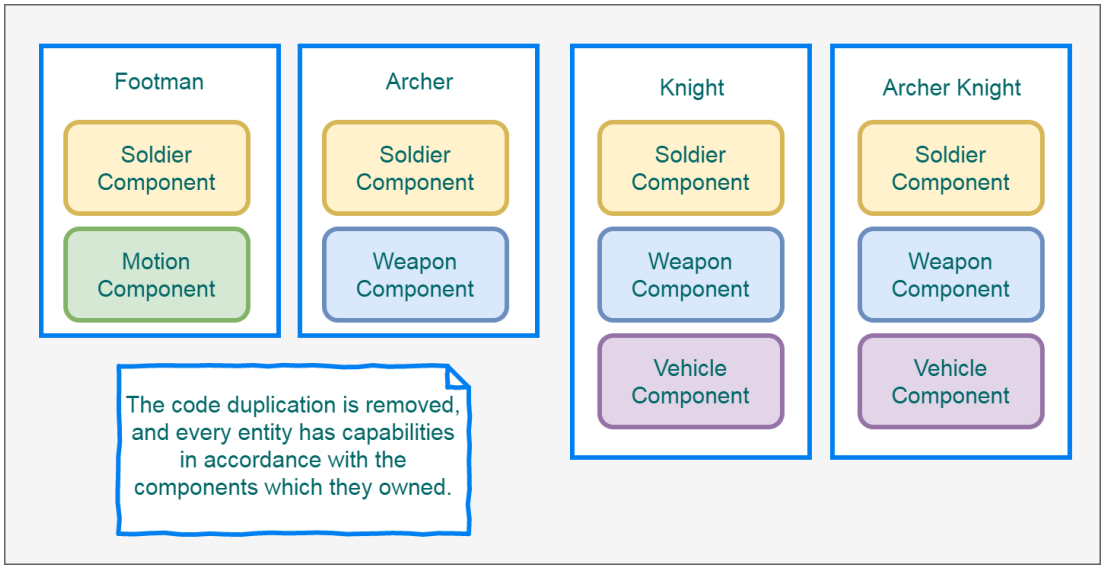

Figure 3.7: Component Pattern example

The component pattern is a very efficient one to manage feature based game objects as it is in Figure 3.7. However, it is worth to mention that game objects will contain a component cluster when this pattern is used. Therefore, an extra code should be written for the interaction among these components and managing their initializations.

In the previous section, MonoBehavior class shows how unity handles the Component Pattern. During the design of the architecture for the roguelike framework, the output should be designed with support for Unity's component pattern to have a better compatibility with the platform.

3.4 Unity Serialization Mechanism

Unity Editor is run based on its serialization engine. Serialization is an automatized process of converting data structures or object states into a form which Unity can persist and reconstruct based on the saved data (see Troelsen [2012]). A built-in serialization usage is provided by the Unity itself which is triggered for reloading the code and inspector window.

The way the developers organize the data can affect the performance of the project. It has to be known the serialization mechanism of the Unity before starting any project.

There are mainly two points which are significant for almost every editor extension.

- Hot code reloading: This serialization is triggered by a modification of the scripts. Whenever Unity is active, it watches the changes on the files. Even a little change triggers the hot reloading. It is a process to see an immediate effect of the scripts. The user does not have to press a button or restart the editor. The code will be auto-deployed and compiled out of the box. While this happens, unity backups all the serializable variables in the script data which have been already initialized. After reloading the code, it restores all the variables back. This process is vital and should be taken care of delicately. Otherwise, the data which has not been serializable will be lost after the hot reload.

- Preview functionality: When the user wants to preview the game, the state of the project is serialized. Whenever the user presses the stop button, all progress will be lost, and the serialized data would be loaded back. All game object states are reconstructed, and this will have a similar influence like hot code reloading. The delicate data should always be serializable.

It is worth to state that the inspection window uses serialization often as well. Nevertheless, this does not have a significant impact on the framework in the scope of the thesis.

What is Serializable: Public variables or variable with the SerializeField attribute are serializable if they are not sophisticated custom types (see Public Relations Report [2017]). Static, const and readonly variables will never be serialized. Custom non-abstract, non-generic classes or a class which extends an abstract or generic class are also serializable when the Serializable attribute is used. Primitive data types, enums, and lists are serializable. However, serialization works until the second level. Data types like dictionaries, list of lists, two or multiple dimensional arrays are not supported. Several solutions are available when unity cannot serialize the intended data. One is implementing the serialization interface by providing methods for serialization and deserialization. Another solution is wrapping the data types into a class. For instance, list of lists cannot be serialized. However, if a class wraps a list, the list of this class will be serializable.

3.5 Plug-in Development for Unity

Unity Asset Marketplace is one of the biggest digital markets and used by thousands of users every month. Accordingly, the engine provides a broad variety of support to extend the engine. Almost everything is possible with low and high-level GUI functions.

Unity API is separated as Unity Engine API and Unity Editor API. The latter supports the editor extensions which also contains the methods to create custom windows. Placing the GUI elements on this custom windows and handling the events are challenging since debugging the editor code is not easy. Nevertheless, it is possible and thousands of developers, game studios are developing domain-specific assets to make an income. In Unity, the scripts create the customized behavior, but it is also possible to include code from outside in the structure of Plugin. There are two kinds of plugins in Unity: Managed plugins and Native plugins.

- Managed plugins: These are dynamically linked libraries (DLL) created with tools like Visual Studio. These DLLs can be included in the Unity easily. DLLs are limited to the capability of the .NET code, and this code is closed. Developers cannot see the content of the library. However, the code is accessible to the Unity and the scripts.

- Native plugins: Native plugins are platform-specific libraries written in native languages like C and C++. This plugin is out of the scope of the thesis.

The roguelike framework is a plug-in which is planned to be open sourced. Therefore the development will be done through the scripts using the C# language. Unity allows creating a custom window by extending the EditorWindow. It has a similar lifecycle as MonoBehaviour has with some changes and these are not documented well in the Unity documentation. However, even knowing when Unity calls OnEnable() method can be a genuine helper. Since the rest of the extension live inside this lifecycle as it is described in Figure 3.8, these methods

should be handled by the developer. It is also worthy to mention that OnGUI() method should contain minimum logic to prevent Unity to behave unexpectedly.

Editor extensions might require data persistence, but game objects are useless on the editor side. However, Unity has a decent solution for this problem; Scriptable Objects (see Public Relations Report [2017]). These objects are used to keep persistable data, and they work independently from a game object. The ScriptableObject class is inherited from the Unity's root Object class, and it can not be constructed with the new keyword. The CreateInstance method provides an instance from the class, and this instance is efficiently serializable to a data file. The serialization rules which explained in the previous section is also valid for Scriptable Objects.

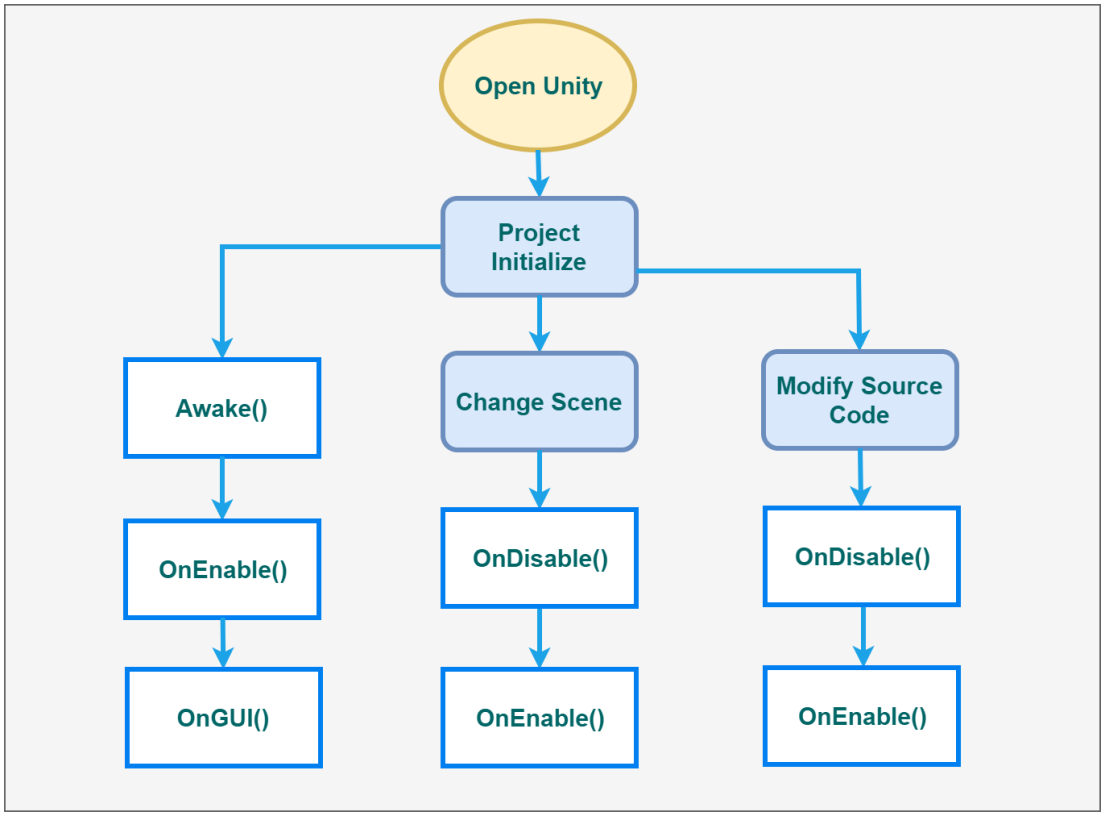

Figure 3.8: Lifecycle for the EditorWindow

3.6 Feasibility of the Project

Unity provides many exciting features and broad support for plugin development. A few key points can be listed as follows;

- Scriptable Object is the best candidate to use for nodes and the node graph.
- The classes that will be inherited from Scriptable Object should be ensured for Unity's serialization.

- A custom dedicated window should be used for the canvas. Graph and the inner sections will be drawn to this canvas.
- Unity's reflection can extract the nodes from the assemblies.
- The interpreter first generates the code, then generates the data-files, finally decorate the scene by attaching the generated code to the game objects.
- Custom serialization methods should be provided since the project requirements are further than primitives. Serializable list of lists and dictionaries should be handled in the scope of the thesis.
- One should mention that the planned project is feasible to build top of the Unity3D game engine. The architecture will be supportive for it. However, build process should be divided into two: Data generation and scene decoration. Because during the data generation new code will be generated. Since this code is generated in a separate assembly, reaching it from the Editor assembly is not possible before Unity applies hot code reloading. Once the Unity deployed the code and generated the meta files, this script will be attachable to the game objects freely. Therefore, the game build function should be handled in two stages.

4. Proposed Framework Architecture

Roguelike Framework (RLF) is a tool which works on top of the Unity3D game engine and aims to help roguelike developers with its node-based visual editor. Easy to use interface gives a collective representation the roguelike for the different roles such as game designer, developer and graphic designer.

The following chapter initially introduces the high-level architecture of Unity, and the rest of the chapter is divided into modules. Every module has the same structure, and they all start with the inter-module evaluation. Then a class design is provided, and the workflow of the module is described in the Running Pipeline sections. Finally, there is a discussion on alternative design choices at the end of every module.

4.1 High Level Architecture

RLF is designed abstractly, and abstract parts are extended by following the scope of the thesis. However, the target of the design is also allowing the tool to have the possibility of adaptation for other domains such as shader editing or behavior trees.

RLF is combination of four different modules;

- Visual Editor
- Node Graph API
- Interpreter
- Roguelike Library and Unity-specific game objects

The first three of them work within the Unity Editor and aids the generation of the game. The last one is accompanied by the generated game to provide services like dungeon generation. Please see Figure 4.1 to have broader view.

Whereas the visual editor works as a user interface and takes commands from the user, the interpreter evaluates the node graph and generates project files based on the graph. Node Graph API contains the actual graph definition and several nodes definitions. The nodes in the API do not give a logical description of them. This role is owned by the interpreter in order to separate the logic from the entity definition to keep the module clean.

Text files are used for data persistence since the Unity works the best with its asset files as a data container. Since asset files use a type of binary serialization, the memory usage is considerably small.

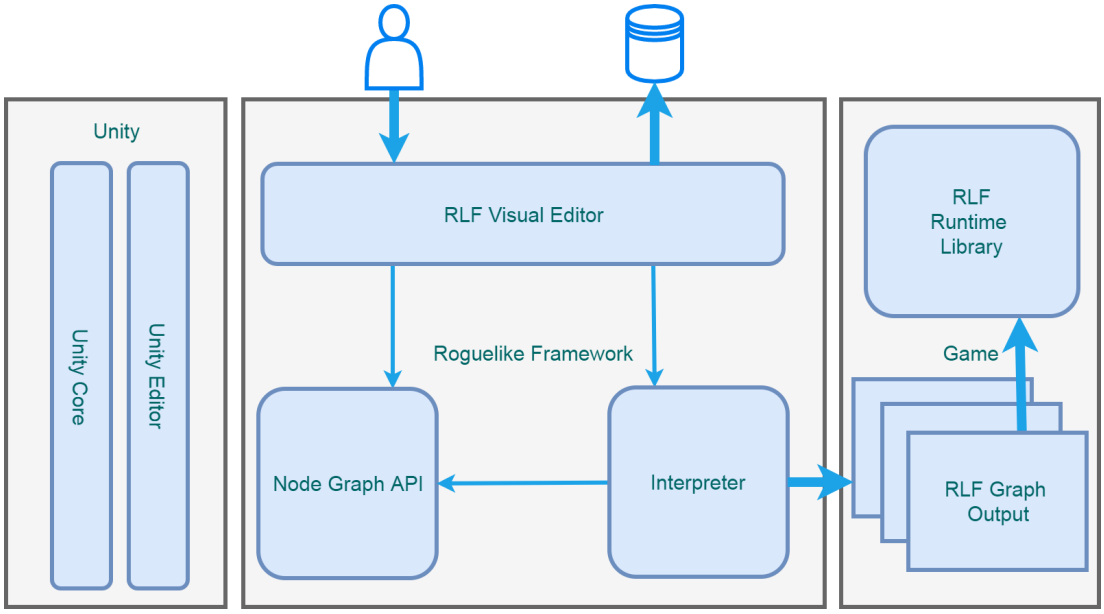

Figure 4.1: High-Level Architecture of the RLF

4.2 Visual Node Editor Module

The programming via a visual interface for a specific domain is the primary idea of the thesis. Visual Node Editor gives to the user a controlling mechanism to build a tree structure. Therefore it has access to both Node Graph API and Interpreter modules.

Primary responsibilities of the module;

- Accepts inputs from the user
- Manages the data persistence
- Wraps all the available nodes using reflection
- Supports node creation via drag and drop method
- Encapsulates the nodes in a context-aware form
- Shows immediate documentation
- Provides a menu for functionalities such as creating a new game or interpreting the graph
- View-based design

As it is emphasized earlier, this module should work as a controlling layer and interface. The logical implementation should cover only the user interface functionalities. It it evident that the link between this module and the other modules exist, but we tried to keep the coupling as low as possible.

The primary reason for this decoupling is keeping the intercommunication between modules clean and maintainable. The secondary benefit which is acquired with this decision is the freedom to use this structure for creating another

domain-specific visual programming environment. Even though this work covers only roguelike games, having an abstract editor is valuable in many ways in the aspect of extending the tool independently.

4.2.1 Class Design

As it is discussed in the previous, the framework works on a dedicated window which extends EditorWindow class. OnGUI() method of the class is used to make the module works with an event-based approach.

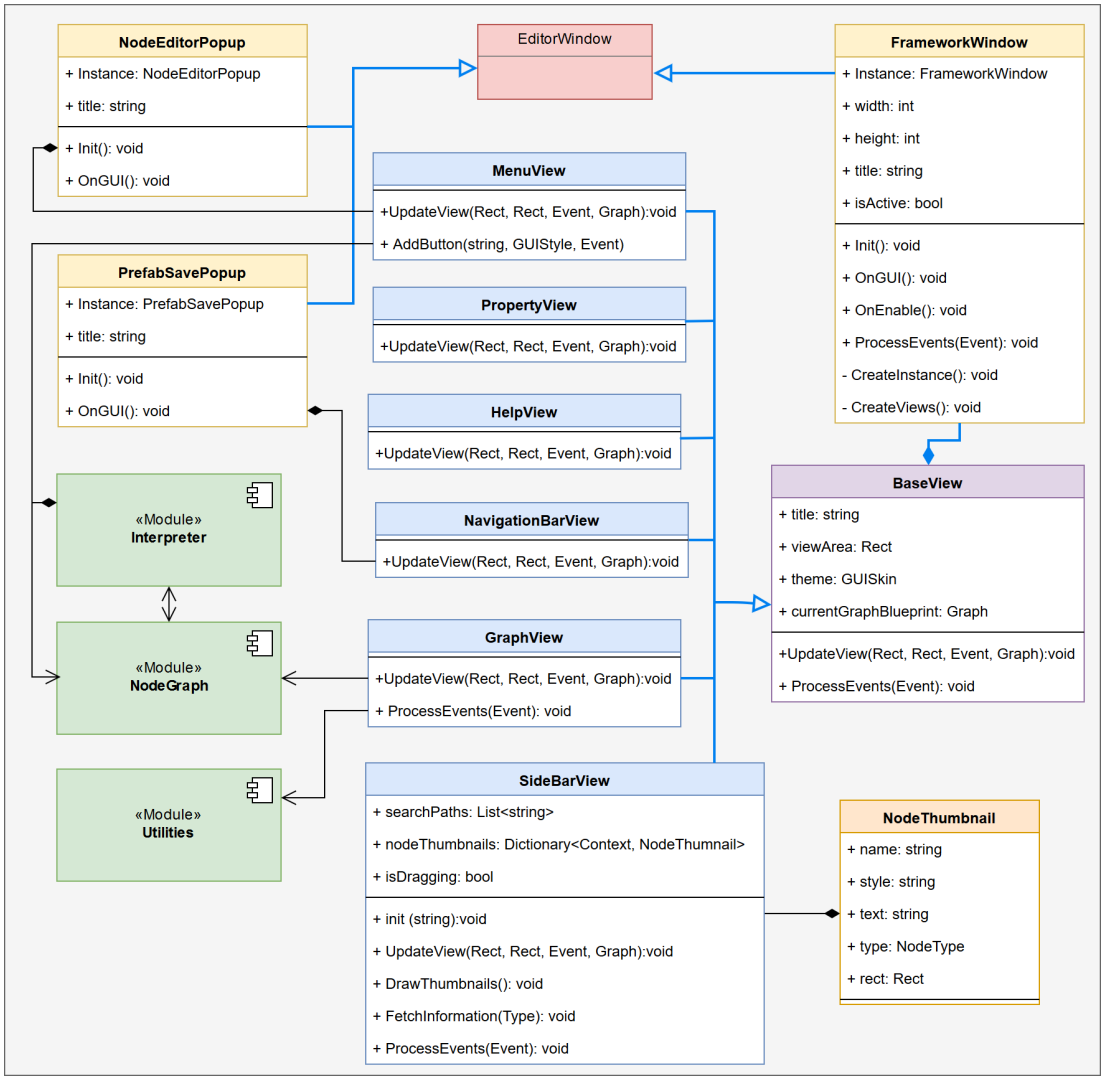

Figure 4.2: Abstract Node Editor design

The class design is given in Figure 4.2.The editor is designed as a view-based layout, separately windowed user interface. Each view has a different responsibility and logic. Accordingly, every input taken from the user should be distributed to all these views and evaluation of the event is done independently. However, views might also share common logic such as drag and drop action among the views.

This user interface is used to control the graph which has a tree structure. The responsibilities of the views are given as follows.

- Menu View: It contains a menu which has high-level functionalities such as creating a new game, interpreting the whole graph. It is the only view that interacts with the Interpreter module.
- Navigation Bar View: It is a dynamic menu which also has a high-level functionality that helps to navigate between contexts of the graph. It only manipulates the tree stack.
- Sidebar View: It is a dynamic node container. During the initialization, it loads all available nodes using reflection. Therefore, the coupling is quite low. It also evaluates the node dragging event from the user.
- Property View: The sole purpose of this view is showing the properties of the node when a single node is selected.
- Help View: Every node has a special meaning and usage. Help view shows the documentation of a node when a single node is selected.
- Graph View: It is the central area of the framework. If one looked at it on the interface, it would seem that all the complicated jobs are issued in this view. However, its sole purpose is transmitting the event to the current graph. Any manipulations in this class would increase the coupling. This design decision is taken to decrease the complexity.

As a remark, this part of the module initially designed with the Abstract Factory pattern (see Abstract Factory Pattern in Gamma et al. [2016]) in order to give a better abstraction to support different window layouts. However, this design decision is given up to reduce the complexity. Accordingly, Facade Pattern is transformed into Delegation Pattern. The usage of the pattern in the architecture made the actions much simpler per view.

Singleton Pattern (see Singleton Pattern in Gamma et al. [2016]) can be quite controversial to use, however in the scenario above requires one instance of a window. Due to the design of the Unity, every component draws itself within the OnGUI() method. If there would be two instances of a window, it will be drawn twice by the Unity Editor runtime. Hence, it was a necessary decision to take, and this pattern is also used for the dialog windows.

4.2.2 Running Pipeline

The user interface reacts based on the Unity Event class which caught the user actions and allow to retrieve it. Instead of evaluating the action in one controller or facade, the event is delegated to all the views and let them evaluate independently (see Delegation Design Pattern in Gamma et al. [2016]).

The working pipeline of the editor significantly affects the whole framework's running principle. Let's give a use case scenario to show how the interaction happens, and event delegation is applied.

- Scenario 1 - A user clicks on the build button: A click is made on the mouse and this event caught by Unity Editor. After that, it calls OnGUI method of the FrameworkWindow class. Inside the OnGUI method, the event is retrieved and distributed to MenuView. It first reacts to the event as visually then send the mouse position to the set of buttons and checks if any of them is clicked. Build button fires based on the mouse position and this whole cycle cause a building call in the interpreter.

- Scenario 2 - A user clicks on a node: This is different than the first scenario since the target is an object outside of the module. In this case, the event will be delivered to the GraphView by the FrameworkWindow object. The GraphView doesn't even check the event, delegates to the graph and lets it handle the action.

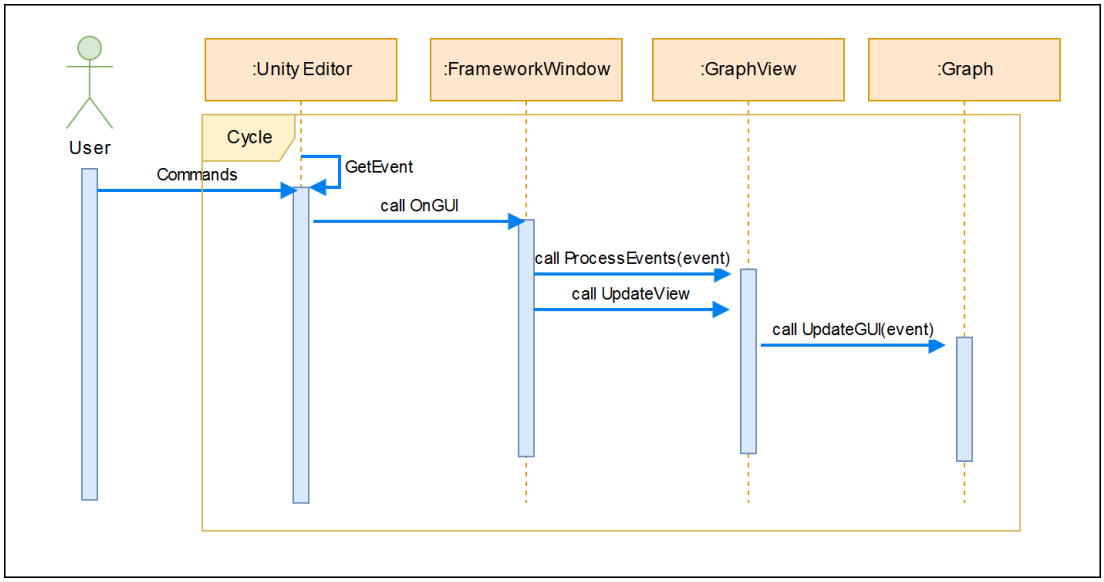

Figure 4.3: Sequence Diagram for the Delegation Pattern

The sequence diagram in the Figure 4.3 tries to illustrate how the chain of interaction happens for the events that target the node graph. It is also worth to mention that Unity applies some automatic actions when it is triggered such as hot code reloading. (see Section 3.4) These actions cause rebuilding the framework window again. Depending on the computer, this process can be 5-6 seconds, and during this period the framework became unresponsive.

4.2.3 Extensibility

If a design can keep its principles against changes, then it would enhance the development lifecycle and be more consistent. Furthermore, if a design allows extensions without breaking the structure, then it will be a future investment. Therefore, every module will be discussed under the extensibility sections.

The primary principle of the visual editor is keeping it as abstract as possible, evaluating the interface related events in the module and delegate the rest to

the dedicated modules. The design should be flexible with the protection of its principles. A couple of extension points of the visual editor are presented below.

Introducing a new view: The views know the current graph which held in the editor since they have a reference to it. Hence they can extract the public information or retrieve the library features via reflection.

Simple views which only shows information regarding the graph or a node is easy as extending the BaseView and adding the fresh view to the framework window. If the view contains sophisticated functionalities, then it should be implemented inside the view definition. However, this event may require reaching some knowledge from another module. In this case, it is best to delegate the event to the other module's related class.

Handling new nodes: Whenever a developer creates a new node, it appears on the user interface as well. This process is done by reflection and shown in the sidebar view in a way which is context-aware. Accordingly, the developer does not have to handle anything in the editor.

Overriding the existed behavior of the views: The default functionality of the views is overridable since the structure has a degree of support for Open-Closed principle (see Meyer [1994]). One can extend any of the existed views to add extra functionality or modify the subsisted behavior. In case of using dependency injection tool, according to modification can be applied through dependency modules (see Schwarz et al. [2012]). Otherwise, the developer should replace the view in the FrameworkWindow class.

4.2.4 Discussion

The design given above is adequate to manage the framework. However possible changes might do the framework even more abstracted than presented. Some design decisions are not included in the proposed architecture because it would add an extra complexity layer which is unnecessary as explained further.

The alternative design introduces two design patterns (see Factory Design Pattern and Controller Design Pattern in Gamma et al. [2016]) and requires extra classes such as Layout, LayoutFactory, and ViewController. Please see Figure 4.4. FrameworkWindow keeps the views and manages them in the current design. However, A view controller initialized with a layout can be created inside Framework window and shift its responsibility to the ViewController class to support the different layouts.

This design makes sense when it is used within a dependency injection framework since the new layout can be composed of overridden versions of the views. However, this might be an over-design since the probability of having a change in this module is not remarkably high.

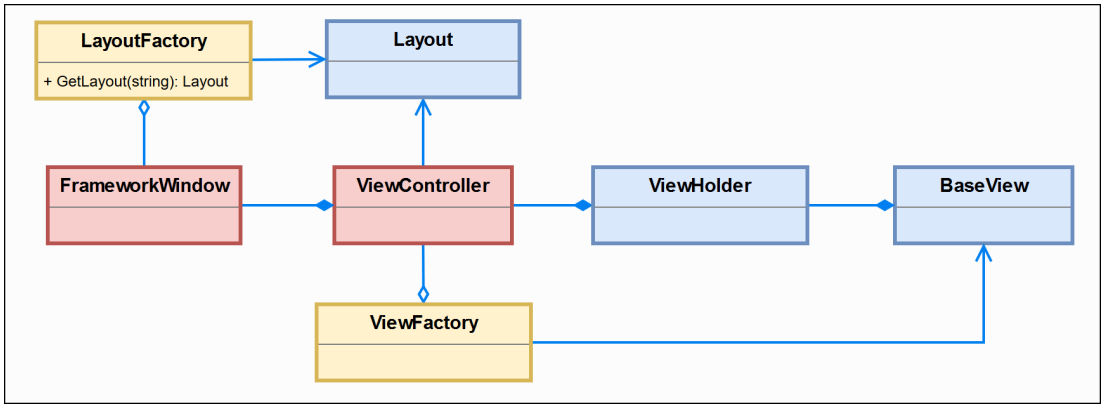

Figure 4.4: Alternative design to support multi layouts

4.3 Node Graph Module

Node Graph Module contains the building blocks of the framework with adequate abstraction without containing logic for the game. The visual editor which is introduced in the previous section interconnected with the Node Graph Module. The commands transmitted from the Unity Editor, filtered and delegated through the framework's interface. Then the graph handles all those filtered events manipulating the nodes and their connections.

Primary responsibilities of the module;

- Accepts filtered event from the user interface
- Has easy to create nodes which have unique ids.
- Allows node linking
- Support type checking
- Has reusable subtrees
- Provides iterable data structure for the graph
- Wraps serializable data
- Support for large workspace which might be even more substantial than the view where the graph is contained.

This module is the content part of the whole framework. It does not contain more logic than it is required. Even though it has some module-specific functions like type checking, there is no game related implementation. The link between this module and the others is reasonably coupled in the scope of the thesis.

If one changes the whole visual editor with another one, there won't be any changes in this module. However, changing one thing in this module would cause to change the Interpreter module as well. It was evident that decoupling this module would be over-design since it would require another module to implement the relation between this module and interpreter. This extra bit of complexity is not necessary since this framework will only support one type of domain.

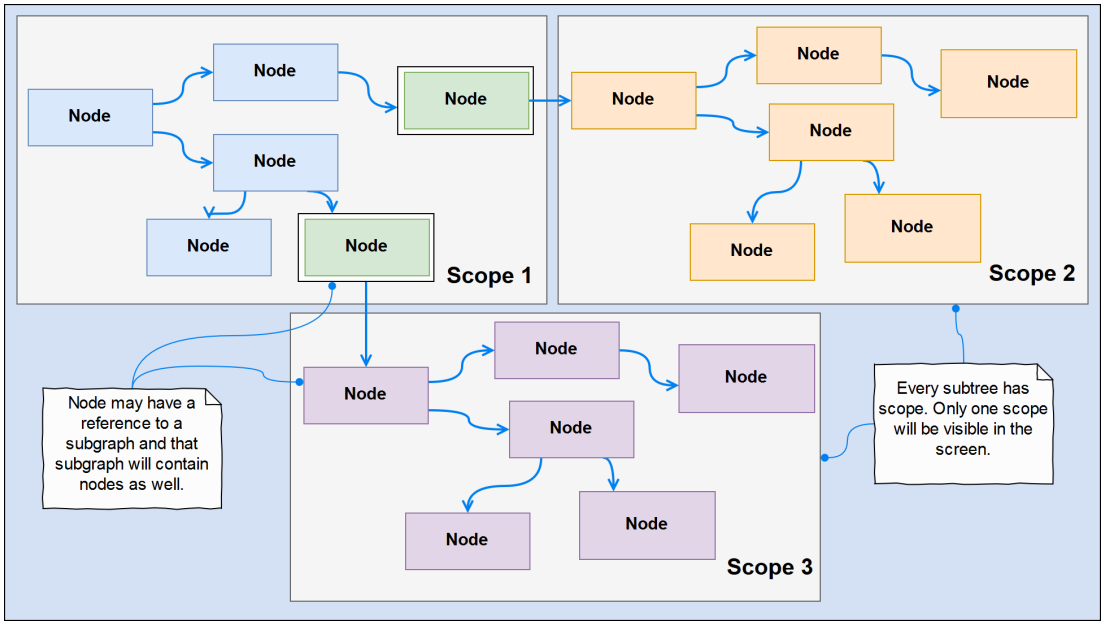

Figure 4.5: Planned structure for the Graph

4.3.1 Class Design

Complying with the responsibilities defined in the previous section requires a hierarchical data type for the graph. Supporting the subtrees is also crucial for the user experience since users may reuse some graphs. For instance, a designed level should be reusable since many levels may look like similar with some differentiation. Doing this requires more than one graph object. An illustration is given above to have a clear picture of this data type and discuss the possible patterns to comply with the structure.

As it can be seen from the Figure 4.5, the graph is precisely a tree data type, however, presenting one big tree is not realistic for the user experience aspect. Therefore scopes are introduced in the structure which is actually a subtree with contextual nodes. There could be only one scope in the interface, and the user can move in between scopes. The green nodes in the first scope are double-clickable, and they are the scope changing points. In the background, there should be still a big tree, since all the tree is a whole of a game logic.

One of the requirements of this module is reusability. Accordingly, the framework should allow users to save a subgraph and load them in another game. It is evident that scopes have their context, hence having a Level Design scope in a Story context is not appropriate. Consequently, the design is in the direction of having a graph contains nodes which belongs to a specific scope.

A node is a logical component, and it is the most sophisticated entity in this module since it has many variations and functionalities such as linking it to another node. A primitive type of node may appear many times in the graph. Therefore, every node has a unique id to separate them from each other. It is softly mentioned that this part will not contain logic regarding the game. However, the nodes can hold information which can be used in the game. For

instance, a number node can contain an integer, and this can be used for the width of the game canvas.

It is also deserving to mention that graph and nodes should be serializable within the rules given in Section 3.4. Therefore they will extend ScriptableObject class of Unity, and due to reusability requirement, these objects should be Clonable as well. The meaning of clone here a bit different than the generic usage of it, because for every cloned node there should be a unique id as well and all the references to this id should be updated accordingly.

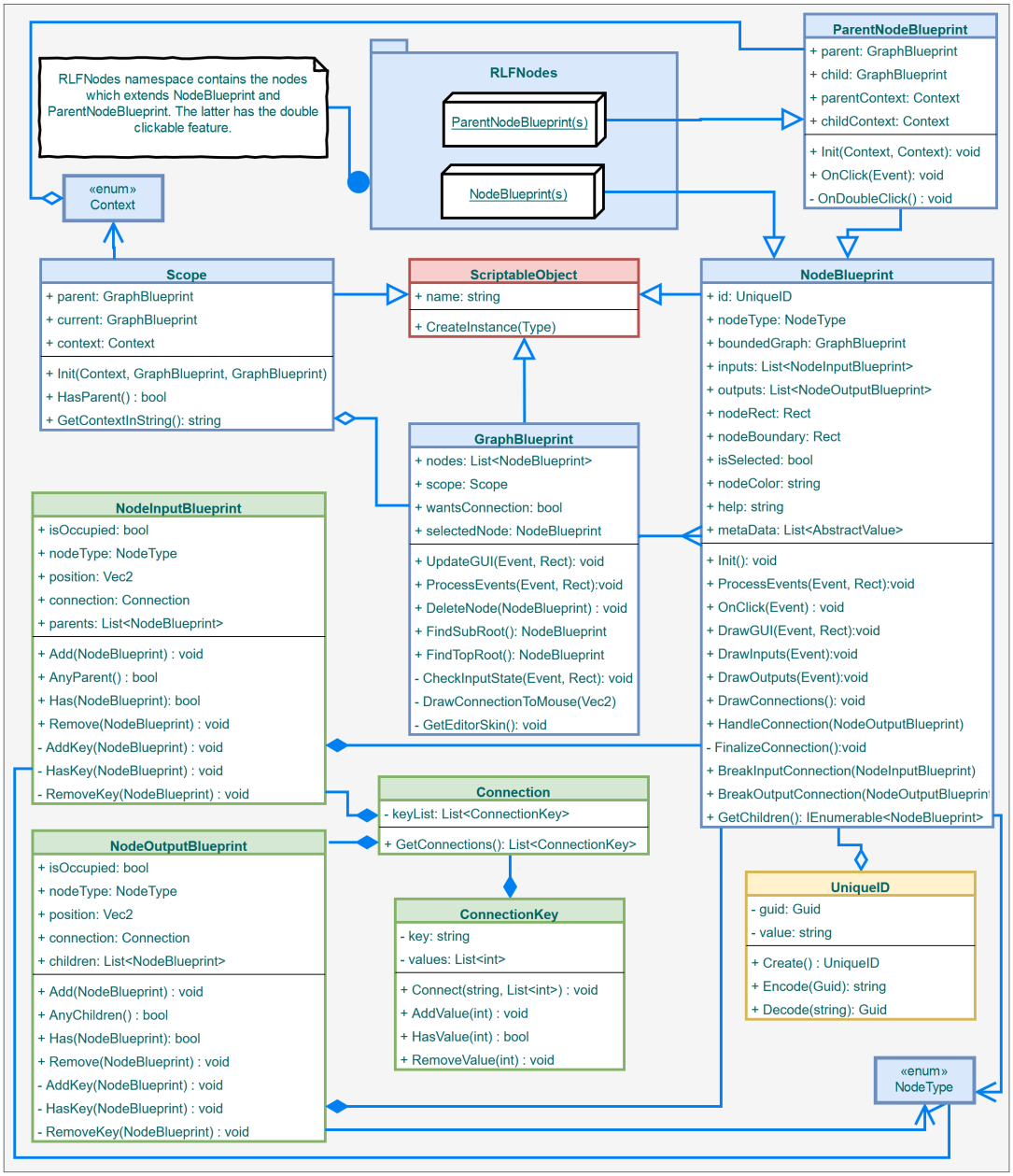

Figure 4.6: Planned structure for the Graph

Planned design is given in Figure 4.6, and it can be seen that the graph here behaves like a controller. There are two critical methods in this class; ProcessEv-

ents and UpdateGUI. These methods with the draw methods in NodeBlueprint class handle all the events which come from the user interface as part of the delegation pattern.

ParentNodeBlueprint class is a clickable node to support the link between node and subtree. When the event is evaluated in the GraphBlueprint, if it catches a mouse click then the callback method is called. If it is a double click, then the scopes are replaced to show the subtree. In case of having a null subtree, a new subtree is created based on the context. It contains a reference to the parent and child graphs.

NodeBlueprint has essential methods which aids to create or remove a link to other nodes. The links are always from the output to input and connection is done by the Connection class and secured by ConnectionKey. Since one output can be connected to multiple inputs, connection key contains a list of values.

The design also supports the enumeration which is necessary to traverse the tree. RLFNodes namespace is read by reflection and shown in the sidebar view as a context-aware since every node has one or multiple contexts. If one can read the requirements of the module, it would be obvious that this design is adequate to make the roguelike framework happen.

4.3.2 Running Pipeline

It is already mentioned that this module is the content part of the framework. Even though the user interface handles some of the user events, all the graph related events are delegated to this module. Type checking is also another logical part of this module to help the user to understand the structure intuitively.

4.3.2.1 Type Checking

If one looks at the design, it will appear that there is no method to handle the types in the NodeBlueprint class. The type checking is done by a separate utility class named NodeTypeUtils in the utility library. A sample snippet is given in Figure 4.7.

This decision is taken to manage the node types in a more significant sense. Having the node types as an enumeration is particularly convenient. Otherwise, there would be many classes to represent the types which might cause lack of maintainability. Unfortunately, enum implementation in C# is still so incapable of binding methods to enumerations as Java does. Therefore supplementary dedicated utilities are created for the type checking mechanism based on the result of this status.

NodeTypeUtils is designed as a rule-based checker. In the snippet, there is a Dictionary which takes a NodeType as a key and the rule as a value, and there is a method called DefineRules() which fills this Dictionary up. The Rule class is a holder for the accepted NodeTypes and whenever there is a node-linking event this mechanism is used statically. The nodes are linked based on the rules.

If IsAccept() method were inside the NodeBlueprint, there would be ten changes in the classes in case of adding a new node which is accepted by ten

```
public static class NodeTypeUtils
{
   private static Dictionary<NodeType, Rule> rules;

   ...

   private static void DefineRules()
   {
      rules[NodeType.LEVEL_GENERATOR] = new
          Rule(NodeType.ONE_ROOM_GENERATOR,
          NodeType.MULTIPLE_ROOMS_GENERATOR);
   }

   ...
}
```

Figure 4.7: Rule definition for Level Generator node type

nodes. On the contrary, having a NodeTypeUtils only requires a modification in the DefineRules() method.

4.3.2.2 Event Handling

There are two main classes which respond to the events in this module: GraphBlueprint and NodeBlueprint. As it is said before the event is coming from the Visual Editor, it is filtered and sent to the graph. Accordingly, it sends only the events which occur in the graph view. Then if there is an event regarding the NodeBlueprint, graph delegates the event there such as double click for the ParentNodeBlueprint.

The events which are supported by the graph;

- Single node selection
- Drag a node
- Delete single node or multiple nodes
- Multiple node selections
- Drag multiple nodes
- Connect nodes
- Break nodes
- Drop a node from the sidebar view

These events are all caught after the filtering events from the user interface. The current design of the evaluation follows the same design with the views' event handling using the ProcessEvents and UpdateGUI methods.

4.3.3 Extensibility

This module's extensibility is quite significant since during the framework development the most of the changes would be expected in the Node Graph API. The design is created based on the essential needs to prevent over-design.

These the points which are expected to change the most;

Creating a new node: This process is touching to the other three modules. However, Node Editor module handles the change based on reflection. Therefore nothing is needed to be modified as a reflect the change. The Interpreted related changes will be explained later in the next section.

Adding a new node is easy as extending the NodeBlueprint class. There are several pieces of information which needs to be added. Here is a code snippet for a sample node in Figure 4.8.

```
public class OneRoomNodeBlueprint : NodeBlueprint
{
   private static GraphBlueprint.Context[] _context =
       {GraphBlueprint.Context.DESIGN};

   public override void Init()
   {
      base.Init();

      shownName = "One Room";
      nodeColor = "ORANGE";

      Inputs.Add(new NodeInputBlueprint(NodeType.ONE_ROOM_GENERATOR));
      nodeRect = new Rect(10, 10, 220, 60 + outputs.Count * 10);
      help = "This node creates a square shaped room.";
   }
}
```

Figure 4.8: Creating a new node with one input

Creating a new type of node: When a new type of node is required, creating it is more tricky than adding a new node which belongs to a type. This requirement appears when a new functionality is needed on the node such as ParentNodeBlueprint which changes the scope when a double click mouse event.

Let's assume that there is a need for a node which doesn't have any input or output. The way to accomplish this task is also extending the NodeBlueprint, and then all the nodes should extend this UnboundedNodeBlueprint class. It seems like just adding a new abstraction layer although this is not enough to complete the task. Since the sidebar view works via reflection, this new interface should be added to SidebarView class' Init method as well. Then the freshly created nodes will appear in the sidebar. One can see how ParentNodeBlueprint is added there and take it as an example to extend further.

Adding a new event: As it is discussed earlier, GraphBlueprint evaluates the events which are delegated through the user interface. This part is initially designed as a state machine. The class first detects the state based on the event and then applies the respective action. What if does the framework requires a new functionality such as multiple selections with Ctrl + Mouse Left Click?

First of all the state and action should be defined as an enumeration inside the GraphBlueprint class. Then this state's event requirement should be defined inside the CheckInputState() method. Then the event should be handled inside the ProcessEvents() method. Such an important task is accomplishable with minor modifications as it is supposed to be.

4.3.4 Discussion

The current design allows the optimal environment for the defined requirements. However, adding a bit of complexity in this module can make things more manageable. These extra functionalities are not included in the design, but they can be added alternatively.

Facade Pattern: In the current design, GraphBlueprint is the controller of the whole module. However, when the responsibilities of the module increases, keeping the Controller Pattern is not appropriate since it breaks the Single Responsibility Principle.

The alternative design only requires creating a GraphFacade class which hands over the event handling mechanism (see Facade Design Pattern in Gamma et al. [2016]). However, this requires some modification in the GraphBlueprint and NodeBlueprint since they cannot share everything with the facade. Managing this change as a refactoring is not an easy task since the modifications require even the node-linking. Event delegation and letting the object handle the event is the Unity's way to handle entities. Therefore in the current design, Facade is not chosen as a controlling mechanism.

Command Pattern: Command pattern is a practice wrapping and stacking the application actions (see Command Design Pattern in Gamma et al. [2016]). Whenever an event fires, a corresponding command is created, and the application handles this command. Every command is revertable and should have one unit of action. Consequently, if the editor would have an undo-redo functionality, this pattern would be advantageous.

This pattern is recessively declined during the design of the framework. There are two reasons for this. First one is that the current design is readily revertable to a version which uses this pattern. Accordingly, the design keeps things simple and avoids to create extra classes. The second reason is that there are some GUI components on the nodes from the Unity Editor. These are already applying the command pattern and works in the case of Ctrl + Z key combination to undo the last action. If the pattern is applied, Ctrl + Z command will undo twice in cases when there is confliction between Unity's Command pattern and the one which belongs to the framework. Changing the universal command for undoing could

be a solution for this problem. Otherwise, this bug will cause unfunctional cases which are avoided in the current design.

4.4 Interpreter Module

The interpreter module is the part where the graph is evaluated and transformed into a roguelike game. This module is tightly coupled with the Node Graph module since the logical meanings of the nodes are implemented in this module. However, this does not mean that the design is not adequately abstracted to extend or change the functionality of the module.

The visual node editor module knows the graph and the interpreter. Consequently, it acts as a controller in the bigger picture. It triggers the interpreter module by giving the necessary graph knowledge.

Primary responsibilities of the module;

- Visits every node of the graph
- Contains two phases: Compile and Build
- Handles the scene decoration
- Generates the code when it is necessary
- Attaches the scripts and confugres them
- Has data-based approach (e.g. level data as JSON)
- Allows adding various visitors to reach seperate goals

This module is the only part that communicates with the scene and assets. It is known that customizing the game via only data files is not possible. Therefore it is compulsory to generate code in case of adding an event or optional functionality. This module has several visitors that have different aims such as generating game objects, attaching scripts or collecting texture data from the nodes.

The interpreter structure has to be changed whenever a new node is added to the Node Graph module. Between these two modules, there is a contract to keep things in a frame since an incompatibility may have an undesired effect on the outcome. This interface is implemented by the Node Graph module, and accordingly one can see the immediate reaction intuitively to implement methods inside the nodes.

4.4.1 Class Design

As it is introduced earlier, this module has a tight connection with the Node Graph module to traverse all game tree and retrieve the required data. The interpreter comes as one module, but there are various visitors (see Visitor Pattern Gamma et al. [2016]) to reach different goals. A service-based architecture is appropriate to use to keep the visitors as abstract as possible. Please see the class design in Figure 4.9. Due to the architecture of the Unity, building a game project divided into two as interpreting the graph and building the game.

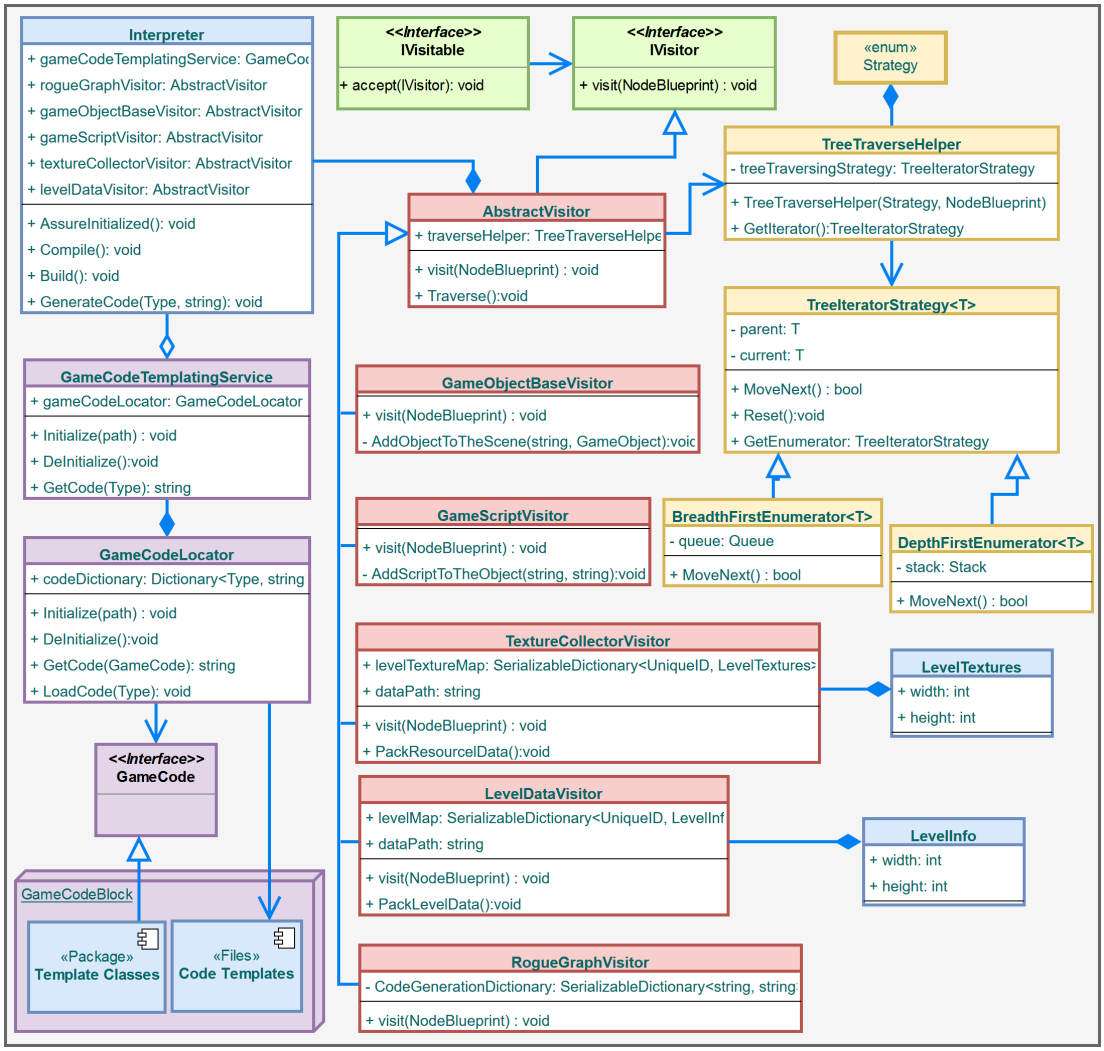

Figure 4.9: Interpreter Module class design

The interpreter module is the module which complies with the Unity Engine API. Therefore, it is fragile, and every update to the Unity can leave the module under threat concerning the compliance. Therefore, the design is handled carefully. The data needs to be interpreted is not low level like AST trees. Nevertheless, multiple visitors and big sized project are entirely possible. Therefore a discussion should be made to chose the main structure in the aspect of performance and maintainability.

In case of creating an evaluation or interpreter, there are two patterns appear as a solution.

- **Interpreter Pattern:** It is one of the well-known design patterns from Gamma et al. [2016]. The very basic idea to have a specialized language which contains terminals and non-terminal expressions. Every expression should correspond to a class, and they all have an interpret() method in it. These methods are called recursively.

- **Visitor Pattern:** This pattern is also one of the well-known patterns. The idea is separating the logic from the structure which the pattern works on.

The Gamma et al. [2016] defines the Visitor as *"Represent an operation to be performed on elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates."* The elements of the language have the freedom to accept a visitor or not.

Both of the patterns have some advantages, and however, two elements are surely crucial in this architecture. The one is increased maintainability, and the another is separating the logic of the Node Graph API. The visitor pattern fits into the requirements in a perfect way which allow managing the nodes outside of the module. Contrarily, the Interpreter pattern requires evaluating the nodes inside the node.

Hills et al. [2011] published a work to compare interpreter and visitor patterns. Interpreter pattern may be slightly faster in some cases. However, the visitor pattern has a better performance and maintainability results in the long run. The case studies given in the regarding article complies with the interpreter module in the scope of the thesis. Therefore the Visitor Pattern is used in order to evaluate the tree. Besides, the adaptation of the tree structure used in the Node Graph module is easy as extending the nodes with an IVisitable interface which can also be seen in the design of this module.

IVisitable has an accept() method which actually calls the visitor's visit() method. It does not matter to have more than one visitor. In the architecture there are some essential visitors are defined. However, more visitors can be added based on the new requirements. Even though every visitor adds a bit of complexity to the framework, it separates the logical parts and makes them maintainable in a better way. The duty of the visitors defined as follows.

- **RogueGraphVisitor:** This visitor carries a code generation map. It visits the nodes and decides which type of script should be generated with the related configuration of the node. After concluding the traversing all tree, the interpreter generates the codes based on the filled map by the visitor.
- **GameObjectBaseVisitor:** Some game object should be placed during the decoration of the scene such as Game Manager, Dungeon Generator. This visitor decides which object should be placed in the scene and removes the unnecessary ones.
- **GameScriptVisitor:** As it is stated earlier Unity has the component-based architecture. Scripts are accepted as a component, and they should be added to the game objects. This visitor attaches the scripts to the game objects.
- **TextureCollectorVisitor:** Image resources are a huge part of a game. This visitor collects all the image data from the nodes and saves into a .json file. This file is sourced in the Resources folder and during the runtime the images loaded from here during the level loading operation.
- **LevelDataVisitor:** This visitor visits the level related nodes and saves the information to a .json file. As a remark, all the level related data

cannot be stated as data. Some details like events are handled during the GameScriptVisitor traverse.

Top of all these visitors an abstraction layer is defined, and it contains visit method to all the nodes. Therefore whenever a new node is created, related visit method should be added first here virtually. Then if any visitor requires that node, it can contain the definition by overriding the abstract visit method.

Visitor pattern gives much freedom to the architecture. First of all, the NodeBlueprint does not have to know any logic. Secondly, the logic can be divided into several parts and these parts can be stored in different visitors.

Different visitors may require traversing the tree differently. As it can be recalled, the GraphBlueprint is enumerable. This feature significantly affects how visitor visits the nodes. A couple of tree traversing strategies are proposed by Morris [1979], and any of these strategies can be applied to the visitors since all the visitors are initialized by a traversal strategy.

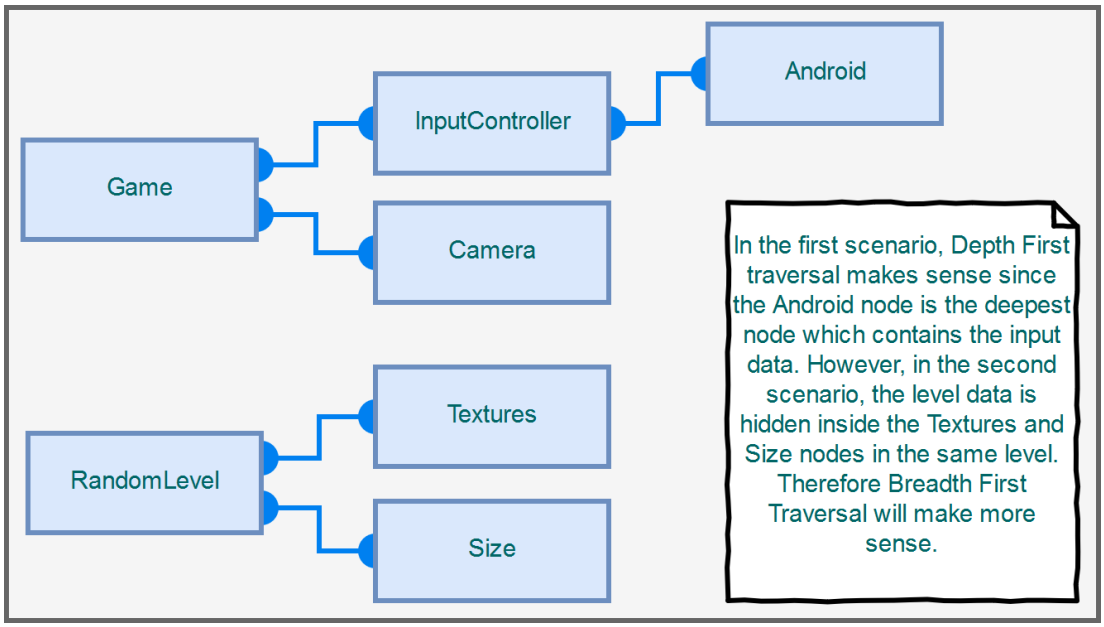

Figure 4.10: Comparison between Depth-First and Breadh-first traversals

Breadth-first Traversal is a level-order traversal, and it is used for all the visitors until now. This decision is given since the game tree structure requires it, but the design is open for other traversal techniques such as Depth-First Traversal. Strategy design pattern (see Gamma et al. [2016]) is used to give the adequate flexibility here. Different scenarios can be seen in Figure 4.10.

It is stated earlier that code generation is involved in the interpretation and RogueGraphVisitor object fills a map that contains the code templates with their configuration settings while traversing. The interpreter object could evaluate the map and generate the sources by itself, however, in case of having such requirements could inflate the class. In similar cases, a safe and maintainable solution is creating services which handle short life-cycled works. Therefore a dedicated

service is created for the code generation. The architecture also allows adding more services to this module.

An excellent candidate for a service is GameCode Templating Service because it also involves background works such as containing the code templates. This service has a sophisticated design which reads the templates by reflection. There is one class which correspond to every template and these all extend the marker interface GameCode. This interface is used to load templates into the memory with their types. Therefore whenever a query comes to the GameCodeTemplatingService with a type, it returns the related string template for the code which will be generated. Then the configurations are applied to the template with the help of a string substitution utility method.

4.4.2 Running Pipeline

The interpreter has two primary methods named compile() and build(). Building the game without the compilation is not possible. Compiling involves the level data creation, texture information extraction and code generation. Building decorates the scene by adding game object and attaching the newly generated code to those game objects. This separation is necessary since there is a hot code reloading process of the Unity during every change of the workspace. When some files are added, there must be a waiting time for the users. The transition between compile() and build() can be seen in Figure 4.11

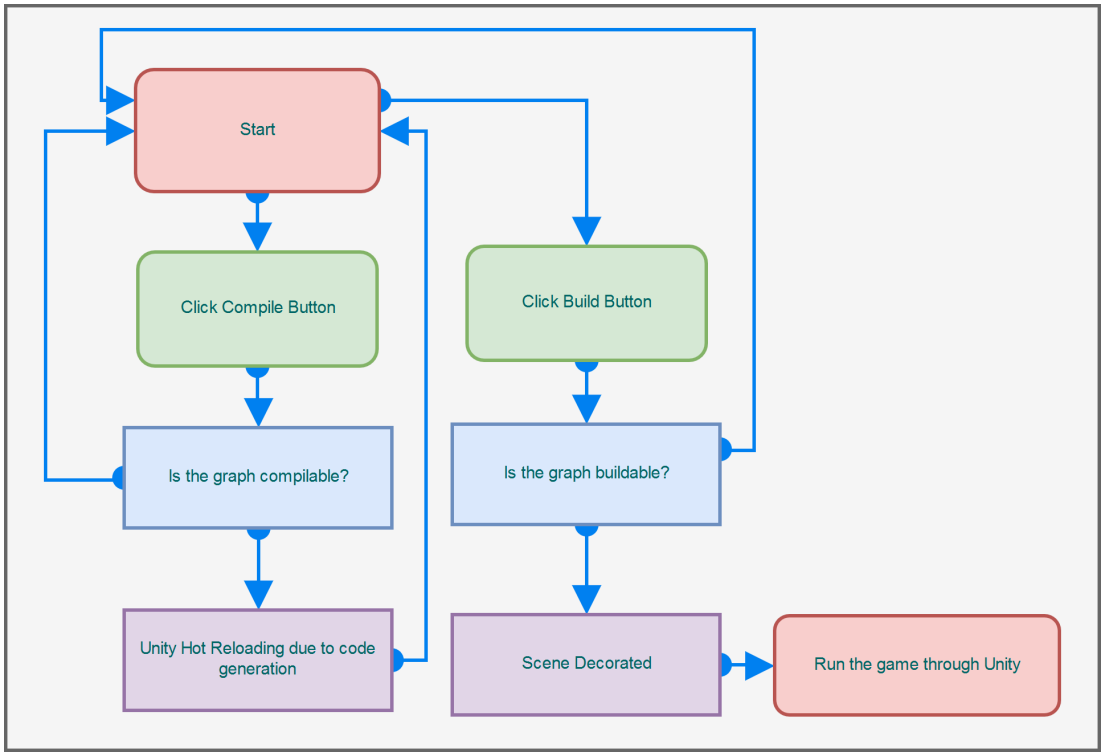

Figure 4.11: The relationship between compile() and build() methods

During the compile() and build() methods lifecycle, different visitors are used

however these visitors can depend on each other. If there is an error during a code generation, this code will not be attachable to any game object. This phenomenon can cause severe errors and most probably the game will not be run by Unity. Once the user fixes the conflicts, the game will be runnable again.

4.4.2.1 JSON File Generation

The state diagram shows the workflow on a higher level. However, low-level operations like JSON file generation are also essential. The file generation is done by the JSON service of the Unity with its wrapped interface for the JSON files. As it is stated before, Unity cannot serialize the list of lists, but there is no problem with the list of classes.

```
public static class JsonHelper
{
    public static T[] FromJson<T>(string json)
    {
        Wrapper<T> wrapper = JsonUtility.FromJson<Wrapper<T>>(json);
        return wrapper.Items;
    }

    public static string ToJson<T>(T[] array, bool prettyPrint)
    {
        Wrapper<T> wrapper = new Wrapper<T>();
        wrapper.Items = array;
        return JsonUtility.ToJson(wrapper, prettyPrint);
    }

    [Serializable]
    private class Wrapper<T>
    {
        public T[] Items;
    }
}
```

Figure 4.12: Wrapping technique to serialize list of lists

This helper class in the Figure 4.12 wraps a list inside a class as a container to overcome the serialization problem of the Unity. This approach worked quite well, and list of lists are serializable thanks to this method. Level data and resources are bundled into these JSON files and when the game initialized these data files are loaded into the memory.

4.4.2.2 Code Generation

Generating code is quite a sophisticated work, especially during the runtime. This project does not require runtime code generation since the methods work on the editor side before running the game. Design-time code generating code is not different from generating a text file since the code is a piece of string type

contained by a file. Accordingly, it is a process of creating dynamic strings and putting them into a file. There are very sophisticated code generation libraries, and what they do is having templates as a base to customize.

Bajovs et al. [2013] demonstrated all the state of art code generation techniques. Two of the methods are very significant for the roguelike framework. The first approach is the visitor based approach since the architecture is already using visitor pattern. A visitor can generate code while it traverses the code. The other approach is generating the code based on templates. Every template has a sort of abstraction and configurable blocks. In this architecture, a combination of these approaches is used as it can be seen in Figure 4.13. A visitor collects configuration data while it traverses then it delegates this information to the templates and the code is generated based on which data is retrieved.

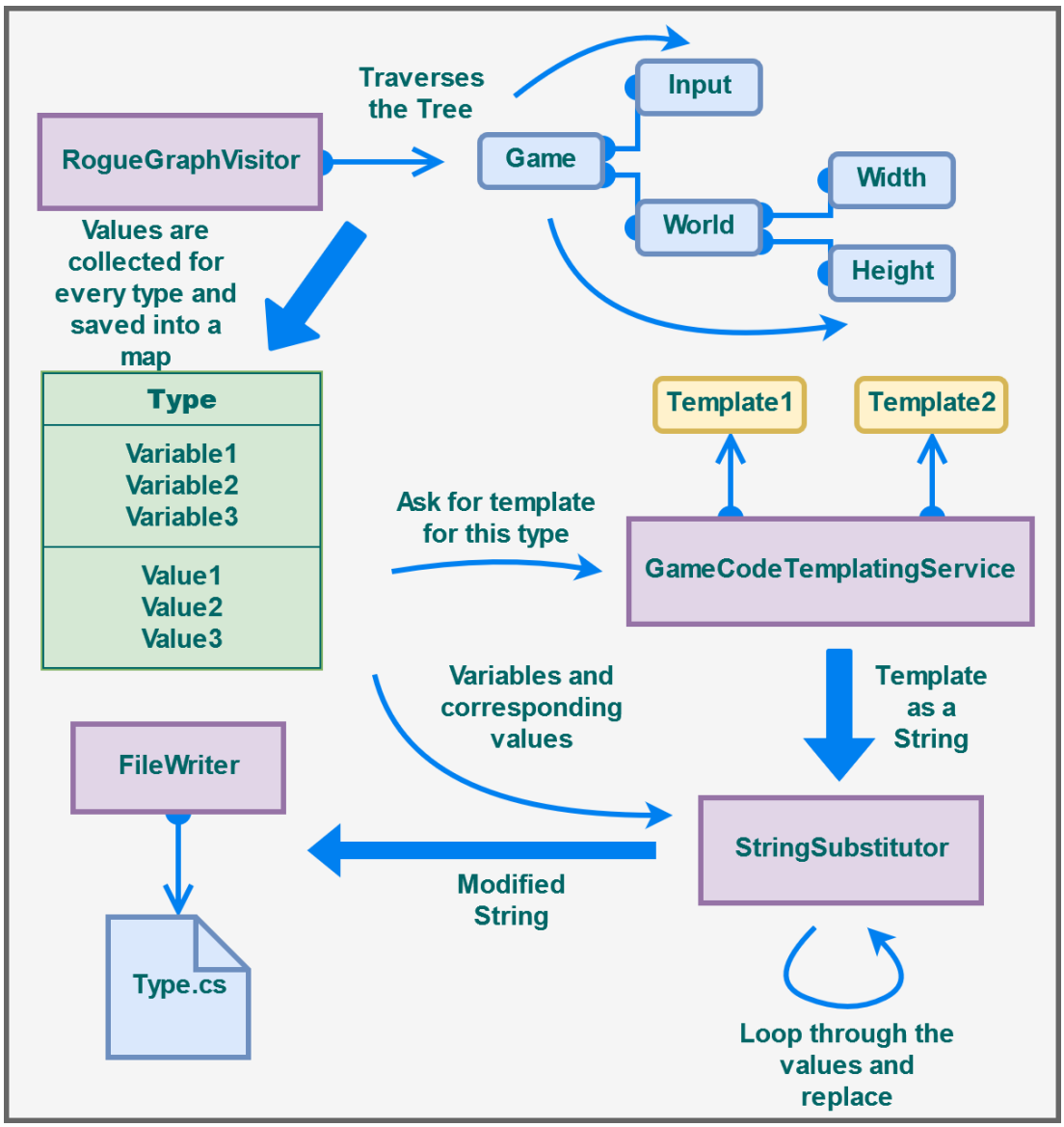

Figure 4.13: The pipeline for generating design time c# code

The templates are stored in separate text files to increase their maintainability.

The service loads them into memory whenever the service itself is initialized. There is a class which corresponds to every template file. Therefore, the service can be triggered by type to retrieve the related string template to manipulate. Finally manipulated strings are written into the .cs file and Unity automatically detects these data to compile.

4.4.3 Extensibility

The possibility of having new requirements in this module is possible, and there will be a few significant code change points will be described in this section.

Adding a new visitor: Currently, five different essential visitors are introduced to the architecture. These visitors are vital to making a working game. However, there could always be a need for a new visitor.

Let's assume that a new type of node set is introduced for networking support which requires data files for customization in the runtime. It is possible to use the RogueGraphVisitor to generate many network helper class but separating the logic for creation of the data files will provide a better maintainability in the later stages.

Creating a new visitor is easy a creating a new class which extends AbstractVisitor and implement the visit modes for to nodes which are required. Then, an object will be constructed in the Interpreter class, and this object should be registered to compile process or build process depending on the aim of the visitor. In this scenario is it the compile process since there is a file generation involved. Accordingly, these files should be handled during the runtime with the runtime libraries.

Adding a new service: The most prominent benefit of using service-based architecture (see Perrey and Lycett [2003]) is the ability to maintain the services independently or depending on another service. Therefore the number of services does not bring any lousy coupling mistakes and harm the overall design.

Besides, adding a new service is about wrapping all the service related class in the namespace of the service. In the three-tier architecture, there are service interface, logic and data layers. It does not have to follow this structure, but if the problem covers all three layers, then it is an excellent candidate for a service.

Let's follow the same example given in the previous section. A visitor is created to collect the networking configuration from the nodes. Once these data are collected, it should be persisted in XML form provided by the networking toolkit. It is possible to create a service named XMLNetworkingService and provide necessary interface to transmit the data required for the XML files. The service can use an XML utility class to modify the structure via data transfer objects (DTO). The service can be initialized and used from the Interpreter class.

Adding a new traverser: The architecture has the freedom to set a tree traverser to every visitor separately. It is stated that this part of the module uses the Strategy pattern. Consequently, it works with the same principle of a plugin. TreeTraverseHelper has the responsibility to return the desired traverse object.

Therefore it will not be wrong the state that it wraps and hides the strategies behind of this utility.

One can extend the TreeIteratorStrategy class and add the factory methods in the TreeTraverseHelper class. After configuring the Interpreter class with the new strategy, the new technique will be used to traverse the nodes.

Adding new code templates This requirement appears when adding a function with a data-based configuration is not possible. For example, the size of a maze is configurable and can be set using a text file, however, triggering an event is not configurable and should be handled using the code. In such a case, code generation is inevitable.

As it is stated before, code generation is done via templates. Every template corresponds to a type class which extends the GameCode marker interface. Small static information can be stored in this file to aid the process. Adding a new template is involved these two steps.

- Create a template text file which contains the code with configurable areas
- Add a new class to the project with the same of the template file. This class can contain optionally string variables for the code configurations.

4.4.4 Discussion

The current design is created based on the requirements. Nevertheless, this design is not the silver bullet. Alternative approaches and improvements can be done in a better development environment.

One more abstraction on visitors: If the design requires more visitors in the future versions of the project, having many visitors inside the Interpreter may be unnecessarily complex. In case of having complicated visitors, there can be added an abstraction layer.

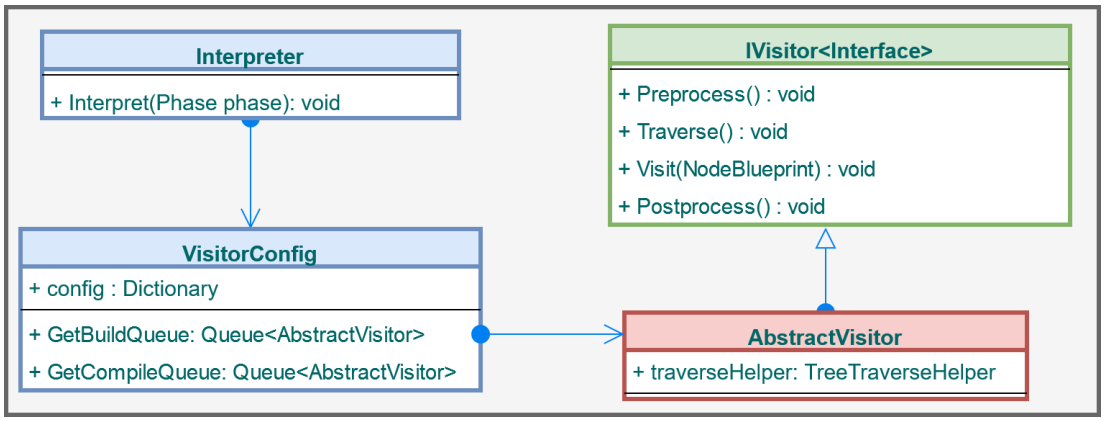

Figure 4.14: One more level abstraction on visitors

As it can be seen in the class diagram in Figure 4.14, there are visitor queues. In this design part, there are only compileVisitorQueue and buildVisitorQueue,

but the number of the queues can be increased based on the requirements. A new interface is introduced to depend on. Therefore the interpreter will call these methods without caring about the visitor and visitor will handle their post process logic by themselves.

Dirty checker: Modern compilers have a dirty checking mechanism that avoids the recompilation of a class or module. This approach can be used in this tool as well. There may be plenty of ways to accomplish this task.

One method can be creating a flag variable in the NodeBlueprint class. This variable is set to false as default. Whenever a node is flagged as dirty, visitors visit only this node and the subtree which takes this node as a root node. This option can be enhanced with different settings as well. Even if a node is not dirty, it can be compiled since it is not an optional node.

This option makes sense when there are more visitors with hundreds of nodes. Otherwise, it will not save a remarkable amount of time.

Smart visitor: As it is mentioned before, breadth-first walk or depth-first walk makes sense depending on the node context. Instead of traversing all the tree with one method, an optional traverse strategy can be defined per node.

The visitors can define their strategies based on the node properties. This type of traversing strategy may produce more robust interpretation skills. There is very strong bound between the traversing strategy and how the visitor interprets a node.

Moving services inside the visitor: Current design offers to have services in the Interpreter class. Because at first sight, duties like code generation seem like the responsibility of the Interpreter class. However, let's assume that there twenty services which do specific tasks. Persisting all these services in the Interpreter would be cumbersome.

A possible solution for this problem can be shifting the services inside the visitors. RogueGraphVisitor can carry the GameCodeTemplateService and handle the code generation by itself. The degree of maintainability can decrease due to the lack of knowledge in which visitors use what service. This problem can be smoothed out using a dependency injection tool and having well-planned module bindings.

4.5 Runtime Library and Utilities

This is the last part of the framework. The three modules mentioned above are quite essentials and while the other modules grow utility classes and runtime library also increase in size proportionally.

It is already mentioned earlier that Unity keeps two separate assemblies as one for runtime and one for the editor. All the modules explained above were part of the editor. However, the game requires some built-in functionality to comply with the output of the framework.

The game should be accepted as a different software, and it should also be designed in a maintainable way.

4.5.1 Class Design

Since this module had a dynamic structure, only a base class design is shown in Figure 4.15 to support the prototype. However, there is no limitation for this model, and it can be extended by the end user freely.

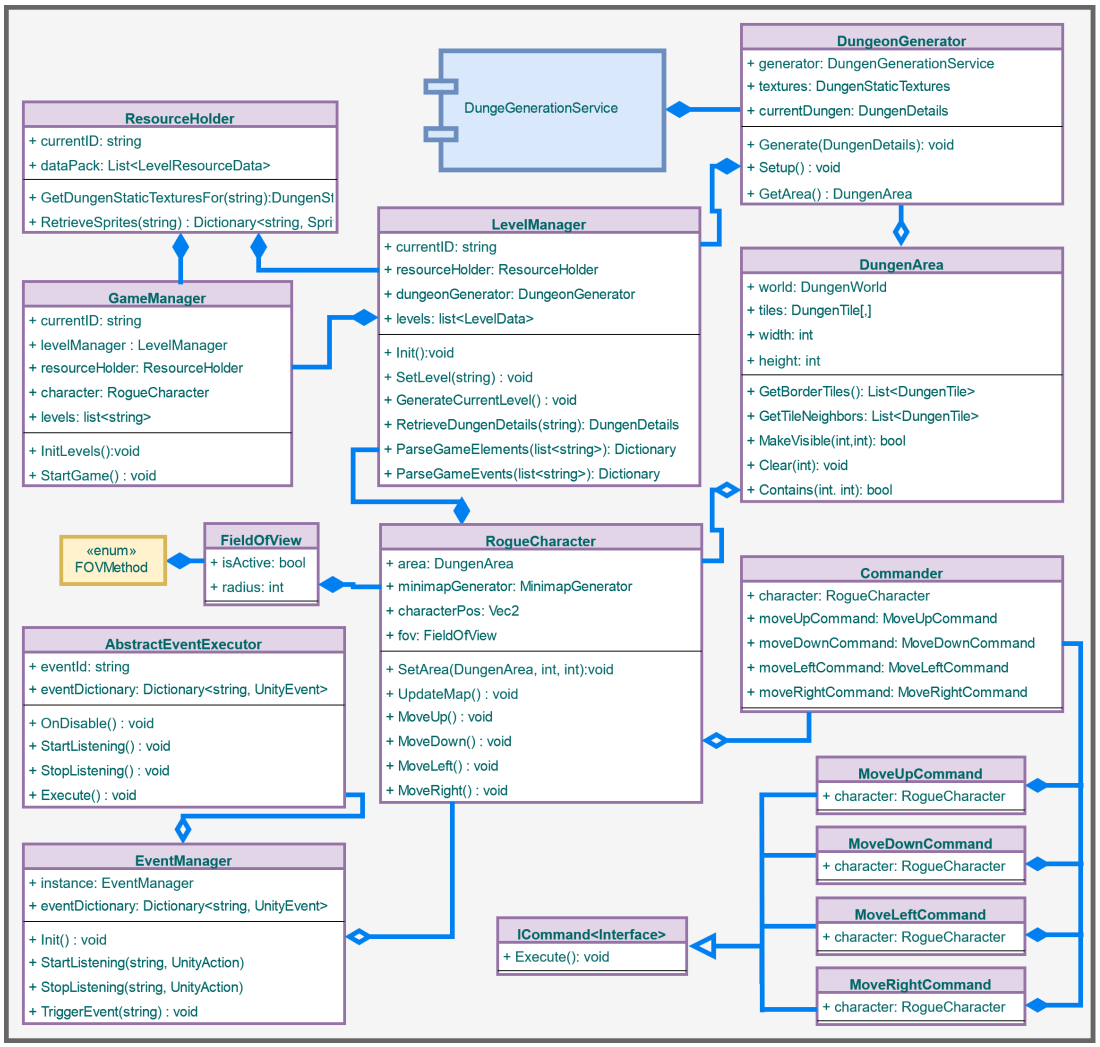

Figure 4.15: Class design for the game output

There is no doubt that the nodes should be planned well. However, many changes may occur during the development or after the release of the node set. Therefore, keeping the design as independent as possible would provide many benefits.

Controlling the game is designed with the command pattern instead of checking the inputs in the RogueCharacter class. Therefore a dedicated game object checks the input, and when the required conditions are applied, it commands

to the character. The character still needs to implement the reactions to every command.

The controller pattern is used in GameManager, LevelManager, and EventManager classes. The controllers only contain the information to make actions happen instead of having the whole logic like generating a dungeon. LevelManager has the DungeonGenerator information and orders it to generate a dungeon with the given dungeon generation methods. For a detailed understanding please see Nystrom [2014].

Field of vision (FoV) functionality uses the strategy pattern in order to stay available for providing new FoV algorithm. Therefore, if a new node created from the editor side, a new strategy should be written here.

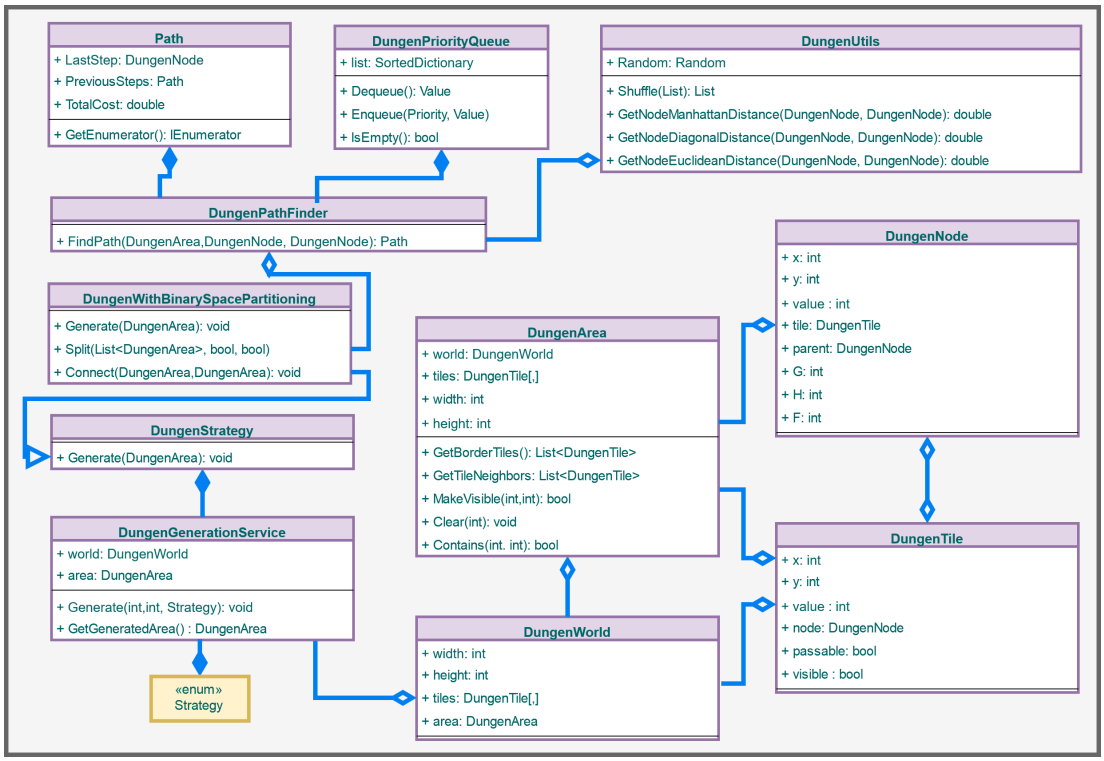

Figure 4.16: Class design for Dungen

Figure 4.16 shows that the whole dungeon creation mechanism wrapped up in a separate service which comes inside the runtime library. A dungeon corresponds to a level, and the level is configurable. Therefore sending the configuration to the service is enough to generate a new level.

The strategy pattern is a very powerful pattern and using it encouraged in the game side architecture since it supports the Open-Closed principle.

4.5.2 Running Pipeline

The base for the game is the dungeon. Therefore it needs to be created first, and all the other functionalities will be dependent on the dungeon instance which is

a graph with nodes. As it is mentioned before, the controller pattern is used in many places.

The controller at the top is GameManager which initializes the LevelManager and asks it to generate the current level. Then, it generates the current level and returns it, and the GameManager initializes the RogueCharacter using the knowledge of the dungeon as Figure 4.17 shows.

The rogue character object gets commands from an input controller and acts based on these commands. It has a reference to the generated world so, after every movement, the character sends feedback to the dungeon and dungeon changes based on the configurations for the field of vision algorithm.

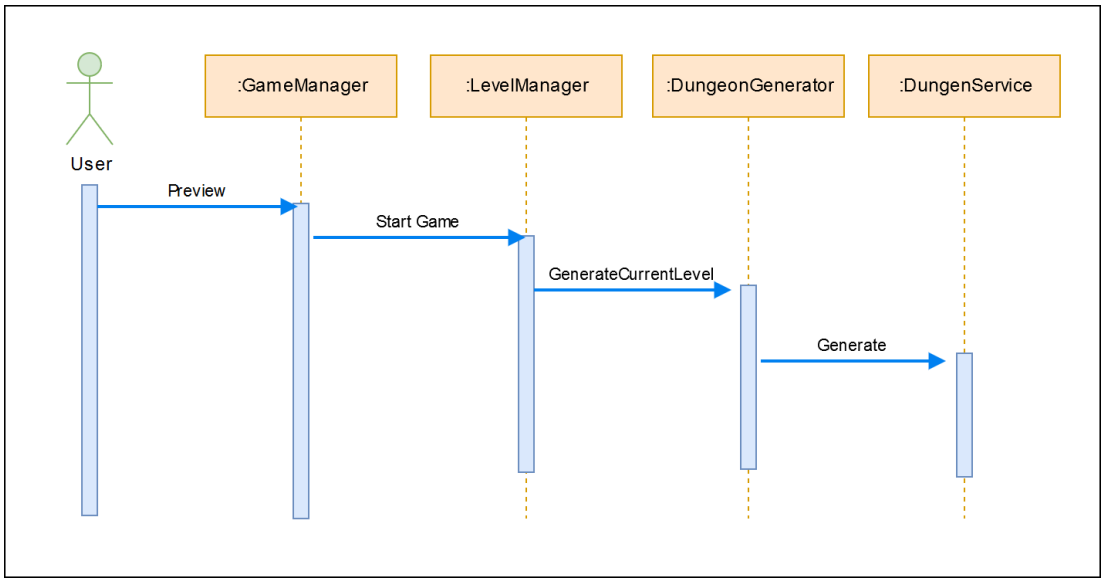

Figure 4.17: Delegation the generate command to the service

Resource Management: Resources are handled via data files. The editor extracts the resource information and saves them into a JSON file given in Figure 4.18. This file is read by ResourceHolder, and the holder loads the targeted resource files with supporting the sprite atlases and initializes itself. Then it works as an interface, and it provides resources to the game objects.

Event Mechanism: Event handling is a compulsory part of every game. The game engine API usually provides this feature. However, the usage in this thesis requires an alternative event mechanism. Therefore, a class called EventManager is created, and this class is capable of handling two types of events.

- **Unity Events:** Unity events are part of the engine, and EventManager uses an Observer pattern to catch these events. These events are hidden from the end user and generated during the compile operation. Thus unity events should be maintained by the developer of the framework.

```
{
"id": "MJa9EHXjE0SxOI3t6EISvA",
"roomWallTextures": [
   "Assets/Resources/rogue.png&name=rogue_21",
   "Assets/Resources/rogue.png&name=rogue_30",
   "Assets/Resources/rogue.png&name=rogue_29"
],
"roomTextures": [
   "Assets/Resources/rogue.png&name=rogue_32",
   "Assets/Resources/rogue.png&name=rogue_33"
],
"obstacleTextures": [],
"enter": "",
"exit": "Assets/Resources/rogue.png&name=rogue_20"
}
```


Figure 4.18: JSON resource definition for one level

- **Custom Events:** This event type is developed due to a necessity of providing event extensions to the user. Besides, Unity Event does not take any parameters which are kind of good for independent events. However, some case like when user collided with an item, the character should know which item it was.

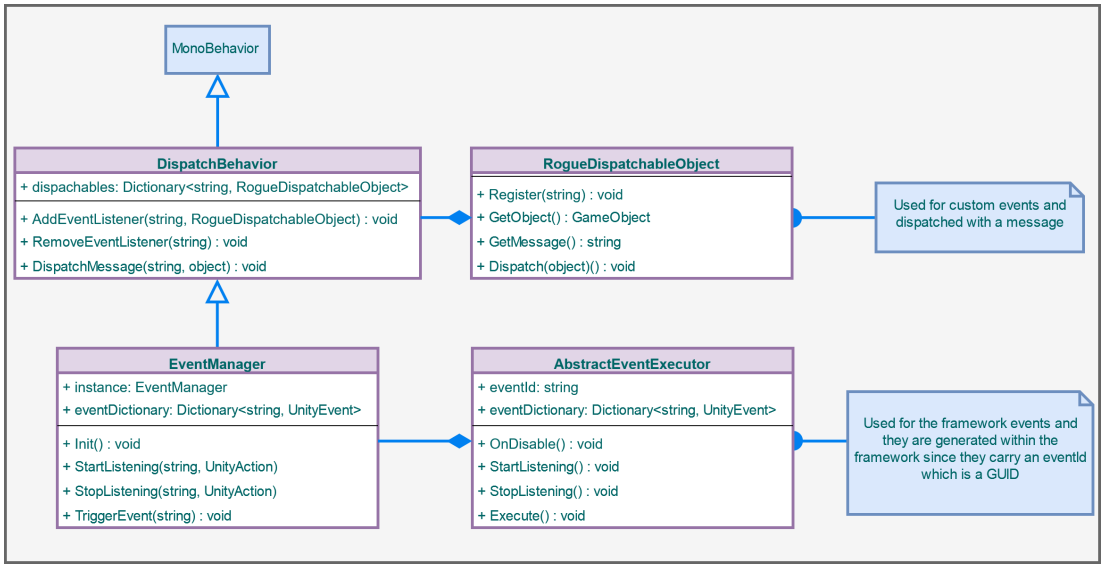

Figure 4.19: Generic and Custom events within the framework

The class diagram of the custom event system is given in Figure 4.19. DispatchBehavior keeps the subscribed game objects and their trigger message. RogueDispatchableObject abstract class registers automatically to the EventManager in case of any sub implementation of it (see Observer Design Pattern

in Gamma et al. [2016]). There are two critical methods which all the subclass' should implement.

- **Dispatch(object):** It should define the reaction to the event. It takes a parameter which is not optional.
- **GetMessage():** It should return the trigger message for the Dispatch method call. There is one situation in which user has to handle this method. A node called BroadcaseMessageEvent requires an implementation from the developer. This node requires a message to the editor and GetMessage() method should return the same message carried by that node. Otherwise, when this event is fired, nothing will happen since there is no handler.

4.5.3 Utilities

Utility classes are created in order easy some operations all over the framework. It is worth to mention that some utils are only usable from the editor.

- **JsonHelper (Editor and Runtime):** It wraps an array into a class and helps to serialize an array of arrays which is not possible with Unity as default.
- **SerializableDictionary (Editor and Runtime):** The Dictionary provided from the language is not serializable by the Unity. Therefore an alternative dictionary is created as serializable by unity.
- **CopyUtils (Editor Only):** It helps to clone a graph or node with replacing its references.
- **FontAwesome(Editor Only):** It wraps the Unicode correspondences of the FontAwesome font library. The icons used in the prototype are retrieved from the FontAwesome icon font library (see Martsoukos [2014]).
- **GraphUtils(Editor Only):** This class contains graph helper methods such as creating nodes or getting startup templates.
- **TextureUtils (Editor Only):** It wraps the texture display methods of the Unity and provides an easy way to create colors or load textures.
- **GuiUtils (Editor Only):** This util class contains practical methods for drawing on the editor window such as creating background grids.
- **NodeTypeUtils (Editor Only):** It is already mentioned that nodes are using a rule-based type checking system. This class provides the rules for the nodes.
- **RogueUtils (Editor Only):** The aim of this utility is providing high-level features like creating a new game.
- **Serializable2DArray (Editor Only):** Unity cannot serialize two-dimensional arrays. This implementation is an alternative way which is serializable.
- **SerializableStack (Editor Only):** Unity cannot serialize Stack as default. This implementation is a serializable alternative to the default Stack class.

4.5.4 Final Notes

Recall that, the architecture of the game and number of utilities will grow in the same direction with the editor. This chapter is proposed to give an example. The organizational approach may vary based on the features in the game. For instance, a minimap feature is included in the prototype. Therefore a minimap controller is included in the package, and this object is initialized in the character object. However, this could have been in the game manager as well. It is highly recommended that providing an architecture with the release of the framework will help the end user and guide when new components are created.

The framework that is created based on the proposed in the architecture functional and tend to grow in a maintained way. In the next chapter, there will be an experiment to create a straightforward roguelike game.

4.6 Results

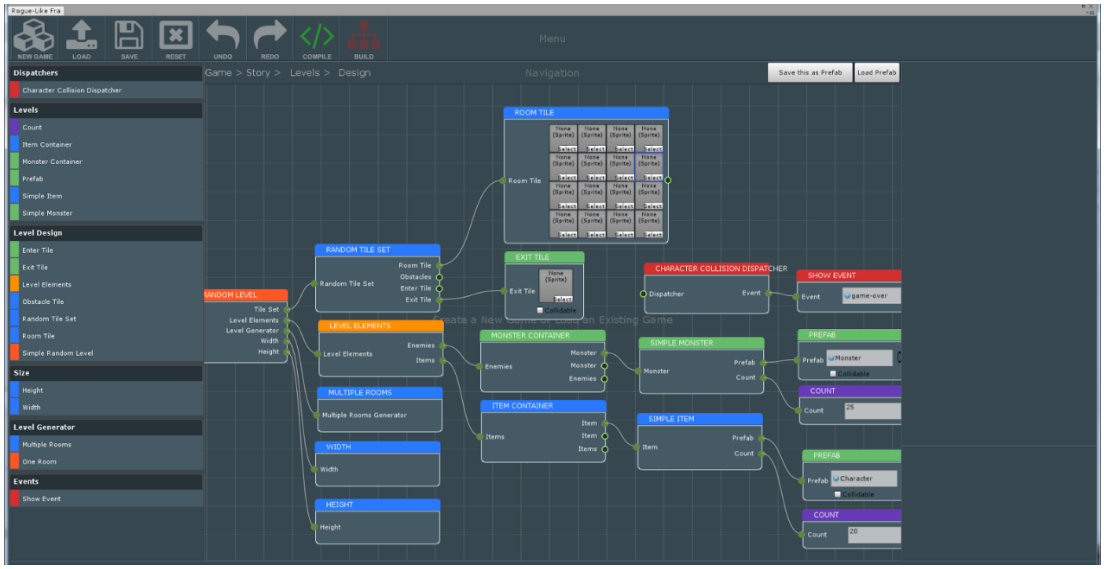

Figure 4.20: User Interface of the prototype

We have made a prototype based on the architecture given in this chapter. In the figure 4.20, It can be the visual editor interface and node editor. On the left side, there is a sidebar which contains possible nodes for the context. On the top, there are two menus provides navigation. The most top one handles the high-level operations such as creating a new game. The bottom one handles navigation inside the game tree. On the right side, there is a properties view and the help section which documents the nodes inside the editor.

In the middle, there are nodes connected to each other to create a meaningful structure. When the end user clicks on the Compile and Build buttons, the game in the figure 4.21 is generated. As it can be seen from the Figure 4.22 as

Figure 4.21: A screenshot from the generated game

well, the game is able to produce randomized content in the roguelike definition. Such projects are best reviewed with the video content, therefore please see the electronic attachment: *rfl-introduction.mp4*.

As a result, this simple game supports;

- Random level generation,
- Monster and item placing,
- Discoverable game world,
- The field of vision,
- In-game events,
- Adjustable minimap.

It is only a prototype, however creating a game like this only takes 5 minutes with it. As the framework is developed more, the output will be more promising than this state accordingly.

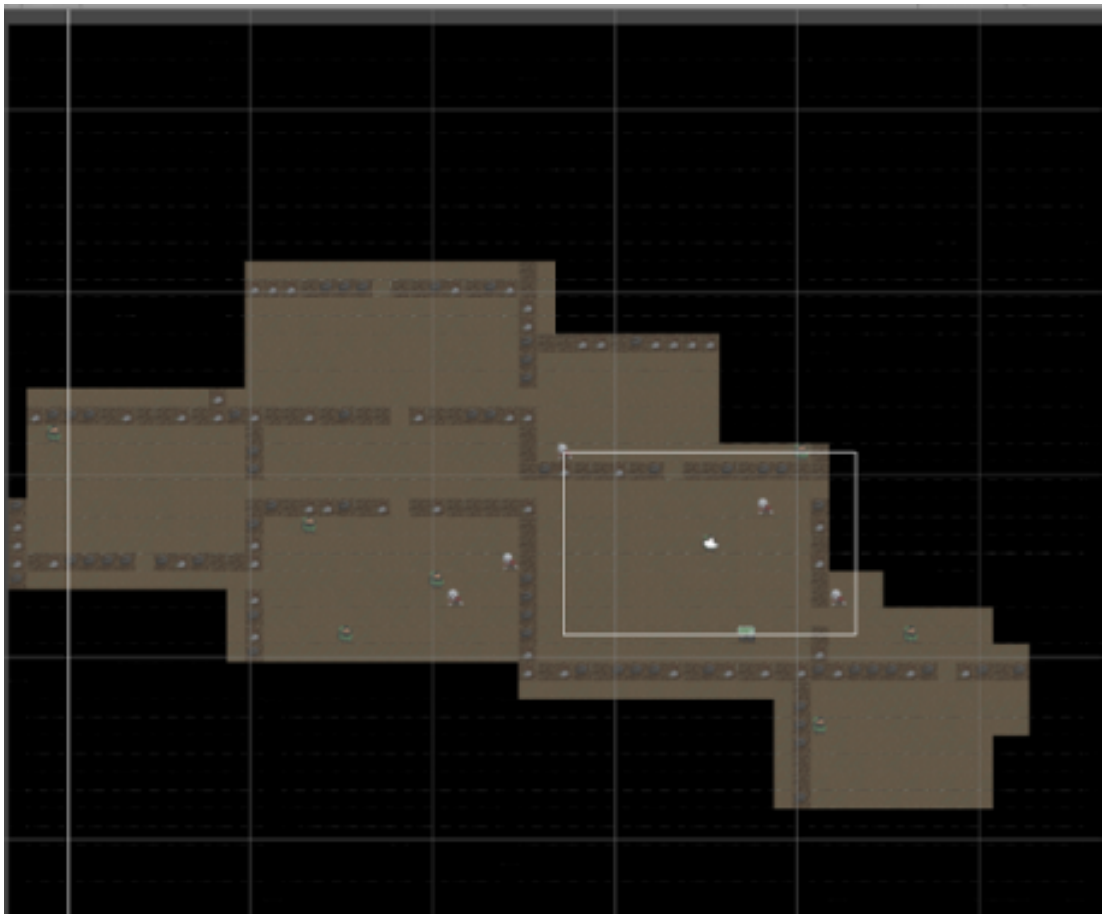

Figure 4.22: Procedurally generated dungeon

5. Conclusion

Visual Programming tools are becoming more appealing to the different user groups. The aims of the visual programming vary, and this programming approach can appear even outside of the computer graphics. This result is entirely natural due to the graphical representation.

The usage of these type of tool in the computer games is usually scripting. However, there are other areas like shader editing and creating artificial intelligence. This way of representation increases the understanding and decreases the time for development.

Duration of the game development should be shorter than ever to compete in the today's market. Therefore, plenty of tools are produced for the game developers and studios. In the scope of this thesis, an approach is proposed to ease the game development process for the roguelike video games development with a visual programming interface.

5.1 Contributions

The tools which have been created for the roguelike games are outdated, and their userbase is shrinking due to usability issues. It is not expected to develop tools outside of the modern game engines from the today's game developer community. Therefore this framework is proposed for the Unity game engine which has a giant user base and the games created with this tool could be exported to different platforms.

There are various problems with the classic approach in game development due to the involvement of the artists, writers, and designers. This tool gives an easy to understand game representation with nodes and all the team can involve to the development lifecycle with seeing the results immediately. Especially for the artists, the graphical manipulations can be done through the texture nodes. This advantage gives a decent chance to work independently.

The framework is designed as abstract as possible. Therefore, the project can be pivoted or extended to another game genre by keeping the same architecture. For instance, another framework can be created by aiming the platform games, or this framework can be extended to support RPG games.

There are many common points in a game genre, and repetitive actions can be represented by visual nodes. This type of programming approach decreases the time to spend for the development and should be used in the following years in the game industry.

5.2 Future Work

The roguelike framework can be extended in various ways. It would be wrong to limit the future work, but a list of features is given below as the most prominent ones.

- **Dependency Injection (DI):** There are a couple of DI libraries on c#. It is a proven beneficial practice to manage the dependencies. Instead of operating all the instances, having a DI library handling this situation is better for the maintainability. It is also worth to state that using DI can increase the performance surprisingly since most of the libraries give smart object creation strategies.

- **Procedural Story Generator:** One of the biggest fun factors in the roguelike games is the procedural content creation. It is also possible to generate procedural stories with user-provided restrictions. The framework has the story node, and the current platform does not support any story content.

- **RPG extension:** Roguelike games and RPG games have inspired from each other since they share some common features. The features need to be required for the RPG games can be introduced to the Node Graph API and the interpreter. This will also extend the user base.

- **Audio support:** There is no audio support inside the framework. However, it is already possible in the Unity. Creating custom events would be enough to play sounds, but Audio support can be given natively. Procedural background music creation can be part of the framework.

- **Keyboard Orientated Controlling:** Drag and drop method is one of the most comfortable controlling types, however, if it becomes repetitive it becomes somewhat cumbersome and not comfortable at all. Therefore, the controlling mechanism can be faster and less distractive by introducing some keyboard shortcuts. For instance, pressing space key when the cursor is on the node output can show a box with possible node set. Then, a mouse click can be used to choose the node and afterward a new node is created with a link automatically.

Bibliography

Dimitar Asenov and Peter Muller. Envision: A fast and flexible visual code editor with fluid interactions (overview). In *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, jul 2014. doi: 10.1109/vlhcc.2014.6883014.

Andrejs Bajovs, Oksana Nikiforova, and Janis Sejans. Code generation from uml model: State of the art and practical implications. *Applied Computer Systems*, 14(1), Jan 2013. doi: 10.2478/acss-2013-0002.

International Roguelike Development Conference. Irdc 2008, 2008. URL `http://www.roguebasin.com/index.php?title=IRDC_2008`.

T. O. Ellis, J. F. Heafner, and V. L. Sibley. The grail project: An experiment in man-machine communications. *Research Memoranda*, Sep 1969.

Dorota Celińska Eryk Kopczyński and Marek Čtrnáct. Hyperrogue: Playing with hyperbolic geometry. In Carlo H. Séquin David Swart and Kristóf Fenyvesi, editors, *Proceedings of Bridges 2017: Mathematics, Art, Music, Architecture, Education, Culture*, pages 9–16, Phoenix, Arizona, 2017. Tessellations Publishing. ISBN 978-1-938664-22-9.

Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 2016.

Mark Hills, Paul Klint, Tijs Van Der Storm, and Jurgen Vinju. A case of visitor versus interpreter pattern. In *Proceedings of the 49th international conference on Objects, models, components, patterns*, Zurich, Switzerland, 2011.

George Martsoukos. An introduction to icon fonts with font awesome and icomoon — sitepoint, Sep 2014. URL `https://www.sitepoint.com/introduction-icon-fonts-font-awesome-icomoon/`.

Emma McDonald. The global games market 2017 — per region & segment, 2017. URL `https://newzoo.com/insights/articles/the-global-games-market-will-reach-108-9-billion-in-2017`.

Farouk Messaoudi, Gwendal Simon, and Adlen Ksentini. Dissecting games engines: The case of unity3d. In *2015 International Workshop on Network and Systems Support for Games (NetGames)*. IEEE, dec 2015. doi: 10.1109/netgames.2015.7382990.

Bertrand Meyer. *Object-Oriented Software Construction (Prentice-Hall International series in computer science)*. Prentice Hall, 1994. ISBN 0136290493.

Joseph M. Morris. Traversing binary trees simply and cheaply. *Information Processing Letters*, 9(5):197–200, 1979. doi: 10.1016/0020-0190(79)90068-1.

Robert Nystrom. *Game programming patterns*. Genever Benning, 2014.

R. Perrey and M. Lycett. Service-oriented architecture. In *2003 Symposium on Applications and the Internet Workshops, 2003. Proceedings.* IEEE Comput. Soc, 2003. doi: 10.1109/saintw.2003.1210138.

Unity Public Relations Report. Fast facts, 2017. URL `https://unity3d.com/public-relations`.

Steve Rabin. *Game AI pro: collected wisdom of game AI professionals.* CRC Press / Taylor&Francis Group, 2014. Chapter 6.

Niko Schwarz, Mircea Lungu, and Oscar Nierstrasz. Seuss: Decoupling responsibilities from static methods for fine-grained configurability. *The Journal of Object Technology*, 11(1):3:1, 2012. doi: 10.5381/jot.2012.11.1.a3.

Andrew Troelsen. File i/o and object serialization. *Pro C# 5.0 and the .NET 4.5 Framework*, page 753–800, 2012. doi: 10.1007/978-1-4302-4234-5_20.

Unity Unity User Documentation. Unity user documentation, 2017. URL `https://docs.unity3d.com/`.

List of Figures

A. Minimum System Requirements

The prototype is tested on;

- Unity3D API v5.4.6.
- Windows 7 and Windows 10
- Graphics card with DX9 (shader model 3.0) or DX11 with feature level 9.3 capabilities.