Adam Jonathan Biggs

Registration number 10097567

2023

Dungeonify: Generating Dungeons & Dragons battle maps from OpenStreetMap

Supervised by Dr Stephen Laycock

University of East Anglia
Faculty of Science
School of Computing Sciences

Abstract

This report describes *Dungeonify*: a system for generating a textured battle map for use in the *Dungeons & Dragons* role-playing game. Typically, a player known as a Game Master is responsible for creating a suitable battle map before play begins. However, the manual design process is time-consuming and producing a detailed map that matches the theme of the game requires significant skill. While high quality maps are available for purchase, they are often too expensive for hobbyists and it can be difficult to find a map that aligns closely to the Game Master's storyline. Recently, there has been growing interest in *automated* battle map generation methods. However, all known methods require a user to define the structure, typically by drawing outlines. This report proposes an alternative approach: a user can find a suitable location on Earth using a Web Mapping Tool (e.g. OpenStreetMap) and "dungeonify" it; that is, transform it to take on the theme and appearance of a *Dungeons & Dragons* battle map. It is thought that this will lower the time and skill level required for the Game Master, and perhaps will encourage more *Dungeons & Dragons* players to take on the Game Master role. The sections of this report cover necessary introduction, describes the design, implementation and evaluation of the *Dungeonify* system before discussing the approach and proposing areas for future work. The report starts by describing the need for an automated *Dungeons & Dragons* Battle Map generator, and provides motivation for the *Dungeonify* approach. Related work is considered before a Design section which covers the project's requirements, risks and proposes a work schedule. Key components of the *Dungeonify* implementation are discussed next before a robust evaluation and analysis section, which assesses the quality of the technique on various worldwide locations. The report finishes with a conclusion and a section that discusses opportunities for further investigation.

Contents

1 Introduction **7**
1.1 Report Plan 8

2 Background **8**
2.1 Battle Map Design Criteria 8
2.2 Virtual Tabletop 9
2.3 Related Work 11
2.3.1 Structure Generation 11
2.3.2 Web Mapping Tool 11
2.3.3 Structure Details 12
2.3.4 Game Dungeon Mapping 12
2.3.5 Structure Exteriors 12
2.3.6 Machine Learning feasibility 13

3 Design **13**
3.1 MoSCoW Analysis 14
3.2 Knowledge Dependency 15
3.2.1 Web Mapping APIs Compared 18
3.3 Risks 19
3.3.1 Scope 19
3.3.2 Equipment 19
3.3.3 Technology 19
3.3.4 Timescale 20
3.3.5 Budget 20
3.4 Program Design Decisions 20
3.5 Planned Testing 22
3.6 Stated Success Criteria 22
3.6.1 Generated Battle Map Success Criteria 23
3.7 Gantt Chart 23
3.7.1 MoSCoW Analysis Coding Tasks 23
3.7.2 *Minimal Viable Product* 24
3.7.3 Testing, Code Delivery and Portfolio Submission 24

4 Implementation **25**
4.1 Full Workflow 25
4.2 Key Components 28
4.2.1 `coord_conversion.py` 28
4.2.2 `osm_factory.py` 29
4.2.3 `gui.py` 30
4.2.4 `dungeonify.py` 31
4.3 Testing 36

5 Results and Evaluation **36**
5.1 Constructing an evaluation dataset 37
5.2 Full Workflow Evaluated 37
5.3 OpenStreetMap Data Consistency Check 38
5.4 Complex Structures Evaluated 39
5.5 Evaluating the Variety of Generated Textures 41
5.6 Running Time of Full Workflow 41
5.6.1 Running time per 4-node structure 42
5.6.2 Running time when selecting multiple structures 42
5.6.3 Running time when selecting more nodes per structure 43
5.6.4 Effect of increasing the battle map size 44
5.7 Running Time of System Components 44
5.7.1 Individual Processes Key 45
5.7.2 Varying number of 4-node structures 45
5.7.3 Varying number of any-node structures 46

6 Conclusion **47**

7 Future Work **48**

References **49**

List of Figures

1 **Manually created battle maps**: Battle maps manually created by the author using Dungeondraft, taking approximately (a) 5 hours and (b) 8 hours of work. 10

2 **Virtual Tabletop applications**: User interface of two popular virtual tabletop applications. *Left*: Roll20 and *Right*: foundryVTT. 10

3 *Dungeonify* Project Gantt Chart 24

4 **System overview**: User selects an OpenStreetMap location (a), Dungeonify accesses structure node data via the API (b), selected buildings are redrawn (c), rotated (d), mapped to grid (e) and textured (f). 27

5 **Loading Dungeonify battle map into Virtual Tabletop**. (a) Example .dd2vtt battle map metadata file and (b) viewing Dungeonify battle map in Virtual Tabletop. 27

6 **System flowchart**: The operation of the program. 28

7 **Qualitative Results**: *Best viewed zoomed in*. Dungeonify run on Evaluation Locations. *Columns*: (a) Input OpenStreetMap location, (b) loaded into Dungeonify buildings selected (pink arrows), (c) buildings mapped to grid and (d) final rotated and textured output. 38

8 **Performance analysis on complex scene**: Evaluation on input with curved buildings. (a) OSM input, (b) Selected structures, (c) Rotation stages, (e) Final battle map, zoomed on the 4th structure. 40

9 **Texture generation**. *Left*: *Dungeonify* asset folder, *Right, Top*: Four structures selected from a street. *Right, Bottom*: The structures generated with variation in textures across wall, window sill, window and door assets. 41

10 **Failure case**: Dungeonify loads a complex structure (a), rotates (b) and textures it (c). Unreachable areas are created (b) which are inaccessible (c). 49

List of Tables

1 Programming Languages Compared 16

2 Virtual Tabletops Compared 17

3 Web Mapping Tools Compared 18
4 **Evaluated Locations**. Access via https://www.openstreetmap.org/{URL} 37
5 **Complex Location**. A complex location with curved structures. Access via https://www.openstreetmap.org/{URL} 40
6 **Running time per structure**. Running time for 4-node structures in the Evaluated Locations. 42
7 **Multiple structures**. Running time for selecting multiple 4-node structures per Evaluated Location. 43
8 **Structures with more nodes**. Evaluating the effect of selecting structures with more nodes. Tested on Evaluation Location 2. 44
9 **Increasing map size**. Evaluating the running time to generate four 4-node structures at larger map sizes (in square inches) for Evaluated Location 1. 45
10 **Avg. running time of different components, 4-node structures**. Running time of different system components, averaged over 5 different choices of structure set. 46
11 **Avg. running time of different components, any nodes**. Running time of different system components, averaged over 5 different choices of structure sets with different numbers of nodes. 47

1 Introduction

A Tabletop Role-Playing Game is a group activity where players describe their characters' actions through speech, and which often uses dice to provide an element of randomness. One well known game system called *Dungeons & Dragons* is set in a medieval fantasy world similar in theme to The Lord of the Rings fantasy series (Tolkien, 1954). In the early days of its creation, *Dungeons & Dragons* was a simple spoken game system – what is commonly referred to as 'the theatre of the mind', meaning everything is imagined like reading a 'choose your own adventure book'. As technology has modernised, so has the format of such Role-Playing Games, moving online and involving a stronger use of computer graphics to visualise game sessions which often use dedicated software called Virtual Tabletop systems.

Dungeons & Dragons requires a Game Master (also called a Dungeon Master) and several other players - usually between 3 and 8 - to play it. A typical *Dungeons & Dragons* game will have the Game Master describe a specific scenario that the players' characters are in and allow the players to interact with it. The Game Master is responsible for preparing the game session. To do this they invent Non-Player Characters, enemy creatures and *battle maps* which the players interact with throughout the game. A battle map, in the context of a Role-Playing Game, is a depiction of a particular scene the Game Master wanted to convey through visuals rather than through 'the theatre of the mind'. Battle Maps have varied uses in the *Dungeons & Dragons* Role-Playing Game system, and despite the name not all of them involve combat.

Battle Maps provide three main functions for a *Dungeons & Dragons* game: (1) to provide the players with a strategic overview of a scene similar to that of a strategy video game, (2) to convey a particular story line which progresses the plot the Game Master has created, and (3) to provide a visually interesting background, allowing the players to feel more immersed within the game.

There are various options available for Game Masters to create battle maps, including manual design methods, purchasing a pre-existing map or using map generation software. However, all of these methods have various drawbacks, either as they limit the imagination of the Game Master or require significant time or expertise. These are fully explored in Section 2 of this report. *Dungeonify* aims to improve on these methods, by generating in-theme battle maps with a layout selected by the Game Master using OpenStreetMap.

1.1 Report Plan

This report describes the design, implementation and evaluation of *Dungeonify*, a system for generating *Dungeons & Dragons* battle maps specified by a web mapping tool (e.g. OpenStreetMap). Section 2 describes the key concepts critical for the design of the project. In addition, related work is discussed and used to inform on later design choices.

Section 3 presents the success criteria for the project, in the form of a MoSCoW Analysis. The critical knowledge dependencies are compared and risks to the success of the project determined. Informed by the related work, progam design decision are stated which involve initial coding exercises to gauge the feasibility of implementing such features. Lastly, a schedule for the project is specified. Section 4 establishes the full workflow pipeline the program uses to meet the success criteria. Each major stage is considered and notable code is shown in greater detail. In addition, how the project was tested throughout its development is shown.

Section 5 shows the results of the project, measured on a number of factors, which informs the effectiveness of the project in the conclusion. Section 6 determines if the *Dungeonify* project has met the requirements of both the MoSCoW analysis and success criteria. Finally, Section 7 establishes ways in which the *Dungeonify* project could be extended.

2 Background

This section describes components required by the *Dungeonify* project and discusses related work.

2.1 Battle Map Design Criteria

The success of a battle map is subjective to the designer and depends on their intended use case. However, the creation of each battle map requires a minimum agreed upon set of standard criteria to check against. These most basic criteria are stated below:

- Structure walls must be built fitting to the inch squared grid.
- Structure walls must meet each other at 90 degree angles.

- Window and door assets must be applied over wall textures and not overlap.
- Internal flooring textures must appear only on the interiors of all the structures.
- External flooring textures must appear only on the exteriors of all the structures.

For a Game Master, creating a suitable battle map involves a lot of work. Firstly, they are required to find a large dataset of 'from a bird's eye view' textures that fit the theme of the world. Secondly, a plan must be created that details what the battle map will be used for – i.e. what location will it show and whether combat will occur on it. Creators then begin layering textures in either a dedicated *Dungeons & Dragons* battle map builder such as Dungeondraft or within a general purpose image processor like GIMP2 or Clip Studio Paint. After several hours of work, the resulting battle map can then exported as either a Virtual Tabletop compatible file or a lossless image file.

Another challenge for some Game Masters is that *Dungeons & Dragons* games are often characterised as providing a true sense of freedom for the players. A consequence of this is that a Game Master can never truly know when they may require a battle map as the players may fight unplanned battles or avoid combat altogether.

For a Game Master to effectively visualise a scene within the bounds of a Role-Playing Game, the creation of visually appealing battle map for a variety of locations is required. Many Game Masters prefer to use pre-existing battle maps found online despite the challenges of making the displayed location fit their chosen storyline, required layout and theme. Due to this, many Game Masters prefer to design their own, which is often time-consuming.

2.2 Virtual Tabletop

A Virtual Tabletop (VTT) is a software system designed to provide tools for a Role-Playing Game to be played online. Core characteristics include rolling virtual dice, storing player characters data and displaying battle maps in a visually interesting way.

Virtual Tabletop systems often attempt to make the time-consuming process of creating a *Dungeons & Dragons* game easier for a Game Master by automating some features. One such example is the importing Virtual Tabletop compatible files, which enables the use of battle maps created earlier or by others. The files store data about a battle map including: the image, the positions of structure walls, windows and doors. A

Figure 1: **Manually created battle maps**: Battle maps manually created by the author using Dungeondraft, taking approximately (a) 5 hours and (b) 8 hours of work.

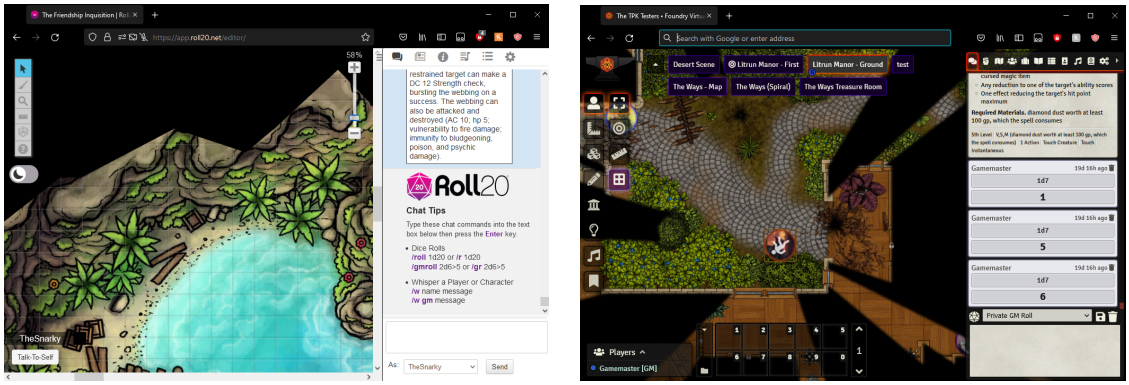

Figure 2: **Virtual Tabletop applications**: User interface of two popular virtual tabletop applications. *Left*: Roll20 and *Right*: foundryVTT.

Virtual Tabletop displays all of this information to render the scene with dynamic lighting, and prevents characters from seeing past walls and closed doors. These features make the game more immersive for both the players and the Game Master.

2.3 Related Work

A number of comparisons can be drawn between the expected core functionality of *Dungeonify* and existing academic projects. These are discussed below:

2.3.1 Structure Generation

Several methods (Kelly, 2021; Grabler et al., 2008) have been presented to generate 2D and 3D structures on a city-wide scale. Rule based decisions for generating structures at a large scale have been proven to be flexible, fast and responsive when producing structures at a large scale. Equally, the ability to program operations on existing map data for the generation of key structure details is a great advantage of such systems. Grabler et al. (2008) introduced a rule-based system which determines the perspective the viewpoint should be from, according to the direction from a structure to its nearest road. Likewise, one could imagine a similar rule-based system for determining the door and window placement could be applied to *Dungeons & Dragons* battle map structures. Despite the advantages of using a completely rule-based generation algorithm for creating many structures within a city, it does prove to be less accurate than using existing real-world mapping data. To this end, several existing web mapping tools are next evaluated for use in *Dungeonify*.

2.3.2 Web Mapping Tool

Initially Google Maps (Mehta et al., 2019) was considered the most effective web mapping tool for this project, due to its extensive use world-wide. Unfortunately, it is impractical to use the tool for an individual student project due to the cost far exceeding the monetary budget this project allows. To this end, an investigation into alternatives was conducted. Academically, the OpenStreetMap (Girres and Touya, 2010) web mapping tool is selected for this project, owing to the free use of its Application Programming Interface (API). In addition, the OpenStreetMap has a very large dataset of structures, roads and terrain types accurate to within 6 metres (Haklay, 2010). This would be more than accurate enough to represent buildings for converting into *Dungeons & Dragons* battle map structures.

2.3.3 Structure Details

There are numerous challenges to placing details on and within a structure. Kelly (2021) described the practicalities of detailing windows on a 3D structure using rule-based algorithms. Similar rule-based algorithms could reasonably be applied to determine internal doorways and furniture visible within each structure. Alternatively, as Lopes et al. (2010) describes, the generation of floor plans using purely procedural methods can make for an effective solution. This task would be feasible within the constraints of the project. Another method could use a lazy generation algorithm which has the added benefit of being able to generate scenes very quickly (Hahn et al., 2006). For *Dungeonify*, however, all the buildings need to be fully constructed when the structures are first generated, mitigating the benefits that a lazy algorithm can provide.

A machine learning approach for designating rooms of each structure and then placing furniture within them would be the ideal solution for this problem, as many agree (Wu et al., 2019; Camozzato et al., 2015). Constructing structure interiors given very little starting information, as Lopes et al. (2010) describes can be made possible even for a drawn plan of a structure. However as will be explained further in the Machine Learning feasibility section below, the disadvantages of using such systems outweigh the positives.

2.3.4 Game Dungeon Mapping

Several algorithms exist for creating *Dungeons & Dragons* style battle maps for *Dungeonify* complexes - a staple for traditional Role-Playing Games . As Mawhorter and Mateas (2010) describes, using tree algorithms can be useful given no limit on horizontal or vertical space (Linden et al., 2014). The generation of *Dungeonify* interiors by having the user determine some key features and then using an algorithm to finish off the rest is a very interesting design principle. If these techniques could be applied for predetermined structures from a web mapping tool this would offer Game Master's flexibility to fit the battle map to a specific theme.

2.3.5 Structure Exteriors

The exteriors of structures is often just as important as the interior. To this end, foliage will often improve the aesthetic quality of a location. Algorithms included in Carey

(2019) could be one such case. However, as *Dungeons & Dragons* battle maps have no vertical height, the inclusion of height maps may be underused in such an application (Emilien et al., 2012).

It is conceivable, instead, to alter the types of exterior textures dependant on the height above sea level of areas near the structures. Examples of this could include trees acclimated to growing at specific heights, or cliff side textures if a large change in height was detected. This information would not be available in most web mapping tools and instead have to be pulled from an elevation height tool, for example from a Digital Surface Model (DSM) or Digital Elevation Model (DEM).

2.3.6 Machine Learning feasibility

As stated above, at first glance, the generation of *Dungeons & Dragons* battle maps using machine learning techniques be the most ideal solution, for exteriors and interiors alike. For example, Generative Adversarial Networks (GANs) (Goodfellow et al., 2014), have been successfully used to generate photorealistic images (Almahairi et al., 2018; Hiasa et al., 2018). For *Dungeonify*, the generator could be trained to improve its generation method at the same time the discriminator checks the output quality. However, GANs are known to require huge datasets of training images to guarantee any accurate results (Almahairi et al., 2018). For battle maps, there are far too few available for this purpose (Hiasa et al., 2018). In addition, training GANs would require a high performance multi-GPU computing system which is unavailable for this project.

3 Design

The design of the project is considered with respect to the aforementioned related work. The coding feasibility requirements for the project and any risks that may complicate it throughout its development are considered. To this end, the design of the project will be split into a number of distinct sections including a MoSCoW Analysis for the project requirements, comparisons determining critical dependencies the project will be built with, a risk analysis and a project schedule.

3.1 MoSCoW Analysis

In its most basic form, the goal of the project is to create an application which enables a user to automatically construct a *Dungeons & Dragons* battle map Virtual Tabletop compatible file from a selected number of structures identified using a web mapping tool (e.g. OpenStreetMap). To achieve this most basic ideal, the program must meet several core targets:

- Access a Web Mapping Tool's Application Programming Interface (API) to retrieve the layout of buildings in a real-world location.
- User Interface (UI) enabling users to select a number of structures.
- Create a battle map image by layering partially transparent textures.
- Generate a Virtual Tabletop compatible file for the battle map image.

The MoSCoW Analysis methodology is an effective tool to expand on the above requirements, categorising them into stages of feasibility.

Must

- Use structure data from a Web Mapping Tool's API.
- Process API data to create a basic *Dungeons & Dragons* battle map - a minimum of one structure with a single set of textures.
- Enable the resulting battle map data to be used in a Virtual Tabletop game by loading the generated Virtual Tabletop compatible file.

Should

- Implement a User Interface which a user can use to select at least one structure from the web mapping tool.
- Utilise algorithms to correctly rotate and reposition structures in the battle map ensuring they meet the Battle Map Design Criteria described above.
- Use the Web Mapping Tool's structure tags to add textures based on the building function. For example, a structure tagged as a special structure of some kind should use a different texture set from all the residential buildings.

Could

- User Interface features enabling basic image processing tools.
- User Interface implemented options which enable the user to specify design features of the battle map.
- Use Digital Elevation Models (DEM) and Digital Surface Models (DSM) to determine texture sets to use. For example, by determining the type of foliage texture based on the distance above sea level of the location on the Web Mapping Tool.
- Vary each individual structure, by randomly determining the textures each structure will generate with.

Will Not Have

- Be able to produce battle maps for themes other than *Dungeons & Dragons*.
- Be able to produce *Dungeons & Dragons* battle maps in three-dimensions. [1].
- Be a full battle map editor. The software is designed to generate a battle map, it will not allow full customisation of the final image and Virtual Tabletop file.

A Minimum Viable Product (MVP) in a software engineering project is often useful to determine the simplest version that meets a minimum set of requirements. In terms of the MoSCoW Analysis above the Minimum Viable Product must meet the needs of the Must section. Once met, time can then be spent on completing elements outlined in the Should and Could categories, taking care to avoid any content leading to the Will Not Have section elements.

3.2 Knowledge Dependency

This section compares various options for the programming language, web mapping tool and Virtual Tabletop system used for *Dungeonify*. The selected tools are listed in each category as knowledge dependencies. Some familiarity with these tools would be useful for the reader before they continue to the later implementation section.

[1] No Virtual Tabletops currently allow importing 3D battle maps.

Table 1: Programming Languages Compared[1]

Programming Language*	Python	Java	TypeScript	.NET C#	C++ OpenGL
Procedural	✓	✓	✓	✓	✓
Object-Oriented	✓	✓	✓	✓	✓
Web Applications	✓	✗	✓	✗	✗
Graphical Libraries	✓	✓	✗	✓	✓
Developer Experience	✓	✗	✓	✓	✗
API Connection	✓	✗	✗	✓	✗
High Ease Of Use	✓	✗	✗	✗	✗

[1] Data compared for usability to code effectively.

∗ Programming languages are compared solely for their effectiveness for building the *Dungeonify* Project. It is important to create the Minimal Viable Product quickly.

Programming Languages Compared

The following table compares many of the programming techniques and methodologies which could be used for *Dungeonify*, based on the MoSCoW Analysis above.

Based on this comparison, the *Dungeonify* project will be created using Python 3.7. The ease and speed of development, especially in connecting to an API, will lead to a fast Minimum Viable Product resulting in a clear advantage over the alternative Programming Languages.

It is assumed the reader has a working knowledge of the Python programming language, the use of APIs and of vector graphics software. Some previous experience with

Table 2: Virtual Tabletops Compared

Virtual Tabletop	FVTT [1]	Roll20	Tabletop Simulator	Astral	Fantasy Grounds	Adobe Photoshop
Free VTT File Importing	✓[a]	✗[b]	✗	✗	✗	✗
DLE	✓	✓	✗	✓	✗	✗
Single Cost	✓	✗	✗	✗	✗	✗
Modern Design	✓	✗	✗	✗	✗	✗
Customisation	✓	✓	✗	✗	✗	✗
Free API	✓	✓	✓	✗	✗	✗

[1] FVTT = Foundry Virtual Tabletop, DLE = Dynamic Lighting Engine

[a] FoundryVTT has a user made module, and usage is free after the initial one-time-payment.

[b] Roll20 does have a user made module, but it requires a monthly paid subscription.

Virtual Tabletop or Game Design software would be useful, but is not critical as these concepts are fully explained in this report.

Virtual Tabletop Software Compared

In the following table, a number of prospective Virtual Tabletop systems will be compared on a number of metrics. The most important aspect of this application is that it has a method of importing Virtual Tabletop compatible files. In addition, choosing one which is both modern and popular will provide a greater chance that a potential Game Master will want to use the *Dungeonify* software.

The most effective Virtual Tabletop system for this application would either be Roll20 - a widely used VTT - or Foundry Virtual Tabletop - a modern VTT with a more financially acceptable cost to use. Due to the increased functionality of the modern Virtual Tabletop, this project aims to generate VTT compatible files for the FoundryVTT sys-

Table 3: Web Mapping Tools Compared

	Open-StreetMap	Google Maps	MapBox	RapidAPI
Free	✓			
Tagging Data	✓	✓		

tem as it will perform better in handling battle map images. In addition, FoundryVTT provides a higher degree of control for the non-paid service to use its Virtual Tabletop file importer.

The VTT compatible files importable into the FoundryVTT system will either be JavaScript Object Notation (.json) files imported with the software alone, or use a unofficial module which imports Dungeondraft to Virtual Tabletop (.dd2vtt) files. This will be investigated further below.

3.2.1 Web Mapping APIs Compared

Several existing Web Mapping Tools and APIs servicing them could be used in the *Dungeonify* project. The most critical detail for this project, however, is that it must be free to use and store data describing the function of each structure. This data can then provide the tagging data to determine the types of textures used for the structures, as it outlined as a requirement in the MoSCoW Analysis above.

The OpenStreetMap API is the best choice for the *Dungeonify* project due to its free usage and stored structure tagging data. As previously considered in the Related Works section above, the data from this user updated collaborative web mapping tool is accurate to within 6 metres, with reference to the Web Mapping Tool subsection in the Related Work above, which is more than effective enough to represent physical structures for use in the *Dungeonify* application.

To more effectively test and evaluate the results at the end of the projects development, care will be taken to include a sample from the OpenStreetMap dataset which covers France. This will ensure that this accuracy is evaluated in this projects results.

3.3 Risks

As *Dungeonify* incorporates several systems, several risks should be considered to ensure its successful development.

3.3.1 Scope

The *Dungeonify* project has a large scope as specified in the MoSCoW Analysis above. It requires access to several distinct systems and the creation of complex data manipulation algorithms to provide a fast Minimum Viable Product. Further, the tasks specified in the Should and Could sections of the MoSCoW Analysis will become unfeasible should the MVP need extra time to be completed. To address these risks a timing schedule built as a Gantt Chart will be created in the Gantt Chart Section below, ensuring the fair allocation of time and resources to each task.

3.3.2 Equipment

A critically important risk for any software engineering project is the loss of the source code data. All computer components are subject to faults and errors and the *Dungeonify* code may be lost due to a hardware fault. To lower this risk, the project will use Git Version Control though the use of GitHub to store the source code data in the cloud. With respect to the program's feature planning, the project will also use the Kanban-style task board of Trello as a means to track the projects progress and for recording the tasks completed each week.

3.3.3 Technology

Using an online web mapping tool as the source for the global positioning data creates several risks. Should the business model of it change then this project would need to be altered. Equally, if the web mapping tool closes or experiences down time, a replacement would be required. Interacting with an API mitigates some of these risks as the *Dungeonify* code will be more adaptable to switching to another API service.

Another concern is the unlicensed use of third-party art assets for generating the *Dungeons & Dragons* battle map image for the project. Using an artist's texture images in a program can require specific licensing agreements. Using non-licensed artwork can also cause difficulties, however, as the artwork may become licensed in the future.

To mitigate this risk, the creators of the Forgotten Adventures (https://www.forgotten-adventures.net/) asset set were contacted. It was made clear that as the project will not be made public, free use of their textures would be acceptable. In addition, the project will be designed to accept any standard of artist assets as long as they meet specific criteria.

3.3.4 Timescale

The timescale for the project is 28 weeks. A significant portion of the time is dedicated to the programming of the project. In addition, several weeks must be provided for the write up of the report and the preparation for the inspection. Time management is a challenge for any project and so careful planning is required. The methods to reduce the risk of over extending the time scale will be covered further in the Project Plan and Gantt Chart sections below.

3.3.5 Budget

As previously referred to, this project is designed not to require a monetary budget for its development. If, however, any of the above risks come to fruition this may have to be readdressed. For example, using the Google Maps API as an alternative to the OpenStreetMap API as the web mapping tool will require a small budget.

3.4 Program Design Decisions

The critical design decisions for the implementation of the *Dungeonify* program must be directly related to the requirements set out in the MoSCoW Analysis above, especially in terms of the Must and Should sections. The code handling access to the OpenStreetMap API, generating a basic UI and loading the OpenStreetMap website using Python code will be tested prior to the start of the implementation. This ensures these critical aspects are feasible for the project and especially for the MVP which must be built within a short timescale.

Web Mapping Tool's API Access

A number of initial coding exercises have been conducted to gauge the feasibility of using data pulled from the OpenStreetMap API.

These use the official OpenStreetMap API Python library to collect data when it is provided with a latitude and longitude for a given zoom scale. The OpenStreetMap website can serve locations at a zoom factor of 19, though some areas can be at an even higher zoom factor. The *Dungeonify* program implemented will expect to use this zoom level factor of 19, which presents global positioning data at a street-sized scale. This will enable a user to more easily select a structure or multiple structures that could reasonably fit onto a *Dungeons & Dragons* battle map.

Virtual Tabletop File Generation

Some initial tests were conducted on the generation of the official FoundryVTT scenes which are programatically represented as JSON files. The aim was to create the underlying image file and link it to the scene data. The official FoundryVTT team was contacted to determine the effectiveness of this strategy. They advised against this method as it would prove to be inefficient and performance intensive for the FoundryVTT system.

It was decided to instead implement code that generates the .dd2vtt file – a standard file format also used by the Dungeondraft battle map generation software. These files can then be imported through the use of an unofficial FoundryVTT module called the Universal Battlemap Importer.

User Interface

Initial development of the User Interface was also conducted to become familiar with the main Graphics User Interface (GUI) generating Python Libraries available. The PyQt5 library was used to generate a basic image import and exporter. This will be critical to the later stages of the project.

Structure Rotation

The first stage of the proposed rotation algorithm will be to rotate every structure such that its longest wall becomes either horizontal or vertical. To accomplish this, some initial tests were carried out to begin manipulating randomly generated basic structure data.

3.5 Planned Testing

The development of the *Dungeonify* project will use the Agile Scrum Development Methodology. A number of stand-up supervisor meetings will be done one-on-one providing an opportunity to test the program in front of an audience and ensure tasks set in the Trello Board are being met. In addition, as part of the Agile development life cycle, a case by case selection of tests will be carried out after every major code implementation, ensuring no major errors are being introduced.

Through the initial coding exercises for the Structure Rotation section above, it was determined that this would be a likely error prone area of code in the final solution. To effectively test this area of code the implementation of Unit Testing will be essential. The reason for this is due to the large number of mathematical calculations which will need to be implemented to rotate and resize a structure. Unit Testing will enable automatic tests to be conducted comparing expected and newly calculated data.

3.6 Stated Success Criteria

The *Dungeonify* success criteria are informed by the Battle Map Design Criteria, the Related Work from the MoSCoW Analysis above. They will be used to determine if the generated battle map output from the program provides a successful result. Every generated battle map must meet or exceed these criteria for the project to be considered a success.

To effectively meet the MoSCoW Analysis requirement of adding textures based on each structures OpenStreetMap data tag, it was decided that a simple example should be used for this project. This will prove the method works without requiring the careful filtering of thousands of data tags from the dataset. Often in the Medieval Fantasy setting of *Dungeons & Dragons*, the structures in the settings towns are made of cheap materials such as wood, with the exception of religious structures. These buildings are often grandiose constructions comprised of the more expensive stone materials and having stained glass windows. With this theme in mind, the Success Criteria stated below will include the requirement that structures with tags of religious connotations must be generated using different textures from the rest of them.

3.6.1 Generated Battle Map Success Criteria

- Use OpenStreetMap data to generate structures.
- A generated battle maps structures walls must meet each other at 90 degree angles.
- A generated battle maps window and door assets must be applied over wall textures and not overlap each other.
- The exact placement of window and door assets in each structure must have variation.
- Structures must have variety of textures.
- Structures tagged as religious must use a different asset set.
- All internal and external flooring textures must appear in and outside of structures, respectfully.
- The Foundry Virtual Tabletop compatible .dd2vtt file must be generated ready to be loaded into a FoundryVTT game.
- The generation process must be faster for a Game Master then when creating a battle map manually.

3.7 Gantt Chart

Scheduling a programming project is difficult, it is rare that such a project gets completed on time. To best ensure that the same does not occur for this project a Gantt Chart has been created. This chart will aid ensuring all the critical tasks are completed, are completed in the correct order and that the project does not require extra development time. The Gantt chart is shown in Figure 3.

3.7.1 MoSCoW Analysis Coding Tasks

The Must, Should and Could Subsections of the MoSCoW analysis will be coded sequentially. However, the *Minimal Viable Product* milestone below must be met before the Should and Could sections can be started.

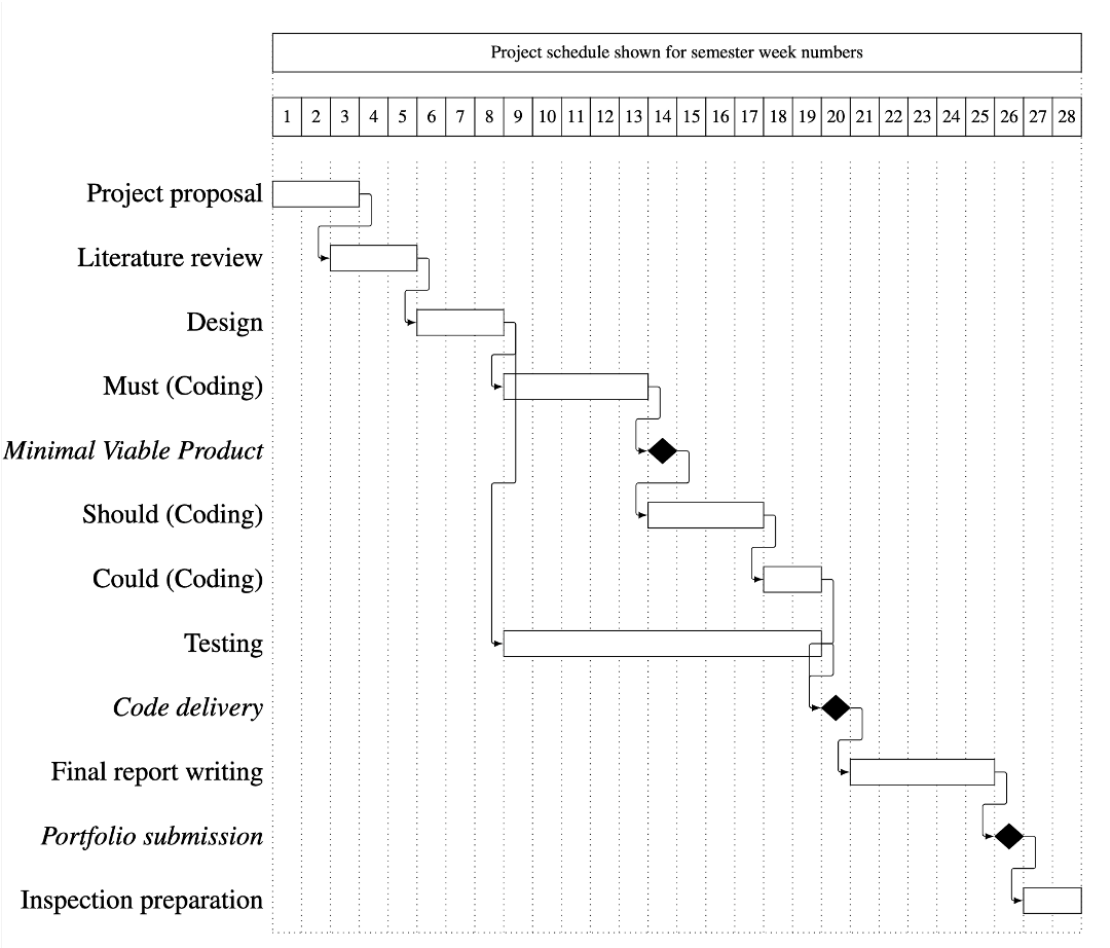

Figure 3: *Dungeonify* Project Gantt Chart

3.7.2 *Minimal Viable Product*

The Minimal Viable Product Task is a milestone that must be completed quickly to ensure that the program functions in its most basic form, thereby proving that the project works and can be developed further.

3.7.3 Testing, Code Delivery and Portfolio Submission

The Testing Task will be worked on throughout the MoSCoW Analysis Coding Tasks due to the Agile Scrum Development methodology being used for this project. The *Dungeonify* code will be made available alongside a report and will be submitted in the final week.

4 Implementation

The *Dungeonify* Project Implementation section comprises a full workflow of the program from the user's perspective. In addition, consideration of the four core python files used to generate the battle map files have been included with further detail of individual methods critical for the programs execution.

4.1 Full Workflow

Below is an order list of the steps taken to run the *Dungeonify* project. Figure 4 shows an overview of the core system steps, Figure 5 show how to load the *Dungeonify* battle map file into a Virtual Tabletop. An example OSM location is referred to as an Evaluated Location, and is described further in the Results and Evaluation Section of the report. This battle map features two selected structures, one being a religious structure.

1. The user launches the *Dungeonify* program.

2. The program presents the user with the OpenStreetMap website open in the default web browser at a standard location (see Figure 4(a)). The user can then use the website as normal to find a location they want to Dungeonify. The user clicks the *Share* button on the browser, copies the URL to the *Link* text box and close the browser.

3. When the web page closes, an input dialog opens allowing the user to paste the URL and click *OK*.

4. The user is then presented with the main *Dungeonify* application window, which shows the recreated data from the OpenStreetMap website (see Figure 4(b)). The user selects a number of structures, and in this example selects the two central structures. Note that the cyan coloured structure is a religious building. As the user clicks structures the *Structures Clicked* counter increases and decreases accordingly. Once satisfied, they click *File* in the menu system and *Draw Selected* (or use the keyboard shortcut *ctrl* + *d*).

5. The *Dungeonify* program initialises a new Dungeonify object which performs all the calculations required to generate the battle map. The first step is to redraw the view the user currently sees using the Draw Selected Structures Stage. This

process redraws the image with only the two structures they selected shown, as can be seen in Figure 4(c). The user clicks *Draw Rotated* (or presses *ctrl + d*) to perform the first rotation

6. The Draw Rotated Structures Stage rotates each structure around its centroid to force the longest node to the wall to rotate to become horizontal or vertical (whichever is nearest) relative to the grid. The rotation is applied to all nodes in the structure, making the whole structure rotate (see Figure 4(d)). This process is used to ensure that for the second rotation step, the program can start with a completed wall. The user can then click *Draw Resized / Rotated* or press *ctrl + D* again to run the next step.

7. The Draw Resized And Rotated Structures Stage performs rotation for every node in the structure consecutively, rotating it to make every wall section vertical or horizontal. It then fixes any nodes that did not rotate to the correct positions and the final result can be seen in Figure 4(e). The user can then click *Draw Dungeonify* or press *ctrl + D* again to create the final battle map.

8. The Draw Dungeonify Stage generates the final textured battle map image. The program loads textures from the Assets folder into lists. It then assign textures - a religious structure it uses a separate asset set list - randomly for each structure. The program first generates the flooring textures, then generates the internal wall, texturing it and then textures the exterior walls with wall assets, windows and a door. The Dungeonify stage, lastly, adds ascents such as pillars to the corners. The final generated battle map can be seen in Figure 4(f).

9. Throughout the Draw Dungeonify Stage the program was also generating the Virtual Tabletop representation of the data, which leads to the generation of the Foundry Virtual Tabletop compatible file (a .dd2vtt file) in this step. The data is formatted to be as efficient as possible, such that there are no unnecessary nodes added to the FVTT compatible file. It sets up wall, window and door boundaries before attaching the image into the file. The beginning of this Battlemap.dd2vtt file can be seen in Figure 5(a).

10. Figure 5(b) shows the user loading the Battlemap.dd2vtt Foundry Virtual Tabletop compatible file into a FoundryVTT game which uses the Universal Battlemap Importer module.

11. Lastly, Figure 5(c) shows the final battle map now loaded in FoundryVTT and ready to be used in a *Dungeons & Dragons* game.

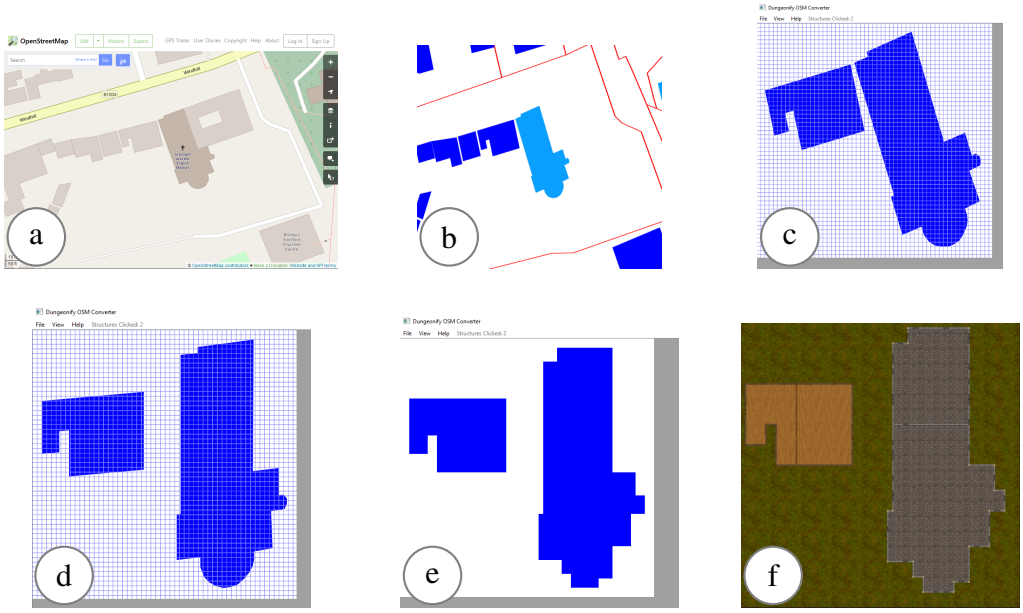

Figure 4: **System overview**: User selects an OpenStreetMap location (a), Dungeonify accesses structure node data via the API (b), selected buildings are redrawn (c), rotated (d), mapped to grid (e) and textured (f).

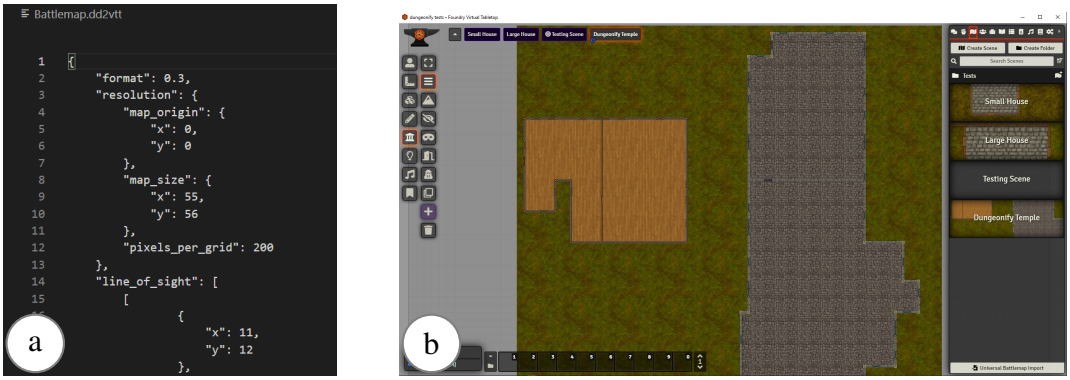

Figure 5: **Loading Dungeonify battle map into Virtual Tabletop**. (a) Example .dd2vtt battle map metadata file and (b) viewing Dungeonify battle map in Virtual Tabletop.

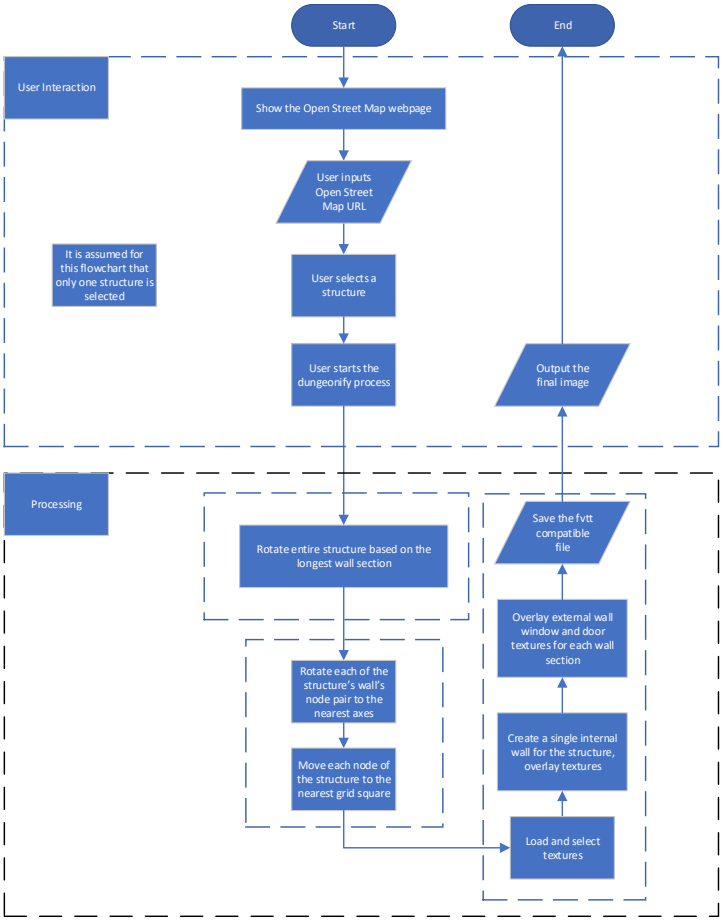

Figure 6: **System flowchart**: The operation of the program.

4.2 Key Components

The implementation of the *Dungeonify* project relies on four critical functions that are implemented in separate Python files: `coord_conversion.py`, `osm_factory.py`, `gui.py` and `dungeonify.py`.

4.2.1 `coord_conversion.py`

The `coord_conversion.py` file contains a class of the same name, which converts latitude and longitude coordinates into screen X and Y coordinates. This is done to flatten the slight curvature of the earth which stretches all polygons in the OpenStreetMap data, which would lead to incorrect calculations for the generation of structures in the

Dungeonify project. It calculates the minimum and maximum screen width and height to generate the image for the current screen resolution, ensuring that the representation of the OpenStreetMap location is of the same location as was on the OpenStreetMap website.

4.2.2 `osm_factory.py`

The `osm_factory.py` file contains the `GenerateOSM` class which is used to connect to and extract the geographical data - structures, roads and boundaries - for a specified OpenStreetMap URL, from the OpenStreetMap API. It first processes the input URL into a latitude and longitude and using the `coord_conversion()` class converts them into screen X and Y coordinates which the OSM API operates with. The connection to the OpenStreetMap API is handled through the following line of code, which requires these coordinates:

```
sector = myApi.Map(topLeft['lng'], bottomRight['lat'],
    bottomRight['lng'], topLeft['lat'])
```

The OpenStreetMap data for these coordinates must be scaled to fit on the *Dungeonify* PyQt5 image viewer. All the data from the OSM API must be converted in a ratio of 30 to 1, to shrink the map size down to screen size. This scaling factor was calculated using the average UK doorway width, approximately 2.5ft or 30 inches, and the average doorway on a printed *Dungeons & Dragons* battle maps width, 1 inch wide. This results in the division by 30 scaling calculation.

The OpenStreetMap data is recorded as a series of ways – collections of nodes in sequence – and placed into data structures sorted by their tag. At this level, each tag defines many aspects of the web mapping tool including structures, roads and terrain boundaries, among many other features. The most important data in the OSM data for the project are the ways forming structures. These ways also contain tags determining the structures function which are used to determine if the building is a religious structure. As each of the ways is processed, it is assigned a colour determined from the `TagData` dictionary, which colour codes them as is detailed in the table below. For example, if the next ways being processed had a tag of 'highway', it would be deemed a road and assigned the colour red. Equally, if the tag of the way after that was 'structure' its second tag would be checked and if it was 'place of worship' it would be assigned

the cyan colour, and otherwise the blue colour.

The PyCairo Library is used to draw the ways data on a new image, which is then saved and loaded into the GUI image viewer as an image file. This shows the recreated OSM location in the *Dungeonify* application.

4.2.3 `gui.py`

The GUI Python file contains the `QDungeonifyViewer` class which serves as the PyQt5 user interface for the *Dungeonify* project. The class presents the user with an image viewer which will display each stage of the *Dungeonify* Pipeline after they have each been completed in the Dungeonify class. The `QDungeonifyViewer` class also contains a number of extra methods, code to record the results for the Results and Evaluation Section and the unit tests specified in the Planned Testing subsection of the Design section.

The `QDungeonifyViewer` class is used to determine when structures are clicked before starting the battle map generation process. It uses a point-in-polygon algorithm, calculating the number of walls a ray traced from the point clicked to the edge of the map are crossed. An odd number means a structure was clicked and can be added to or removed from the list of selected structures.

When the completion of all stages of the *Dungeonify* pipeline and the generation of the textured battle map image is completed, the Foundry Virtual Tabletop compatible file is then created. This process uses a list data structure which was built as the `DrawDungeonify()` process textured the structures.

The code forming this data structure was written to ensure no unnecessary data are stored in the resulting FVTT compatible file. This has been done through temporarily storing the last texture's type added, and not adding the next added texture to the list if they are of the same texture type. If this process was not done then every unnecessary position stored would incur a slight but increasing performance decrease in the FoundryVTT software. When building the actual battle map .dd2vtt file, sets of wall definitions, window definitions and door definitions are injected into predefined strings lists. Each of the lists is then injected into the final string representing the full file format, in addition to adding the size of the battle map and the generated base64 encoded battle map image.

4.2.4 `dungeonify.py`

The *Dungeonify* Python file contains the `Dungeonify` class which provides the core functionality critical to the success of the project. With regards to the System Flowchart it handles all of the Processing stages. It is used to process the ways data from the `GenerateOSM` class and produce image files showing each of the four *Dungeonify* Pipeline stages. These stages are:

- `drawSelectedStructures()` - generates a new image of only the selected structures.
- `drawRotatedStructures()` - generates a new image of the structures rotated so the longest wall is horizonatl or vertical.
- `redrawForDegreesStructures()` - generates a new image of the structures with all walls horizontal or vertical.
- `drawForDungeonified()` - generates a new image forming the final textured battle map.

Each of these methods, with the exception of `drawSelectedStructures()` which has a simple operation, have increasingly detailed algorithms which are described in the subsections below.

Stage 2: `drawRotatedStructures()`

this first rotation stage is used to prepare every structure for the second rotation stage, `redrawForDegreesStructures()`, which requires one wall section - two consecutive nodes - to be horizontal or vertical relative to the grid. The algorithm first finds the longest wall section and calculates the angle between it as a vector and the horizontal or vertical plane, whichever it is closest to. Every node in the structure is then rotated by this angle relative to the centroid of the structure, the structure often being an irregular polygon. A new image is created with each of the roated nodes on, ready to be displayed in the GUI.

Stage 3: `redrawForDegreesStructures()`

The second rotation stage is significantly more complex. The algorithm needs to rotate and reposition every node so each fits to the grid, as is required in the Success Criteria.

The algorithm performs five critical data structure manipulation processes and calculations to conform each structures node set of data into a standard form:

```
self.removeDuplicateNodes(structure)
self.rotateAllNodes(currentStructure)
self.simplifyNodes(currentStructure)
self.fixNegativeNodes(currentStructure)
self.fillNodeGaps(currentStructure)
```

For each structure the following is conducted:

1. `self.removeDuplicateNodes(structure)` - Every pair of consecutive nodes in the structure is compared. If they are not equal then the first of those nodes is added to a new structure list. This step is required to ensure there are no irregularities preexisting from the OpenStreetMap data remain in the structure's node data.

2. `self.rotateAllNodes(currentStructure)` - Every node in the structure is then rotated by the angle required to make the wall horizontal or vertical - whichever is closest. Before this new node is added to the new structures node list, it is rounded to the nearest inch square of the grid. This process ensures the structure retains some of its characteristic shape even after all the nodes which formed it have been rotated.

3. `self.simplifyNodes(currentStructure)` - The next step simplifies groups of three nodes, which together form a straight line, into a pair of two nodes. The rotation step can make multiple consecutive nodes all rotate and then round to the same horizontal or vertical line. To ensure all the node structures are as efficient as possible, the central node of each such case is removed.

4. `self.fixNegativeNodes(currentStructure)` - The penultimate step is to ensure that all the rotated nodes are still within the bounds of the image. To do this, every node in the structure is increased by the absolute value of the largest negative X and Y coordinates in the structure's nodes. This process ensures that the generated image will not have the structures drawn appear off screen.

5. `self.fillNodeGaps(currentStructure)` - The last step of the resize and rotate algorithm fills in gaps between consecutive nodes. Every pair of consecu-

tive nodes must, at this stage, have either an equal X coordinate or and equal Y coordinate. This ensures that the next wall section is at a 90 degree turn to the one before it. Every case is checked, including the last node and first node of the data structure and if required an additional node is added between them.

The algorithm then generates the new image file showing the now correctly positioned nodes for each structure.

Stage 4: `drawForDungeonified()`

The battle map generation stage is used to form the final battle map image from a collection of loaded in *Dungeons & Dragons* artist textures. The algorithm must conduct a number of steps to accomplish this goal. It must load the assets it will use for both the religious and non-religious structures. It must then generate the flooring texture for the battle map, followed by generating a single internal wall, using the `generateAssetsBetweenNodes()` procedure, for each structure. It then generates all the external walls of each structure using the same procedure and finally adds ascent details by positioning pillar textures to the corners of each structure.

Assets: An important characteristic of the code used in loading assets is that the *Dungeonify* project is designed to texture battle maps using any standardised set of artist's textures. This was implemented to help mitigate any problems arising as was outlined in the Technology Risks section in the Design above. The textures which the program accepts are separated into specifically named folders in the root asset folder which the program loads into lists when running the `drawForDungeonified()` code.

The method used to load texture assets relies on a single number determining the resolution of all texture images. The `self.assetSize = 200.0` property is used to establish the pixels per inch ratio used by the Forgotten Adventures assets for the final battle map. All input textures loaded from the asset library are scaled to fit on this resolution size. To alter the program to use assets all scaled for 70 pixels per inch, as is the standard for Roll20 battle maps, the `self.assetSize` value would need to be changed accordingly. The code will then operate using the new asset set.

Another important assumption about the assets is that they are all in the same orientation. All the walls, windows, window sills and doors must appear horizontal in the asset folder. This is to ensure that they do not overlap incorrect sections of the walls.

The flooring texture is generated first, which uses image masking to create sections of

repeating interior flooring textures for each structure's floor plan, set onto the base grass texture. After this, two dictionaries are initialised, with one containing all the textures which any of the normal structures will use and the other containing the set of textures which religious structures will use.

Internal Wall Generation: The generation of an internal wall was an extra feature implemented to increase the variation between each structure in the final battle map.

The code used to produce the internal wall relies on an alteration to the Recursive Division algorithm often used to generate mazes among other structures. It will only perform the room separation technique if the overall area of the structure is large enough to accommodate it, which was left at a base value of 50 units squared. The algorithm first determines the assets that the internal wall will use randomly and determines if the structure is a religious building or not.

The new internal wall is built halfway along the wall section which has the longest length, which is done to maximise the chances that this will divide the structure into two relatively evenly sized rooms. for whichever orientation the wall is in the starting node and the iterator are determined. The ending node is calculated by counting incrementally from the starting node until a wall section on the opposite side of the structure is found. This data are then entered into the `generateAssetsBetweenNodes()` method, which will be described in the External Wall Section below, that places the assets and adds a door randomly on it. The last thing it does is determine the two new rooms of the structure, which with further development could be used to set up a second phase of internal wall construction enabling more than one pair of internal rooms to be generated in each structure.

External Wall Generation: The generation of all the exterior walls is handled in the `generateExterior()` method. There are a number of steps that must be accomplished in order to layer textures correctly:

1. The structure is assigned randomly determined wall, window, window sill, door and pillar textures from texture dictionaries. Which dictionary depends on if the structure is tagged as a religious building, as mentioned in the Assets Section above.

2. The next step is to find a location on the one of the walls of the structure to place the main door, discounting positions that are too close to the corners of the structure. The door position is randomly determined.

3. This step sets up the arguments passed into the `generateAssetsBetweenNodes()` method. There is no easy calculation for determining if the structure, originally from the OpenStreetMap API, was built clockwise or anticlockwise. This means that the method must be able to handle generating textures in both directions. To this end, the difference between the start node and the next node is used to calculate the iterator value which determines the direction the structure's nodes were built in. The difference between each node's X or Y coordinates determines if the current wall is horizontal or vertical. All this data is passed into the method which layers the textures and builds the data structure used to generate the Foundry Virtual Tabletop compatible file.

4. the `generateAssetsBetweenNodes()` method serves to create every straight wall section between any two nodes for the *Dungeonify* solution.

`generateAssetsBetweenNodes()`: This algorithm is used to enable the creation of wall sections containing window sills, windows and doors positioned with regards to the Success Criteria. The algorithm ensures that regardless of the orientation of the wall section or the direction the node data was formed for it (clockwise or anticlockwise), the wall section will always build successfully. The algorithm's operation performs the following steps:

1. If the wall section is vertical then the individual assets used to texture it will need to be rotated 90 degrees.

2. If the current wall section must add a door texture, then the door is given a randomly determined index between the start and end positions, exclusive in both cases, of the wall section. The door cannot be given the index at either end of the wall or an intersection of textures will occur.

3. The coordinate which does not change is set. If the wall section is horizontal this must be the Y value whereas if the wall section is vertical then this must be the X value.

4. The loop steps through every node position between the start node and end node stepping by the iterator value to ensure it is generating in the correct direction. It checks that the current node is not at either end of the wall section as this space must be set as a wall asset to avoid texture intersection. Otherwise, if this index

must be the door it textures it or given a 33.3% chance a window is generated, which positions a window sill and then a window above it. The remaining 66.6% chance is given to constructing additional wall pieces in the wall section. This algorithm functions successfully for both horizontal, vertical and clockwise, anti-clockwise wall sections.

5. Throughout the above process after every asset is positioned, a stored `lastGenerated` variable is used to record the asset's type. This is used to form the Foundry Virtual Tabletop compatible file data structure by ensuring no unnecessary data are added for continual sections of the same asset type.

4.3 Testing

A number of unit tests were also created in the `gui.py` file to ensure the structures rotated by the `redrawForDegreesStructures()` method always fit to the grid. This was essential as the algorithm which layers the textures requires all wall sections to be either horizontal or vertical, as was required in the battle map Success Criteria. These unit tests compare each structure's nodes for specifically selected buildings for locations to what successfully recorded node structures for them are.

5 Results and Evaluation

Several comparisons are made to evaluate the effectiveness of the *Dungeonify* project against the Stated Success Criteria from Section 3. An evaluation dataset is constructed that contains a wide range of geographic locations. The full *Dungeonify* process from OSM map to output battle map file is first evaluated qualitatively. Next, a data consistency check is performed to check whether all OSM building structures appear in the *Dungeonify* application. *Dungeonify* is then tested on a separate location that contains challenging curved structures. The quality of the texture generation module is next evaluated, to test that all Dungeonfied buildings appear have sufficient variety. Finally, the running time of the full *Dungeonify* process and individual components are evaluated.

ID	Location	URL
1	Bishop's Stortford, UK	map=19/51.87166/0.15663
2	Norwich, UK	map=19/52.62956/1.26577
3	Bristol, UK	map=19/51.48076/-2.63602
4	Orléans, France	map=19/47.91488/1.93173
5	Tokyo, Japan	map=19/35.68263/139.72467

Table 4: **Evaluated Locations**. Access via https://www.openstreetmap.org/{URL}

5.1 Constructing an evaluation dataset

Five distinct locations were selected from the OpenStreetMap website to be representative of diverse user selected places. These include three locations in the United Kingdom; a medium-sized town and two cities. Another location was selected in France, due to France being used to test the accuracy of the OpenStreetMap dataset which was discussed in the Related Work Section. The last location selected features a population dense city in Japan, providing a strong contrast to the cities selected in the United Kingdom. These are detailed in Section 5.4.

5.2 Full Workflow Evaluated

The first consideration is to qualitatively inspect the full *Dungeonify* process. Figure 7 shows the various *Dungeonify* stages, tested on the 5 locations in the evaluation dataset. This serves as an excellent example location to show the capabilities of the program in generating the final *Dungeons & Dragons* battle map and Foundry Virtual Tabletop compatible file for a number of requirements given in the Stated Success Criteria.

To determine how successful the produced battle map image is, the Stated Success Criteria must be considered. Note that all of the walls of each structure meet at 90-degree angles and all the window and door assets have been both randomly determined and randomly positioned, non-overlapping on each of the walls. All of the floor textures – from the grass exterior to each of the structures interiors – have also been randomised. The structure tagged as religious has used textures randomly determined from a speciality set of assets used to represent them as distinct stone structures. In addition, as shown in Figure 5, the Foundry Virtual Tabletop compatible file was successfully generated and loaded into FoundryVTT.

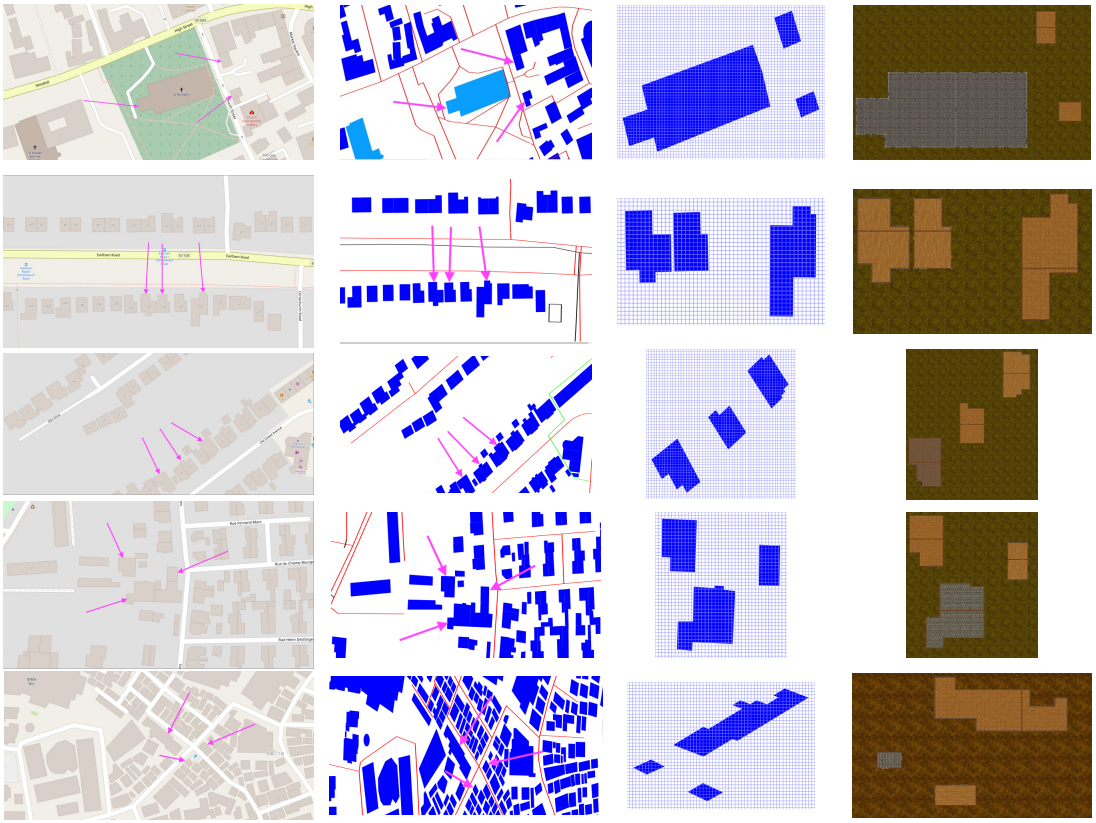

Figure 7: **Qualitative Results**: *Best viewed zoomed in*. Dungeonify run on Evaluation Locations. *Columns*: (a) Input OpenStreetMap location, (b) loaded into Dungeonify buildings selected (pink arrows), (c) buildings mapped to grid and (d) final rotated and textured output.

Since *Dungeonify* successfully renders all of these assets and correctly forms the FoundryVTT .dd2vtt file, the program successfully meets the majority of the battle map generation requirements laid out in the Success Criteria.

5.3 OpenStreetMap Data Consistency Check

This section checks that the data extracted from the OpenStreetMap and redrawn by *Dungeonify* are consistent to satisfy the first point in the Success Criteria.

This is verified by connecting to the OpenStreetMap API with each of the five Evaluated Locations in the evaluation dataset. The input URL was entered for each of the

Evaluated Locations, shown in Figure 7(a). Figure 7(b) shows each Evaluated Location after it has been found and redrawn by *Dungeonify*.

As the figure shows, *Dungeonify* was able to successfully recreate the majority of the structures existing in the OpenStreetMap API dataset. It was also able to successfully colourise the data: roads (red), boundaries (green), non-religious structures (blue) and religious structures (cyan).

The conversion process is not infallible however, as can be seen in Figure 7(a), row 2. Here, one of the structures has been drawn as a hollow black lined rectangle. An investigation was conducted into the root cause of this issue, and it was found that the original OpenStreetMap data had an inaccuracy. Since the OpenStreetMap website can be edited by any user, these errors can arise but are thought to be rare. Another example of the lack of standardisation in the OpenStreetMap data required an update to the code in the `generateAssetsBetweenNodes()` method which generates the battle map wall sections. The code revision now accepts lists of structure nodes formatted in both clockwise and anticlockwise directions, as discussed in the Implementation Section.

Overall, the use of OSM data for the *Dungeonify* project has been successful, and fully meets the requirements set out in the stated success criteria. The fact that the project is able to use structures from around the world enables it to better meet the needs of a Game Master to create their desired layout, storyline and theme.

5.4 Complex Structures Evaluated

From above, the *Dungeonify* project is able to handle rectangular structures, but how it performs with especially complex structures is yet to be considered. Structures with curved or circular wall sections – which are not usually considered for creating *Dungeons & Dragons* battle maps because of the extra time commitment to create attaching walls – are in fact able to be generated using the *Dungeonify* program.

The *Dungeonify* `redrawForDegreesStructures()` algorithm enables automatically handles fitting such wall sections on the grid, often retains the geometric characteristics remarkably well. To evaluate this, a new location has been considered below that features multiple curved buildings and a rectangular structure which tapers at one end.

The OpenStreetMap location for Brussels is shown in Figure 8(a). Figure 8(b) shows the *Dungeonify* application containing the four complex curved structures selected for

ID	Location	URL
6	Brussels, Belgium	map=19/50.81823/4.39470

Table 5: **Complex Location**. A complex location with curved structures. Access via https://www.openstreetmap.org/{URL}

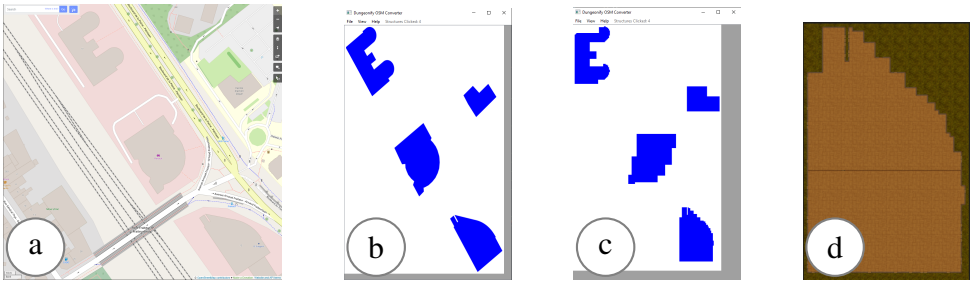

Figure 8: **Performance analysis on complex scene**: Evaluation on input with curved buildings. (a) OSM input, (b) Selected structures, (c) Rotation stages, (e) Final battle map, zoomed on the 4th structure.

this experiment. The first rotation step, as normal, rotates each structure such that his longest wall becomes horizontal or vertical, as is shown in Figure Figure 8(c). After the second rotation step, the `redrawForDegreesStructures()` algorithm converts the curves of each structure have to horizontal or vertical walls but still retains the characteristic shape of the original structure. The bottom right structure in Figure 8(d) is the best example of this as it retains the shape of the curving wall section. The top-right tapered structure has been simplified into a more rectangular shape and the top left structure, while not perfectly representative of its previous form, has retained elements of the half-circle sections. The last structure, in the center, has been changed more dramatically than the others. The cause of this is due to the nodes used to represent it. The structures which have curves are better represented if their curves are made using many nodes. This structure uses too few to represent the curve.

The effectiveness of the project's battle map generation is further solidified when looking at the completed curved structure battle map in Figure 8(e). All the wall sections forming the original curve follow the rules laid out In the Stated Success Criteria, with the most critical one being that windows and doors do not intersect other textures which in this case means the corner pillars.

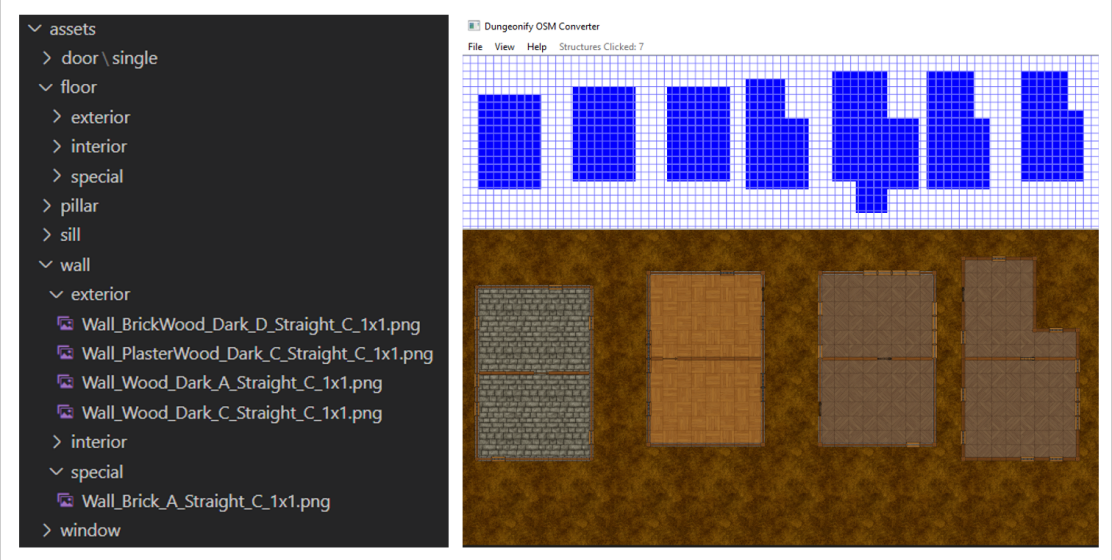

Figure 9: **Texture generation**. *Left*: *Dungeonify* asset folder, *Right, Top*: Four structures selected from a street. *Right, Bottom*: The structures generated with variation in textures across wall, window sill, window and door assets.

5.5 Evaluating the Variety of Generated Textures

In the Full Workflow Evaluated Section above, the differences between texturing the religious and non-religious structures is evident, however the variation in textures possible for non-religious structures has yet to be seen.

Figure 9 shows an example of selecting 4 similar buildings from a single street for texturing. The textures themselves are stored in the Asset folder on disc in a hard coded file structure. New textures can then be added to the correct folders with the program automatically and randomly selecting from them every time it is executed. It can be seen that the generated *Dungeons & Dragons* external wall, window sill, window, door, pillar and flooring sections of the battle map structures have been created with a variety of textures. These results show that the variation aspect of the stated success criteria has been met.

5.6 Running Time of Full Workflow

The previous evaluation sections show that *Dungeonify* is successful at completing various objectives in the Stated Success Criteria. This and the following sections serve to

Location ID	1	1	2	2	3	3	4	4	5	5
Struct ID	1	2	1	2	1	2	1	2	1	2
Time (s)	2.84	3.25	3.30	2.59	2.40	2.60	1.17	**12.2**	3.25	3.03

Table 6: **Running time per structure**. Running time for 4-node structures in the Evaluated Locations.

satisfy the final criterion: checking the time efficiency of the full *Dungeonify* process and of various components, testing on the 5 Evaluation Location above. The hypothesis for these results is that as the number of selected structures, number of nodes and overall map sizes increases, so will the time required to process it. However, of most importance is to check that the *Dungeonify* battle map generation process runs faster than the time it would take for a Game Master to design the map manually.

5.6.1 Running time per 4-node structure

Table 6 shows the running time per 4 node structure in the Evaluated Locations. The results show a similar pattern for generating the final battle map within 5 seconds. However, there was a single anomaly for experiment Location ID 4, Structure ID 2 which took 12 seconds to generate. The suspected reason for this anomaly stems from the size of the selected structure which was approximately a factor of 3 times as large as the other randomly selected structures. The stated hypothesis is challenged for this case, as this anomaly may suggest the running time of *Dungeonify* is not directly related to the number of structures selected.

5.6.2 Running time when selecting multiple structures

Table 7 analyses the effect of increasing the number of structures selected for each Evaluated Location, from one to four. The table shows significant variance in results further concluding that the original hypothesis may be incorrect. Table 7 shows that the number of selected four node structures does not perfectly determine the time *Dungeonify* will take to produce a battle map, as is shown by the averages for both 3 and 4 selected structures.

Time	Number of Structures			
Exp ID	1	2	3	4
1	2.84	6.44	12.4	9.86
2	3.25	10.3	10.7	16.1
3	3.30	5.75	15.3	60.4
4	2.59	4.20	9.92	6.51
5	2.40	5.35	14.2	12.4
6	2.60	22.8	14.9	8.77
7	1.17	6.04	10.7	12.3
8	12.2	9.00	54.9	9.89
9	3.25	4.17	6.71	16.2
10	3.03	5.71	12.1	12.6
Avg	3.66	7.98	16.2	16.5

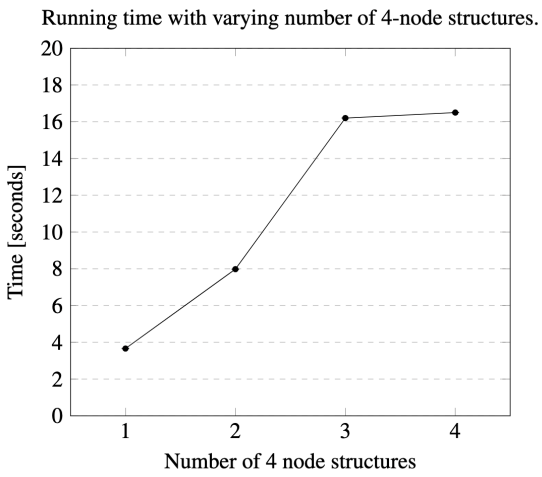

Table 7: **Multiple structures**. Running time for selecting multiple 4-node structures per Evaluated Location.

5.6.3 Running time when selecting more nodes per structure

Table 8 shows the effect of selecting structures with more nodes. A set of four 4-node, 6-node, 8-node and 10-node structures were identified. The table shows the running time required to generate each set. The first interesting result from this table is that the 10-node structures average running time is significantly higher than all the others. The cause of this may be when selecting structures of 4-node or 6-node the average size of such structures is small. Whereas structures which are 8-node or 10-node are much larger.In addition, as seen by the average of 4-node being higher than both 6-node and 8-node averages, the number of nodes appears not to provide a strong relationship to the running time of *Dungeonify*. On further investigation, the cause of this could be due to the number of 4-node structures often being higher, and therefore more spread out, in each Evaluated Location. Because the structures are selected randomly, there is a higher chance of such structures being selected far apart, which would lead to a larger total battle map area. The stated hypothesis may need to be altered as either area calculation above may more closely relate to the running time.

Time (s)	Nodes per Structure			
Exp ID	4	6	8	10
1	7.10	7.59	9.16	321
2	33.9	12.0	11.7	427
3	19.9	16.7	18.4	651
4	64.2	22.2	29.1	369
5	26.3	12.9	14.7	301
Avg	30.3	14.3	16.6	414

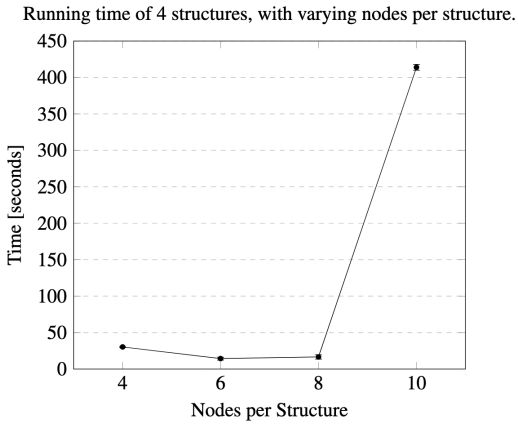

Table 8: **Structures with more nodes**. Evaluating the effect of selecting structures with more nodes. Tested on Evaluation Location 2.

5.6.4 Effect of increasing the battle map size

Table 9 analyses the effect of plotting the running time against the total battle map size for the previous data. Table 9 shows the results. As can be seen there is a clear relationship between the total map size and the running time forming a linear to exponential increase.

Note, battle maps generated with a total map size larger than 7000 square inches cause an Out Of Memory exception, crashing *Dungeonify*. After investigation, the cause was the initial generation of the grass texture base image. The memory required to initialise such as massive image exceeded the maximum memory allocated to the program. This issue is not critical however, as FoundryVTT would have difficulty loading a battle map file of this size, especially from a Virtual Tabletop compatible file due to all the extra vector information stored with the image.

Given the results shown in Table 9 the hypothesis appears to require changing. To ensure this new hypothesis is complete another test analysing the cases above against the running time of the System Components of *Dungeonify* should be required.

5.7 Running Time of System Components

Additional experiments have been conducted on the individual *Dungeonify* components. To ascertain how easily the program could be scaled to produce larger battle maps more

Exp ID	Map Size	Time (s)
1	518	6.85
2	1504	19.9
3	2457	29.6
4	2592	23.2
5	5088	93.4
6	6384	99.4
7	6624	129

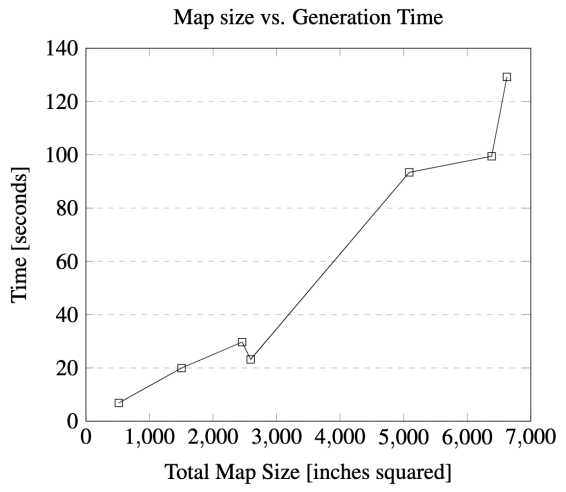

Table 9: **Increasing map size**. Evaluating the running time to generate four 4-node structures at larger map sizes (in square inches) for Evaluated Location 1.

quickly, the six core processes have been timed when testing Evaluated Location 1.

5.7.1 Individual Processes Key

For readability of the following graphs acronyms are used to represent each of the processes:

Process	Acronym
OpenStreetMap	OSM
Selection Structures	SS
First Stage Rotated Structures	RS
Second Stage Rotated Structures	R&RS
Dungeonify	DS
Virtual Tabletop File	VTT

5.7.2 Varying number of 4-node structures

Table 10 shows the average running time of individual *Dungeonify* processes compared to the average battle map size for them. The cumulative Table 10 graph shows that the 'DS' - Dungeonify process requires an exponentially larger running time to process. In addition, it shows that the the average map area influenced the running times of pro-

Process	Avg. Running Time (s)			
Num. Structs	1	2	3	4
Map Size	316.6	2259	5709.4	3283.4
OSM	4.33	4.08	4.42	4.55
SS	0.03	0.05	0.10	0.07
RS	0.01	0.05	0.09	0.08
R&RS	0.01	0.04	0.09	0.07
DS	3.14	15.3	112	44.6
VTT	0.300	1.14	1.51	1.47

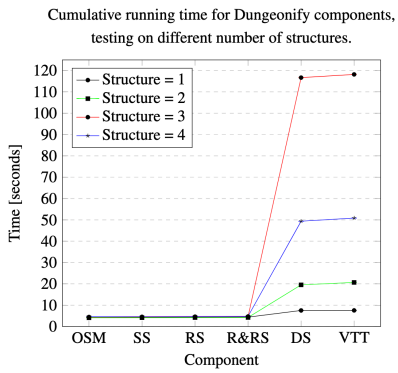

Table 10: **Avg. running time of different components, 4-node structures**. Running time of different system components, averaged over 5 different choices of structure set.

cesses more than the number of structures did. The reason for this was investigated, with the cause again returning to the generation of the initial base floor texture in addition to adding to it.

5.7.3 Varying number of any-node structures

To make absolutely certain there are no outliers to this new hypothesis, A final experiment was conducted comparing 1 structure with 4 structure battle maps shown in Table 11. In each case the structures selected was completely random with no limit on the nodes in each structure. In addition, the locations for each battle map was randomised around the world.

The results shown in the Table 11 graph conclude, along with all previous running time experiments, that the hypothesis must be changed. The new hypothesis is that the running time is most affected by the total battle map area being generated. The higher number of structures selected simply dictates that a larger battle map area is more likely to be used.

Process	Avg. Time (s)	
Num. Structs	1	4
Map Size	1912.6	4682
OSM	4.86	5.17
SS	0.05	0.09
RS	0.05	0.08
R&RS	0.05	0.09
DS	16.0	75.0
VTT	1.18	2.14

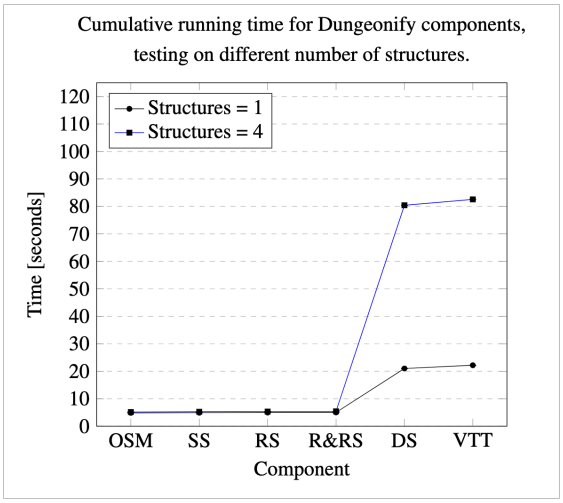

Table 11: **Avg. running time of different components, any nodes**. Running time of different system components, averaged over 5 different choices of structure sets with different numbers of nodes.

6 Conclusion

The Evaluation results in *Dungeonify* bring evidenced to the fact that the requirements laid out in the Stated Success Criteria, which were informed from the most critical aspects of the MoSCoW Analysis, have been met. *Dungeonify* has been shown to generate *Dungeons & Dragons* battle maps using the OpenStreetMap API data and texturing using randomly determined textures.

Must: The connection to the OpenStreetMap API and the use of the data to create a basic battle map and create the Foundry Virtual Tabletop compatible file was achieved for the MVP deadline.

Should: The implementation of a basic user interface enabling the selection of one or more structures in a recreated OSM location was achieved. Constructing algorithms to correctly rotate and reposition each of the structures to the grid provided the key requirements of battle map creation. In addition applying textures based on the tag each structure had enabled the creation of unique structures in battle map generation.

Could: Two extra features were able to be implemented once the MVP was built; basic image processing tools and randomised textures for each structure gave *Dungeonify* a strong degree of variability for a Game Master's themes. The addition of using Digital Elevation Models and Digital Surface Models to aid in the generation of scenery proved

to be outside the scope of the project, due to the intensive configuration these models would require to be coupled to the OSM data. The DEM and DSM feature was instead replaced by the implementation of a feature improving the design of structures by generating a single internal wall. This provides a new layer of complexity to generated battle maps by limiting the vision of player characters, which is also significantly more than any additional scenery that a DEM or DSM would provide.

Will Not Have: Care has been taken throughout the development *Dungeonify* to avoid implementing unnecessary features, such that the project scope remained feasible. The theme has remained in the *Dungeons & Dragons* system for strictly 2-dimensional battle maps. Also, the GUI interface has intentionally remained basic to ensure that it didn't start to look and act like a full *Dungeons & Dragons* battle map editor.

Overall: The Results and Evaluation section show that *Dungeonify* successfully generates battle maps in line with the Stated Success Criteria. The solution uses the OpenStreetMap datasets to generate a number of structures fitting to a 90 degree angle grid and applying randomly determined artist textures formulating the final battle map. The program can handle texturing specialty structures, namely religious buildings, which used their own asset set. In addition, the running time comparisons show that for any input set, the time to generate the battle map will always be less than the time required for game master to create the battle map manually. It also shows that the limiting factor for being able to scale the program to produce battle maps of a larger size is caused by an outside factor; the maximum memory allocated to the program. The battle maps of these exceedingly large sizes, generated for a whole town for example, would not serve the game master very well as FoundryVTT cannot maintain performance at this file size. Having successfully met all the Must and Should features, in addition to several notable Could features, of the Moscow Analysis the *Dungeonify* program is deemed a successful project. The battle maps themselves each meet the Battle Map Stated Success Criteria, which means the output generated by *Dungeonify* has also been a success.

7 Future Work

A redesign of the user interface to simplify the use for non-expert users could be done. The current GUI, implemented for the minimum requirements, could add extra features dictating how components of the project generate battle maps, by linking with existing code. For example, the project texture set assumes each inch by inch grid square is a 200

by 200 pixel image file. An improvement would be to add a text box, so the resolution of the grid and assets can be user specified.

As seen in the `dungeonify()` class, the generation of the interior wall uses a modified recursive division algorithm. An improvement to the project could be to enable this method to recursively generate rooms for each structure.

Lastly, the author is aware of a bug that occasionally occurs for structures with jutting out sections of the floor as shown in Figure 10. The `redrawForDegreesStructures()` algorithm completely separates these sections from the internal structure. Then the single entrance door may be added to them, preventing access to the rest of the structure.

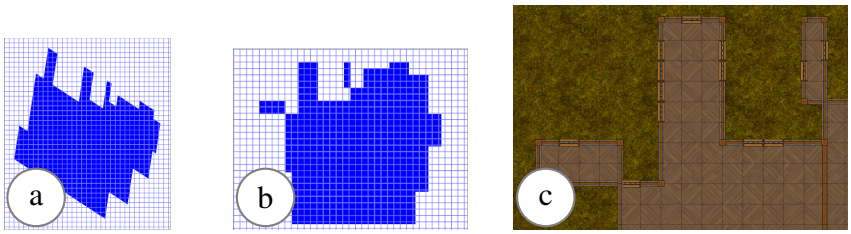

Figure 10: **Failure case**: Dungeonify loads a complex structure (a), rotates (b) and textures it (c). Unreachable areas are created (b) which are inaccessible (c).

References

Almahairi, A., Rajeshwar, S., Sordoni, A., Bachman, P., and Courville, A. (2018). Augmented CycleGAN: Learning many-to-many mappings from unpaired data.

Camozzato, D., Dihl, L., Silveira, I., Marson, F., and Musse, S. R. (2015). Procedural floor plan generation from building sketches.

Carey, B. (2019). Procedural forest generation with l-system instancing.

Emilien, A., Bernhardt, A., Peytavie, A., Cani, M.-P., and Galin, E. (2012). Procedural generation of villages on arbitrary terrains. *The Visual Computer*, 28.

Girres, J.-F. and Touya, G. (2010). Quality assessment of the french openstreetmap dataset. *Transactions in GIS*, 14(4):435–459.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative adversarial nets. In Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N., and Weinberger, K., editors, *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc.

Grabler, F., Agrawala, M., Sumner, R. W., and Pauly, M. (2008). Automatic generation of tourist maps. *SIGGRAPH 2008*, pages 100:1–100:11.

Hahn, E., Bose, P., and Whitehead, A. (2006). Lazy generation of building interiors in realtime.

Haklay, M. (2010). How good is volunteered geographical information? a comparative study of openstreetmap and ordnance survey datasets. *Environment and Planning B: Planning and Design*, 37(4):682–703.

Hiasa, Y., Otake, Y., Takao, M., Matsuoka, T., Takashima, K., Carass, A., Prince, J. L., Sugano, N., and Sato, Y. (2018). Cross-modality image synthesis from unpaired data using cyclegan.

Kelly, T. (2021). *CityEngine: An Introduction to Rule-Based Modeling*, pages 637–662. Springer Singapore, Singapore.

Linden, R., Lopes, R., and Bidarra, R. (2014). Procedural generation of dungeons. *Computational Intelligence and AI in Games, IEEE Transactions on*, 6:78–89.

Lopes, R., Tutenel, T., Smelik, R., de Kraker, K. J., and Bidarra, R. (2010). A constrained growth method for procedural floor plan generation.

Mawhorter, P. and Mateas, M. (2010). Procedural level generation using occupancy-regulated extension. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, pages 351–358.

Mehta, H., Kanani, P., and Lande, P. (2019). Google maps. *Int. J. Comput. Appl*, 178(8):41–46.

Tolkien, J. (1954). *The Hobbit The Lord of the Rings*. Houghton Mifflin Harcourt.

Wu, W., Fu, X.-M., Tang, R., Wang, Y., Qi, Y.-H., and Liu, L. (2019). Data-driven interior plan generation for residential buildings. *ACM Transactions on Graphics (SIGGRAPH Asia)*, 38(6).