Investigating Flood-Fill algorithms for settlement generation using a diagram to dictate its layout.

By Jack Hopkins [180246647]

Project Supervisor: Dr Gary Ushaw

# Abstract

With the market and demand for open world video games increasing, the ability to create and prototype settlements for these games in a shorter timeframe has been desired. To retain creative control over the generation, a diagram (bird’s eye view) of the settlement is inputted with different colours representing different sectors. These sectors are then mapped out within multiple data-structures using Flood-Fill algorithms. This dissertation will examine the prototype created and compare the algorithms implemented within in it.

Contents

[Abstract 1](#_Toc69742279)

[1 Contents 2](#_Toc69742280)

[1 Introduction (5% ~400 - 600 words) 3](#_Toc69742281)

[1.1 Original Aims & Objectives 4](#_Toc69742282)

[1.1.1 Objectives 4](#_Toc69742283)

[1.1.2 How the objective and Aims have Changed 5](#_Toc69742284)

[1.2 Project Outline 5](#_Toc69742285)

[2 Background Technical Material (10% ~800-1200 Words) 5](#_Toc69742286)

[3 Implementation (What was done, and how) (40% ~3200 -4800 words) 5](#_Toc69742287)

[3.1 Stack-based Recursive implementation (Four-Way) 5](#_Toc69742288)

[3.1.1 Stack Storage 5](#_Toc69742289)

[3.1.2 Queue Storage 5](#_Toc69742290)

[3.2 Walk-Based Filling 6](#_Toc69742291)

[3.3 Graph-Theoretic Filling 6](#_Toc69742292)

[3.4 Span Filling 6](#_Toc69742293)

[3.5 Neighbour Checking Algorithm 6](#_Toc69742294)

[4 Results and Evaulation (30% ~2400 – 3600 Words) 6](#_Toc69742295)

[5 Conclusion (10% ~800 – 1200 Words) 6](#_Toc69742296)

[6 Form/References (5%) 6](#_Toc69742297)

[7 References 6](#_Toc69742298)

# Introduction (5% ~400 - 600 words)

In the current landscape of AAA video game releases, there are many that are 3D open-world titles. [1] Such as, Skyrim, Witcher 3, GTA, Legend of Zelda: Breath of the Wild and more. This also includes smaller studios and indie developers. However, the bigger these worlds get, the more manhours is needed. This can result in the now infamous crunch culture, as seen with CD Projeckt Red’s *Cyberpunk2077,* [2] as the game’s scope became so large that on launch day, there were a plethora of bugs and glitches. [3]

This is just one example of the classic “Quantity vs Quality” conflict and one area where this conflict is highlighted in Open World games is in Settlement Generation (e.g. towns, cities and villages). Some developers have dealt with the issue by having randomly created terrain, like in Minecraft’s Villages [4]. Others, researchers have tried to address using Perlin Noise [5], fractals, L-systems, tiling system, Voronoi texture basis [6] [7], and wave function collapse. [8]

A picture containing text, sign

Description automatically generatedBut what if we wanted to, instead of purely random generate settlements, create settlements from a plan/diagram that would add constraints and structure. (Fig. 1) This would allow for Designers to have more control over the world they are creating. The aim here, is hit a middle ground where developers can still create interesting, unique, sensible settlements while also speeding up the process, so not so as many man hours are needed.

Figure . City diagram proof of concept.

Furthermore, this addresses a need within smaller/independent video game studios too, who have fewer resources and could be used to quickly prototype and experiment with town layouts.

This project will go beyond existing works by focusing on specifically video game settlement creation. Not attempting to use video game graphical engines to create a city to simulate real world phenomena, [9] or settlement creation in a specific video game and by using a predesigned map to guid the layout of the city not a randomly generated one [6]. I am also personally motivated by this project because I create video games in my own time. [10] Thus, finding a way to lighten possible workloads for myself, and other Game Dev Hobbyists in the future, is exciting!

Over the course of the project I have begun to focus in specifically using Flood-Fill algorithms to map out the sectors of the map. The 3 different Flood-Fill Algorithms (FFAs) that were used were:

* Four Way [Stack-Based Recursion] (4W).
* Walk-Based Filling [Fixed Memory Method] (WBF).
* Span Filling (SF).

And one algorithm of my own creation: Neighbour Checking (NC).

These models will be comparing in 3 different categories:

* Memory use.
* Initial settlement creation speed.
* Scalability

The reason for choosing variations of the FF algorithm was because FF is designed to determine an area of connected nodes with a matching attribute. It is often used in the File or “Bucket” tool (e.g. Paint, Photoshop, Etc.). [11] This meant that they were easily adapted to produce a data-structure of Vector2 (Coordinates) for each shape on the map.

**How did it turn out? (This will be added once I’ve completed my testing).**

## Original Aims & Objectives

### Objectives

#### Research Settlement Design from 3 different eras of history. ???? (MAYBE DELETE IDK)

Developing an understanding of settlement layouts over the ages will allow me to program for more sensible and realistic city layouts. Furthermore, for the tool to be flexible and useful for many projects is must be able to create settlements from different design styles. The different eras I am thinking of exploring, is a Native American Village, Medieval Town around, and Modern City.

#### Learn and Understand C#’s and 3D Unity Engine’s tools.

It is vital I understand how to use the Unity Engine, and its preferred language C#. This will include how it read in textures, create a flexible key system, and how to load and unload models.

#### Research 4 Flood Fill algorithms.

Develop knowledge and understanding of 4W, WBF, GTF and SF. This will be done by reading pseudocode of each algorithm, and then finding an example of these algorithms implemented.

#### Implement Flood Fill algorithms.

After understanding how a variety of different settlement types are set out, I find the common themes among different the examples of a specific type and have buildings spawned according to those commonalities. This would include from not having buildings spawn over each other, rotating building around curved areas building structures, to moving building around geographical structures (e.g. Rivers).

#### Evaluate results and Test Prototype.

Develop a prototype that allows you to run all 4 algorithms and then test each one of them in the areas listed above. Compare these results and evaluate which circumstances each would be preferable (this could depend on the machine, settlement complexity, etc.).

### How the objective and Aims have Changed

The objective and changed mostly away from the research and understanding of various settlement types (e.g. Modern city, Medieval Town, Native American Camp, etc.) to focusing on FFAs. This was because there was a lack of focus on the technical side of the project and it was veering off to a more into anthropology, geography, and sociology.

## Project Outline

Introduction

*An introduction to the dissertation detailing the motivation, aims, objectives as well as the initial problem.*

Background and Research

*Presents the context and background research done for the project.*

Implementation

*A discussion on what the prototype can do and how I went about implementing the background research into it.*

Results and Evaluation

A section on displaying and analysing the results of each FFA procured from the prototype.

Conclusion

A summary on the fulfilment of objectives, summary of the achievements of the project, development of personal skills, and possible future work.

# Background Technical Material (10% ~800-1200 Words)

This chapter should cover two things in detail:

* A review of existing academic works. Use Google Scholar, identify papers related to the field. Start broad (e.g. procedural generation in general) and then narrow down to your particular topic (e.g. city generation). In particular, state why each paper mentioned is relevant to your project and how it has in influenced your decisions.
* The algorithms and technology used, in detail. If you are implementing an existing algorithm, this is where you describe the algorithm in detail in such a way that a competent programmer could also implement.
* It’s okay to have a few references to games articles, but the bulk should be peer reviewed academic papers (journals and conferences).
* Then summarise the findings of the background chapter, relating to decisions made for your project. You should then refer back to these findings in the Implementation and Evaluation chapters (i.e. when describing implementation decisions, support them by referring back to find from the research, similarly when identifying an evaluation method, or describing specific results, support them with evidence from other people’s research, state that they are different and why that might be).

Within this chapter, I will be discussing the background research that went into my project. Starting with a broad overview of the literature on Procedural Generation, narrowing down to the specifics of settlement generation. Pseudocode and high-level explanations of the Flood Fill algorithms examined will be detailed. As well as the reasoning behind to style of diagram used for the settlement to be generated from.

## Procedural Generation

Automated world creation is often done through procedural generation which is a method of creating data algorithmically as opposed to manually. This is commonly done through some pseudorandom method, such as Perlin Noise [5], Wave Function Collapse [8] [12], etc. Unsurprisingly, this is a very powerful tool within the realm of video games, as it widens the horizon of possibilities for games developers to exercise their creative authority. Whether it be, reactive music [13], unique worlds for each playthrough [14] or inimitable storylines; all these things [15] - as Tanya X. Short and Tarn Adams put it:

[Allows] Players [to] have their own personal journey but still have enough common experience to share their tales with others… [16]

When starting out research, I found the paper *Procedural Content Generation for Games: A Survey* [17] very useful as it gave a very in-depth look into a great swath of procedural generation techniques; from particle effects to buildings. One of the things it touched upon was whole city generation which then sparked my curiosity of looking how different algorithms could create different results and patterns in city layouts. One example was this talk by *Oskar Stålberg* at *IndieCade Europe 2019* where he discussed how he created organic towns from square tiles using a Wave Function Collapse. [18]

This showed me that there were ways to create settlements that didn’t seem necessarily rigid and soulless. After all, settlements in real life are created organically and evolve over decades if not centuries.



Figure . Evolution of NYC [46]

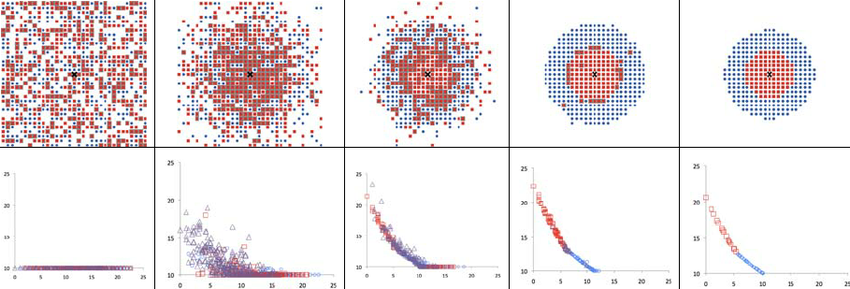


Figure . Evolution of the shape of the city (first line) and of the price of land as a function of the distance to the centre (second line) during a simulation. [19]

Looking more closely into *Procedural Content Generations for Games: A Survey*, they cited a paper by *George Kelly* and *Hugh McCabe* where they published a survey about multiple different techniques around City Generation. [6] They discussed many techniques such as Fractals, Tilling and the previously mentioned Perlin Noise. However, none of these appealed to me. This was because many of these seemed to be focus purely on modern day cites and weren’t flexible enough to accommodate different city structure types. Such as medieval European settlements, which are very popular in the RPG genre, or maybe something even more niche settlement style, such as native American settlements.

This led my down the path of analysing different city types in general [20] which, in turn, made me rediscover Land Use Models (LUM) which I had learnt about in middle school.

## Land Use Models (LUMs)

Regarding the design of the diagram, it was based of the design of LUM which are abstractions of complex city layouts. However, the simplification of each settlement’s layout varies on the type of LUM used:

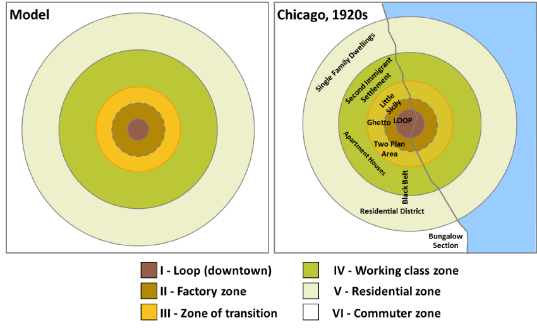
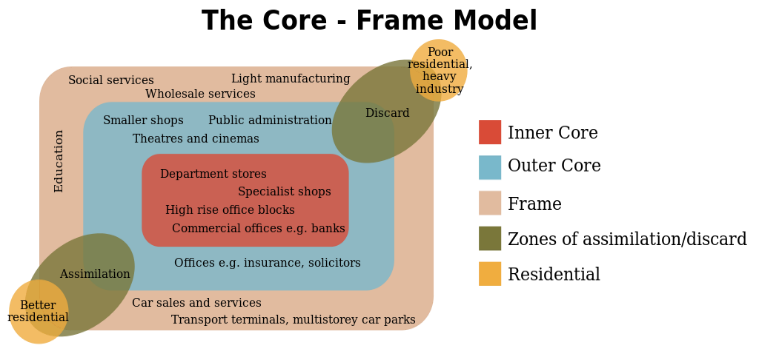
* Concentric Zone / Burgess Model [21] (Figure 2.3) [22]
* Core Frame Model [23] (Figure 2.4) [24]
* Sector/Hoyt Model [25] (Figure 2.5) [26]
* Multiple Nuclei Model [27] (Figure 2.6) [28]

Figure 2.4 Core Frame Model

Figure 2.3 Burgess Model

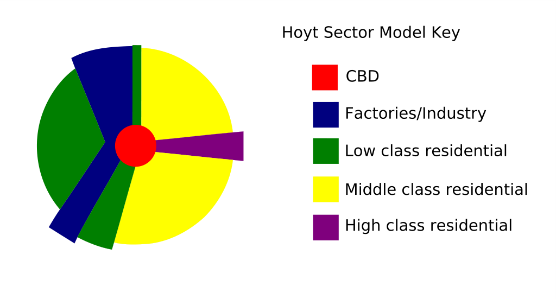
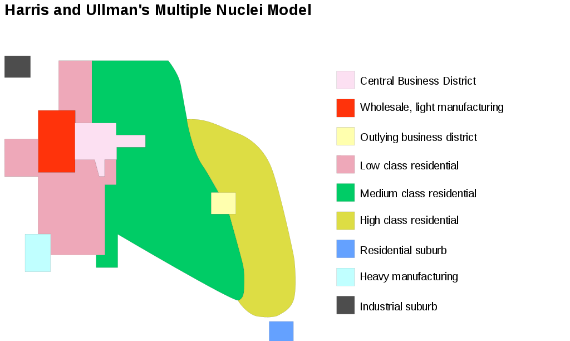
These diagrams take a very abstract view of city by mapping out their main socio-economic areas.

Figure .4 Sector Model

Figure .5 Multiple Nuclei Model

These models were inspirational in design only and in the end a less abstract approach was taken. (see Figure 1.1) This led to common structures such as roads and walls to be manually imputed. Furthermore, the labelling of the sectors and what buildings spawn withing them is up to the digression of the user. If the user wanted, they could use a procedurally generated urban LUM for a more realistic modern day city layout. [29] [30]

While there hasn’t been much research done into using LUMs for generating cities there has been some research in procedural modelling of urban layout - using certain proportions of land allotted to each sector [31] (page 3339 specifically) – and Procedurally generating city layouts based on LUMS. [29]

In the end, while not technically relevant, this tangent is what inspired me to base my settlement generation of a premade texture, which a level designer could create themselves and then input into the system. They could have different buildings spawn in different areas - effectively creating the different sectors at seen in the LUMs above.

However, as mentioned before, this project now lacked a technical side. Instead of trying to investigate a technical problem and publish results on my findings, I instead was trying to create a product that would be usable by developers in the future. Not exactly what a dissertation in designed for.

## Flood Fill Algorithms

Eventually, I landed on examining the used of Flood Fill algorithms. This was because during my time trying to implement my prototype, I needed to find a way to map out each sector (a 2d polygon) into a data structure as a set of coordinates for each pixel. Flood Fill algorithms – also known as seed fill algorithms- are designed to find adjacent pixels of a common characteristic (in this case colour) and only stop until all connected pixel, with said common characteristic, are found (such as the earlier mentioned bucket tool) [11]. There are many different types of flood fill algorithms and different methods of implementing them.

Some other examples where flood fill algorithms have been applied for maze-solving [32] and passive audio detection and tracking – through either four-dimenational source tracking (x, y, z, time) and signal detection when applied to spectrograms [33].

For this project I took a wide variety of flood fill algorithm for colour filling and adapted them to my software.

## Four Way

One of the earliest flood fill algorithms was detailed in *Principles of Interactive Computer Graphics* by *William Newton* and *Robert F. Sproull* [34] where:

[The] filling operation starts by replacing the value of a single pixel, and then spreads throughout the raster, replacing the value of any pixel that contains the old colo[u]r. The spreading operation stops whenever it encounters a pixel that doesn’t not contain the “old” colo[u]r.

The algorithm spreads out by examining the adjacent pixels *above, below, left* and *right* of the initial pixel. This can then be either repeated recursively or linearly until you have found all adjacent pixels.

This algorithm was first suggested using a recursive method (see Pseudo code in chapter 3.3.1). [34] [35]

## Span Filling / Scanline

Span Filling, also referred to as Scanline Fill, is an optimisation of the previous Four Way algorithm. Instead of filling pixel by pixel, it fills a whole scanline.

Initially, it was described by Alvy Ray Smith, who referred to it as Tint Fill, as such:

Tint fill fills along a scanline under the rule that it can never go uphill. It can fill along level ground or downhill only. A scanline segment for tint fill consists of all the pixels proceeding from the seedpoint right (and left), which have the same tint as the seedpoint and a value which is either the same or less than the pixel just left (right). Thus, a scanline segment is a section of a hill or mesa. [36]

A more modern explanation of the algorithm would be such:

Starting with a seed point, the algorithm fills left and right of it until it hits and edge. From this, it will scan the same horizontal areas of the lines above and below, searching for new seed-points to continue the algorithm.

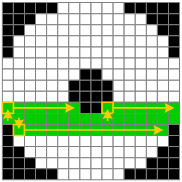
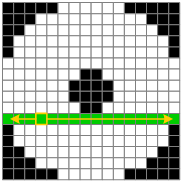
 [37]

Figure 2.1 First, filling in horizontally from seed (left picture), then find a new seed in the lines above and below (right picture).

This is further detailed within *Computer Graphics: C Version* when discussing the using spans within the context of Boundary Fill - essentially Flood Fill when you are only trying to find the boarders. [38]

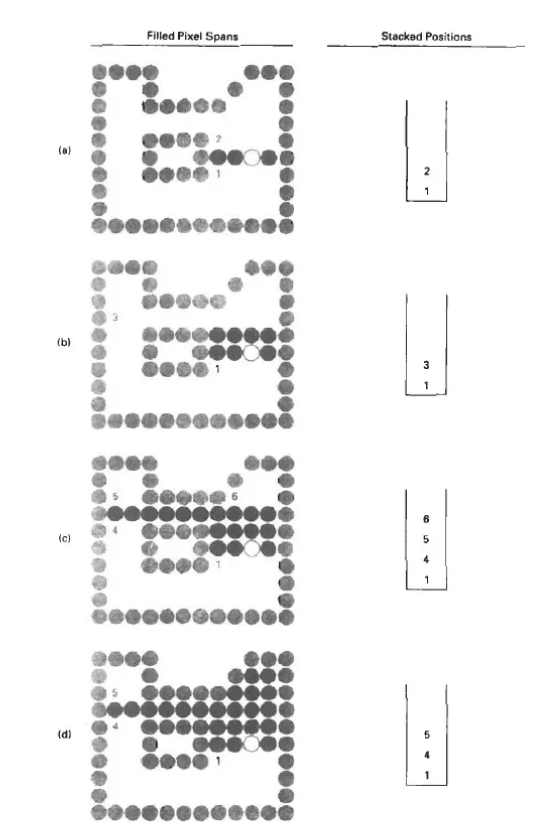
 [35]

Figure . Flood Fill across pixel spans for a 4-connected area.

1. The filled initial pixel span, showing the position of the initial point (open circle) and the stacked positions for pixel spans on adjacent scan lines.
2. Filled pixel span on the first scan line above the initial scan line and the current contents of the stack.
3. Filled pixel spans on the first two scan line and the current contents of the stack.
4. Completed pixel spans for the upper-right portion of the defined region and the remaining stacked positions to be processed.

## Painter / Walk Based Filling (Fixed-memory method)

Walk Based Filling, or what I like to refer to as the Painter algorithm, is the most complex algorithm examined in this dissertation. The aim of this method is to uses minimal memory for four-connected regions as described by Dominik Henrich:

[W]e describe algorithms that need little additional memory that is of constant size so that it can be reserved in advance. … Roughly speaking, the global filling strategy is: move around in the region and fill it in such a manner that the region remains connected. [39]

Essentially, from the initial seed point, the cursor moves around the region painting each pixel without painting themselves into a corner, hence, why I refer to it as the Painter algorithm. The cursor only can see the 8 pixels around them and can only move in to the four-connected regions. The cursor follows the “right-hand rule” (RHR), this mean, figuratively, the cursor is always moving in the direction so that there will be an adjacent wall directly on its right. This leads to the cursor finding themselves in one of these conditions:

1. One boundary pixel is filled.
2. Two of the boundary pixels are filled.
3. Three of the boundary pixels are filled/
4. All four boundary pixels are filled.
5. Zero boundary pixel are filled.

Add Images:

All directions are relative to whichever way the cursor is facing.

**Case #0:** Move forward until you find a boarder. Don’t paint until boarder is found.

**Case #1:** Check the front 8-corners (front left and front left) to see whether they are filled or not. If both are empty, then the cursor can continue using RHR. If either or both are filled, then this creates an intersection of multiple paths that cannot be filled, which means if the current pixel was painted that may prohibited the painter from returning and filling the other side of the paths.

To solve this issue, we introduce a “mark” to define where the junction is and which direction the cursor was facing when the mark was placed. After this mark is created to paint move forward according to the RHR while no painting. Once, it reaches either a dead end (all pixels filled apart from the one behind) then the cursor turns around and continues to paint following RHR, if it there is an opening, more than 2 filled pixels, it places another mark and continues following the RHR.

**Case #2:** When encountering a path with only 2 free pixels this could lead to another intersection of multiple paths. Likewise, with #1 if the front 8-corners are filled then it follows the logic set out int Case #1.

Otherwise, a mark is placed for the first 2-pixel boundary to remember where the opening of the passage is and in what direction the painter was moving. If the cursor is encountered again and that cursor is going in the same direction as the mark, then it is safe to paint the square with the mark and to continue in the same direction. Through some unknown path, the pixels on the other side of the mask can be reached, hence it is able to be painted in the future. The mark is then reset.

If the cursor finds itself connecting with a mark different to its current orientation, then a loop of some sort as happened, causing the cursor to return to the mark. This found loop must be removed. The park is reset, and the cursor paints in the direction indicated previously by the mark, now following the “Left-hand Rule” (LHR). The cursor keeps moving forward until an intersection is encountered, with 1 or less filled bounding pixels. Continuing using the LHR, the courser now searches for a pixel with a boundary of two pixels. This then allows for the loop to be broken and the algorithm continues.

**Case #3:** There is only one open path, thus continue painting allowing this paths.

**Case #4:** There are no more pixels to be painted. The cursor paints the pixel it is on and stops the algorithm.

This was algorithm was first published by Dominik Henrich in 1994. [39]

# Implementation (What was done, and how) (40% ~3200 -4800 words)

- In detail, what you did and why you did it. This is your work. If you designed or added to an algorithm talk about that here. If you created a program to implement an existing algorithm, talk about that. Probably you combined various techniques, algorithms and technologies, so talk about how and why you did that.

- It is okay to discuss any blind alleys you spent time on – describe that the idea was and why it was abandoned or changed, in the broader context. - As mentioned above refer decisions back to findings from the research chapter that support them.

- Code snippets can be useful if they support the text but not a requirement

- Screenshots are definitely useful if relevant. Show the progress of the project from early protype to final build. - End with a summary of the implementation and mention how it will be evaluated

## Overview

To start off my implementation, I had to select how I wanted to approach the project: to either attempt to implement it at a low-level, with technologies such as OpenGL, or at a more high-level with technologies such as the Unity Engine. Initially, implementation was attempted in OpenGL with C++ but due my lack of experience with such tools and overall aim of the project, I ended up deciding against it. The project eventually moved towards the idea of using pre-existing meshes to create a settlement, which were tools that Unity was already able to use. Furthermore, I had much more experience with Unity as a technology then with OpenGL and Unity is currently a much more favoured unity engine for indie programmers.

When starting my project, I came across the Color class and it’s GetPixel() function within the UnityEngine. I initially used this when I was trying to use monochrome colours to create a height map of somesort. This was before I had landed on the LUM style of diagram (Figure 3.1). This was then represented by cubes floating in the air and ended up looking like some sort of voxel map.

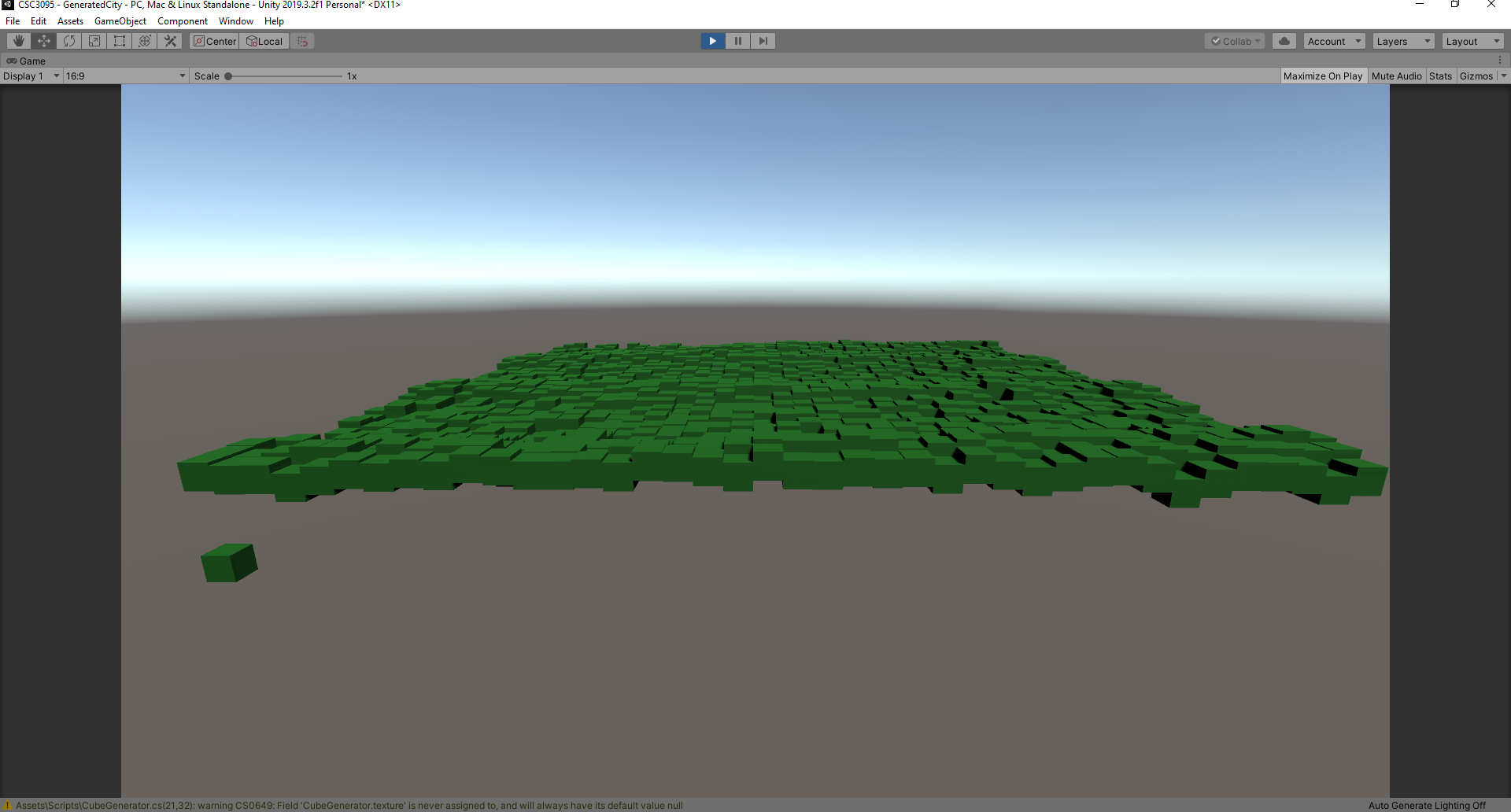


Figure . Voxel Style Map which based Cube's Y value according to the luminoncity of the pixel (white would like to a higher Y value).

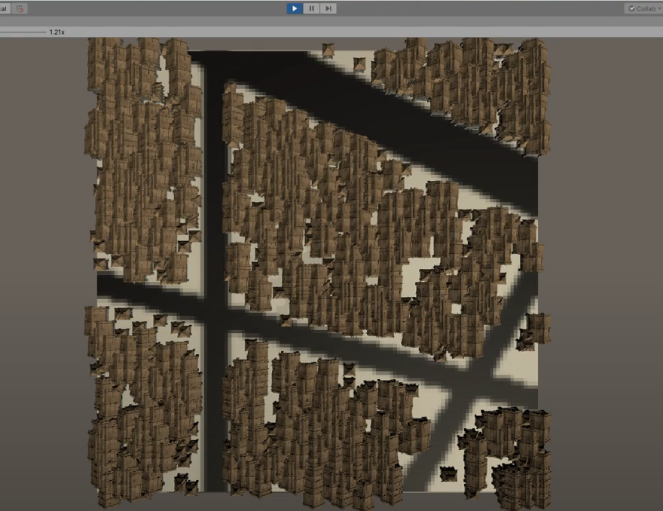
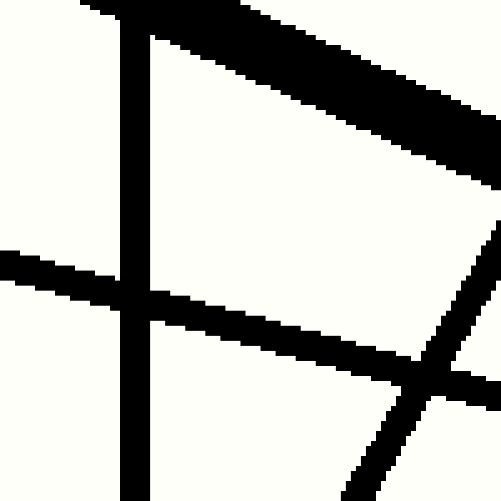
Next, I went experimenting with just trying to place houses on a singular colour. This was before I had landed on the idea of investigating flood fill algorithms. I tested many times using 100px\*100px size texture with black lines across it. The aim was to get it so the houses to spawn not on top of these black lines. (Figure 3.2)

Figure . On the left is the texture that was inputted. On the right is the output in Unity. The brown items are the houses.

However, this did lead to a few issues mainly that irregularity and non-sensical nature of how the houses were positioned. They weren’t in straight lines or in rows and I came to realisation that trying to make a settle generator with totally sensible housing placements was not really possible within the scope of this dissertation. Thus, the research into different settlement types was ultimately dropped.

This then led me to finally finding flood fill algorithms. From here I implemented the simple Four Way algorithm and found it to be a little slow and coming into conflict with the dreaded “Stack Overflow”. Therefore, I pivoted to implementing the Linear version of the Four Way method for the time being. I then adapted it so that it would only spawn Meshes on the perimeter of the shape (Figure 3.3). This functionally was not really need and unnecessary since I later realised that allowing the meshes to be placed anywhere within the shape, made more sense. Instead of implementing a proper flood fill algorithm, I was implementing a Boundary Fill algorithm. [38]

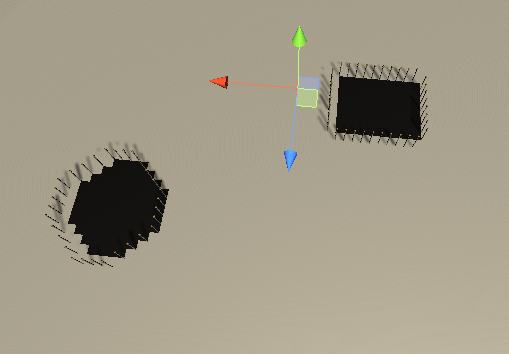


Figure . Meshes being placed on the perimeter of the shape.

However, between implementing the Linear Four Way method and coming to conclusion to not use just the perimeter, I had developed a method of perimeter checking on my own. In concept it is very simple and inefficient but because of how *FloodFill.cs* was set up and you don’t automatically pass in a seed value to set off with (instead having to find once for yourself) I decided that this algorithm could theoretically be more efficient. This was what is now referred to as Perimeter Fill.

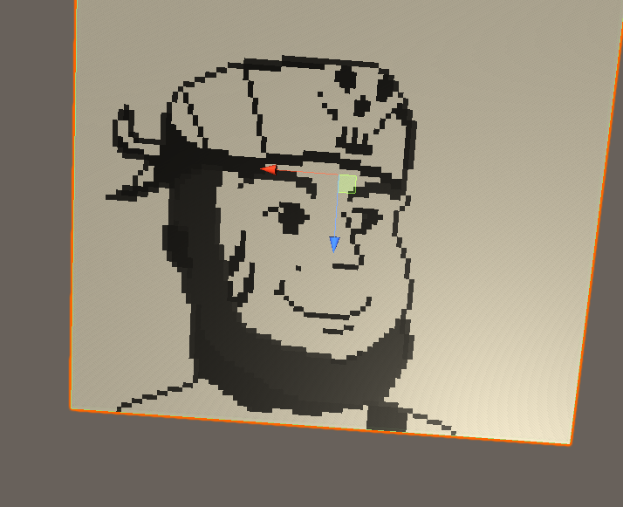
Perimeter fill is very simple to understand. You check every pixel in the texture (one by one) and when you come into contact with a pixel that has an equal colour value to the colour you are looking for, it then checks the four pixels around it (like 4 way) to see if any of the pixel are not equal in colour value to the colour parameter passed in. If at least one of the 4 adjacent pixels is not of that colour then the original pixel is added to the data structure.

Figure . An example of Perimeter fill working on a pixel art character I drew. This was a joke I sent to a few friends at the time, but in hindsight it is a very good example of showing the perimeter fill working.

This is when I started implementing the rest of the flood fill algorithm, first starting with Scan Line as it seemed like a simple and naturel progression from Four Way. While being a little more complex, it ultimately was building upon the foundations of its predecessor. Instead of adding pixel by pixel, it added row by row. Furthermore, theoretically it should be much faster because it didn’t have to check each pixel up to 4 times and instead maybe read a pixel a maximum of 3 times, still inefficient but an improvement.

With these two algorithms under my belt, I was able to start looking into more complicated shapes then a simple ‘+’ sign. This is when, as a joke, I took a picture of the pixel art character I had drawn and then got it to place meshes around it’s boarders, effectively making a picture out of building parts.

Next was to experiment with different colours. This meant that I had to create another for loop within *Main();* in *BuildingGenerator.cs*. Up top of this, I had to create a *SerializedField* of a *Color[]* so that the user could input there colours of choice. This is where Unity’s Engine came in super useful, not only could you input you precise colours into the array using either RGB or HSV, but you could also use the colour tools. (Figure 3.5).

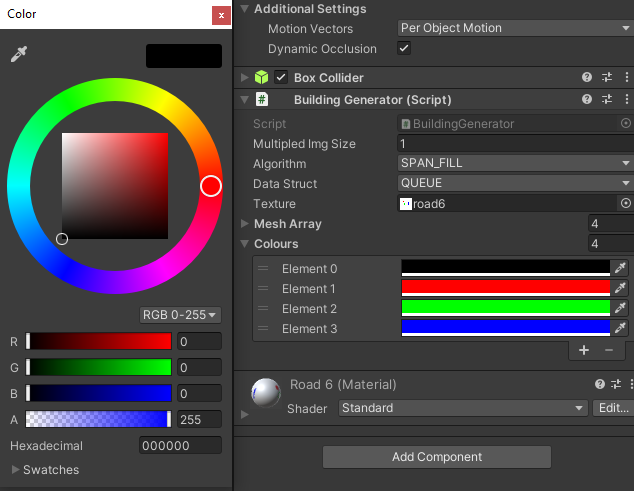


Figure . Colours Array in Unity Inspector

The first test of multiple colours had 4 colours (RGBA values):

- Black [0,0,0,255]

- Red [255,0,0,255]

- Green [0,255,0,255]

- Blue [0,0,255,255]

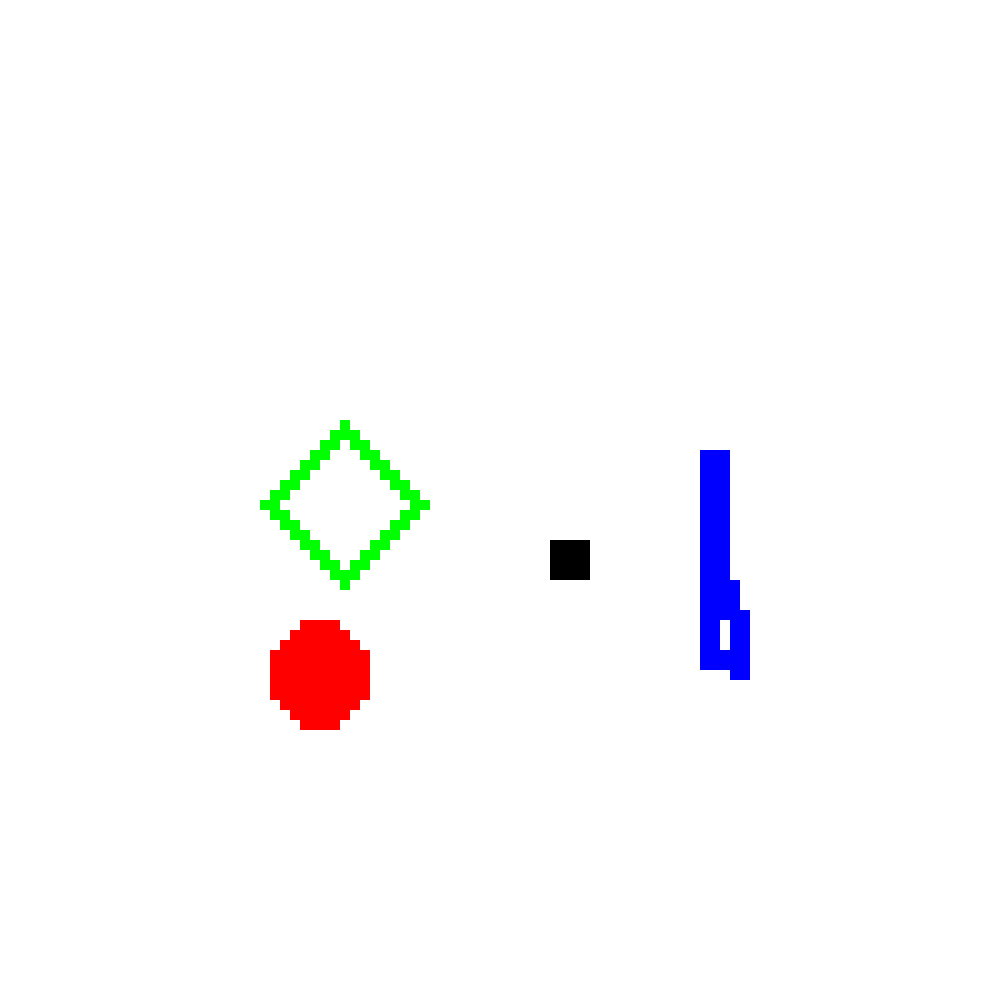


Figure . 100x100px test of different shapes and colours

With each colour was a different shape coordinated with each colour, square, circle, hollowed diamond, and a tall-skinny blob respectively.

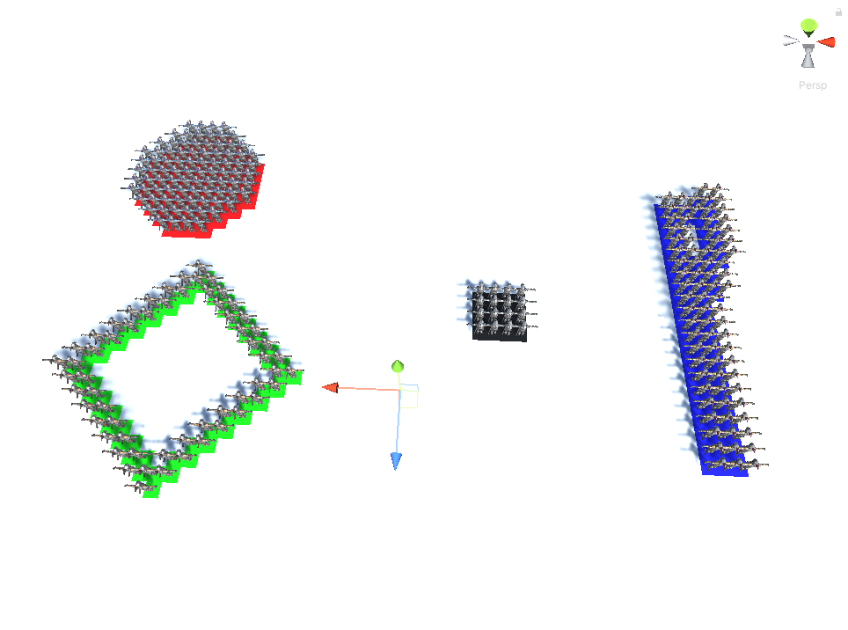
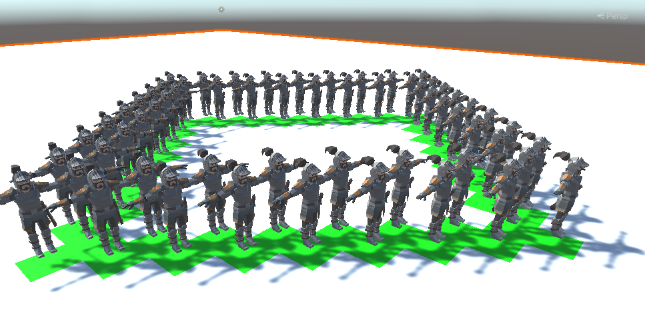
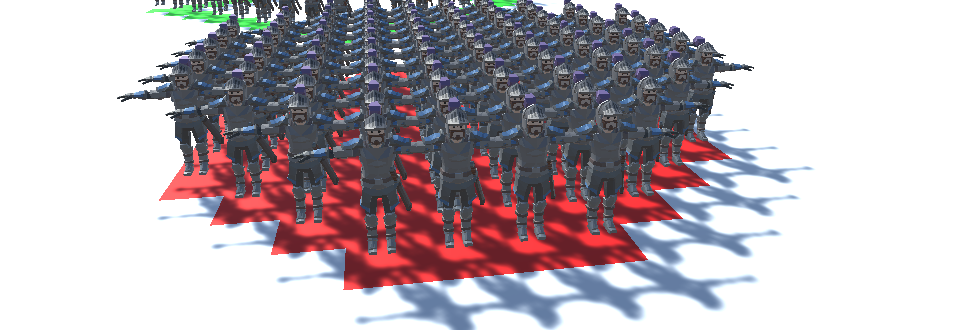
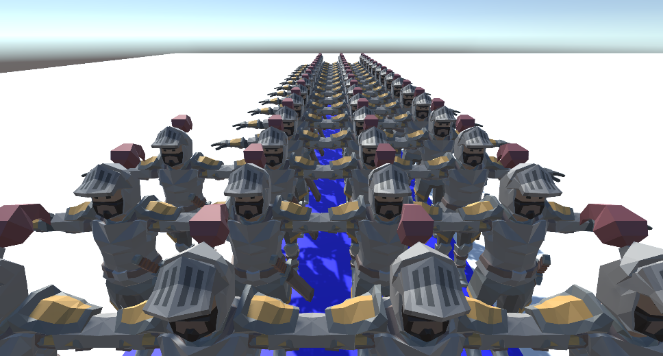


Figure . Showing Flood Fill algorithms working on multiple shape types and colours. Notice how there is a different Knight mesh on each colour. There is a different Mesh aligned for each colour.

However, both of these methods were quite memory intensive, and this is when I started to implement the Walk-based-filling method. This would be by far that hardest to implement since it’s logic was not simple - having multiple cases to take into consideration, more complex pseudo-code in which I had to change what whould have been *goto* statements into *while*, and *if* statements with multiple methods. Furthermore, due to its complexity and many while loops, it was the hardest to debug. With Unity freezing on start-up and having to have me force quite it countless times as I crawled through the debugger. Furthermore, there were a few logical cases not seemingly covered by the pseudo code I was using, particularly in the case where you only have 1 adjacent pixel block off to you, but you can still go down an ambiguous path. Thus, I added a check for the front 2 pixels (both Front Left and Front Right) to see if this gap was only 1 pixel big.

After this I then created a proper test town with 6 different colours all represent different areas of town, whether it be roads, poor housing, entertainment district, wall, etc. [See more in chapter 4] and tested it casually to see if all the algorithms worked.

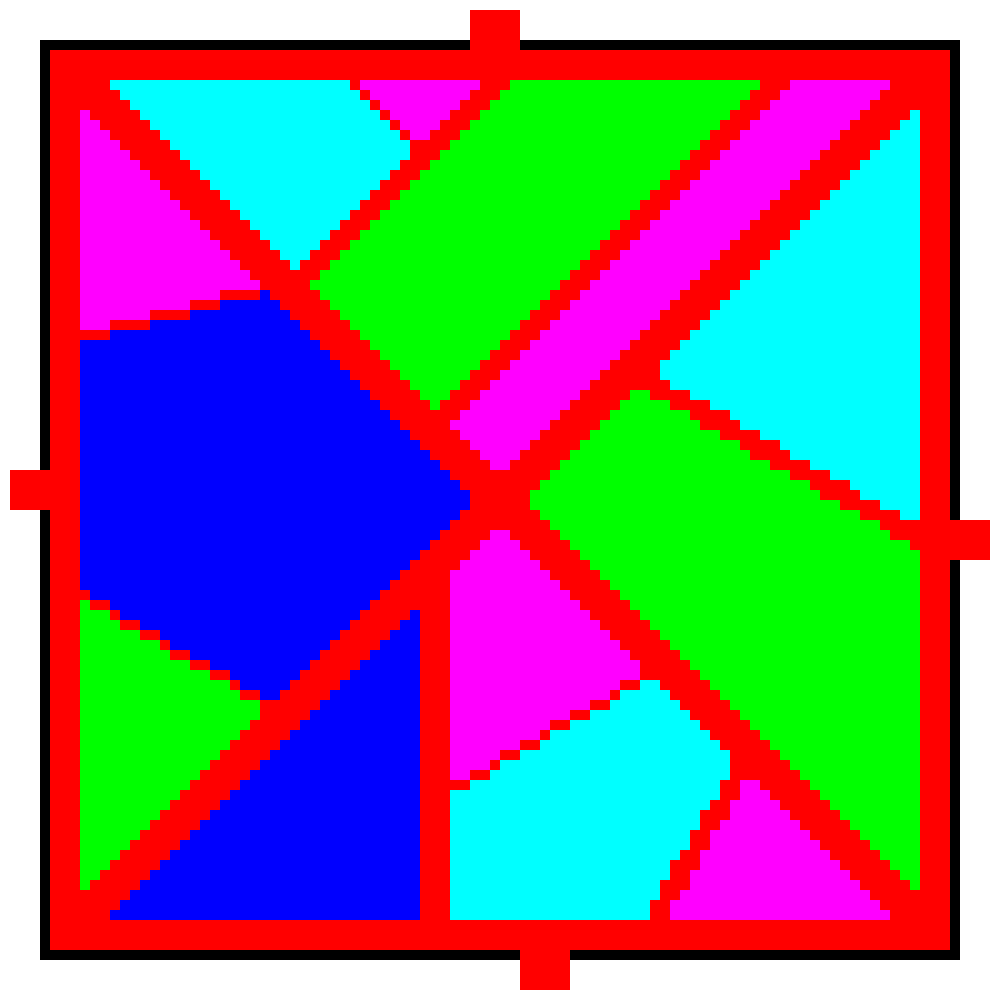


Figure . Full Town Plan

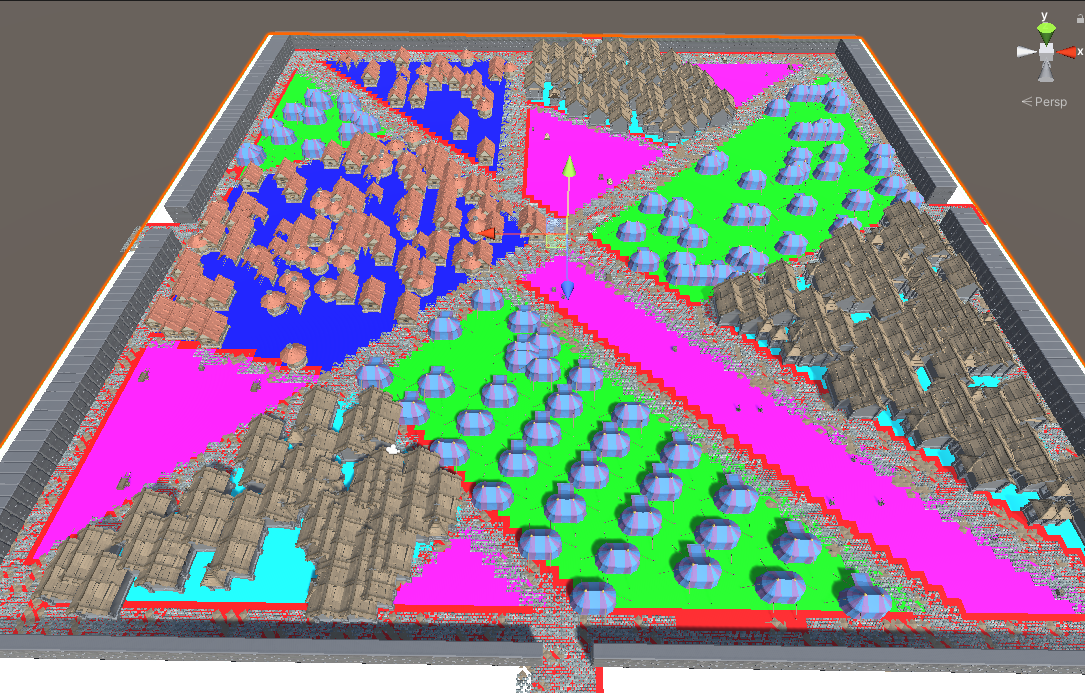


Figure . Populated town in Unity. Notice the different buildings within different sectors.

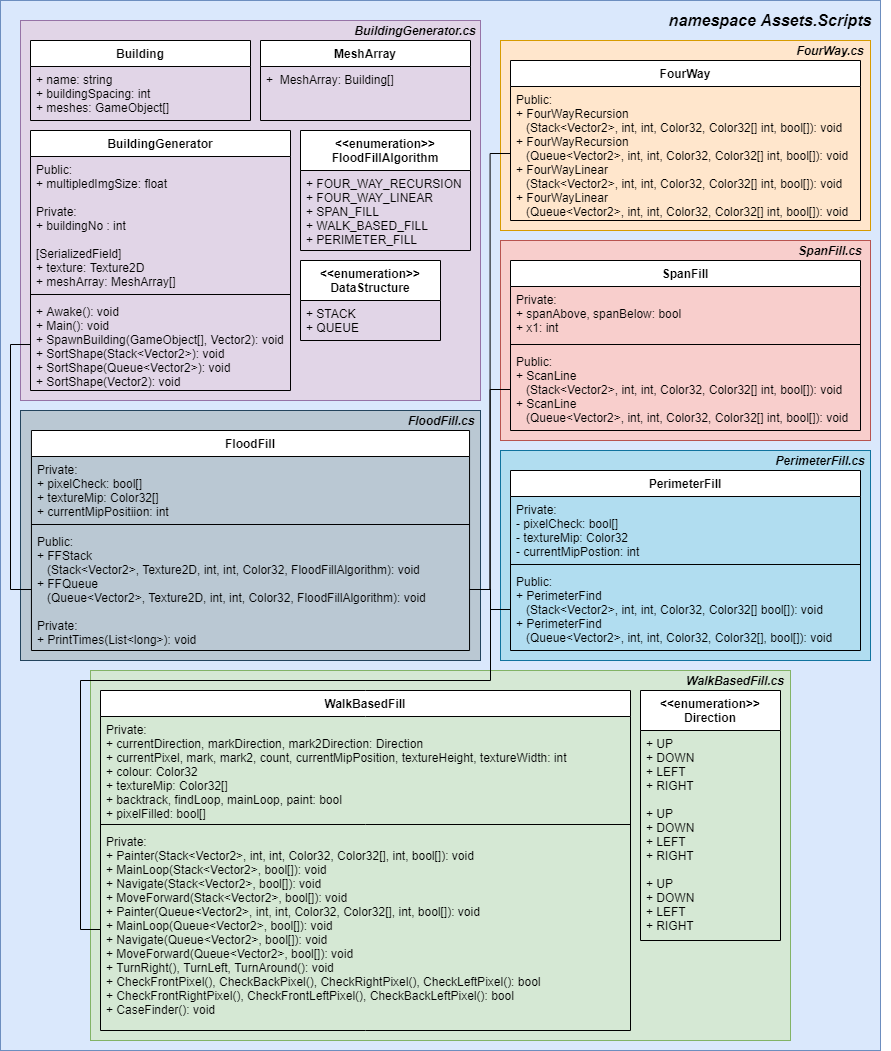


Figure . UML diagram of the code base

## BuildingGenerator.cs

As seen in the UML diagram above (Figure 3.1) the main script is *BuildingGenerator.cs*. This script is attached to the scene object that is representing the ground. It is from this script that you can adjust the variables for the whole program:

* The settlement size Magnifier.
* Select which algorithm you would like the settlement drawn by (Four Way, Span Fill, Walk Based Fill, Perimeter Fill). This is done via an enum.
* Select which data structure you would like (Stack or Queue). This is also done via an enum.
* Array of Buildings/Meshes you wish to spawn in certain sectors. Within this you can Name and input the amount of space you would like each building to be in.
* Array of Colours that correspond to each sector.

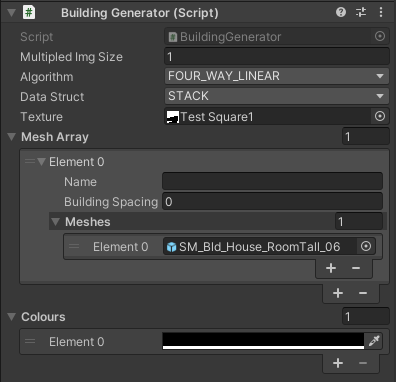


Figure . BuildingGenerator.cs in the Unity Inspector tab

Within the script are 3 important functions:

SpawnBuildingFloodFill();

This method is called on *Awake()* and creates a *FloodFill* object. From this it iterates through every colour in the *Color32[]* finding the pixels of the selected colour and added it to either a *Stack<Vector2>* or *Queue<Vector2>*. The shapes then can be sorted by *SortShape()* and then *SpawnBuilding()* is called. *SpawnBuilding()* spawns a single mesh and within each sort there is a variable that enables the user to control how much space is in between these building.

SortShape(Vector2[]);

Sort shape is most useful for when you are just wanting to spawn buildings on the perimeter of the shape. This was method was created when the implementation of the flood fill algorithms was focused more on the perimeter of the shapes then the whole area. Thus, this algorithm is not optimised and instead it is an attempt to get the job done when using *PerimiterCheck()*.

**Pseudo-Code:**

****

This algorithm was needed for when each building wasn’t spawn right next to eachother (i.e. when *meshArray[i].buildingSpacing* != 1). This was because the pixels are added to the array in the same order that *Texture2D.GetPixels32()*lays it out:

The returned array is a flattened 2D array, where pixels are laid out left to right, bottom to top (i.e. row after row). Array size is width by height of the mip level used. [40]

This means that the space of buildings would be uneven around the perimeter. This method aims to sort the *Vector2D[]* into an anti-clockwise direction.

SpawnBuilding(GameObject[], Vector2);

The position of the pixel is passed in as a *Vector2* and then a random building/mesh from the *GameObject[]* is chosen to be spawned.

## FloodFill.cs

From *BuildingGenerator.cs* (within *SpawnBuildingFloodFill()*), *FloodFill.cs* (either *FFStack()* or *FFQueue()*) is a called. From these methods, the selected flood fill algorithm will be called – the one selected in the Unity Inspector.

The main thing to note about this class is how it then deviates from how most FloodFill algorithms are implemented. Most FloodFill algorithms have the seed assigned at the pixel where the mouse / paint bucket is clicked upon. However, the goal of this project is to map out every pixel of the same colour. This means that we use *FloodFill.cs* to iterate through the Mip Map produced via *Texture2D.GetPixels32()* until we encounter a pixel of the same colour as the one passed through as a parameter. This is the reason why *Texture2D.GetPixels32()* was used, as it is faster *Texture2D.GetPixels32()* once then *Texture2D.GetPixel()* repeatedly. [40]

A further adaptation for this project, was the need for an addition *bool[].* Every element in this boolean array would correspond to a pixel within the Mip Map. For a traditional bucket tool, the colour of the pixels would be changed as the algorithm is running. However, since we are mere just trying to collect these pixels into a data structure, and not change the initial texture, we need a way to mark a pixel as one that has already been check. Otherwise, we could find ourselves unnecessarily checking the same pixel multiple times. Thus, every iteration we also mark *pixelCheck[currentMipPos]* as true. This *bool[]* is then updated after each running of a flood fill algorithm since all of the pixels of that shape have now been checked.

## Brute Force

As is somewhat standard when comparing algorithms, having some sort of brute force algorithm to set a base line for the other algorithms to compare against very useful. However, as we will see later (in chapter 4) Brute Force actually seemed to compete quite well with the rest and outperform in some of the tests. The reasoning why was because the initial seed (pixel) had to be found automatically, and was not inputted manually, if all of the shapes had seeds that were entered in manually then Brute Force unlikely be able to compare. However, we shall save the rest of the analysis for later.

### Pseudo-Code: Brute Force

The pseudo-code was devised by myself for this project and was very simple to implement:

**Brute Force**(Shape) :

for all pixels in the texture

if (Pixel is selected colour)

add Pixel to Shape

**endFor**

## FourWay.cs

When one of the FourWay algorithms are selected (either FOUR\_WAY\_RECURSION or FOUR\_WAY\_LINEAR). As shown in Chapter 2.3 you can have either a recursive implementation or a linear implementation. The Linear is almost always preferable because on even a moderately sized image you could blow you stack, leading to a *StackOverflow* error. I implemented both in order to test this out, mainly because in my research the pseudo code given was of the recursive implementation [35], but many more modern resources suggest if you are doing the standard Four Way algorithm it would be best to implement it not recursively. [41] [42]

When implementing linearly, it was often suggested to either use a stack or a queue as a data structure, and while there is not theoretical difference between them, I was inclined to test them both against each other in the context of this project (see chapter 4).

### Pseudo-Code: Recursive Implementation

This pseudo-code was adapted from both *Principles of Interactive Computer Graphics* [34]and *Computer Graphics: C Version* [35].

**FourPointRecursion**(Shape) :

**if** (Pixel is within Texture)

**if** (Pixel is selected colour **&&** Pixel has **not** been checked yet)

add Pixel coordinates to Shape;

set Pixel as Checked;

**endIf**

**if** (Pixel has been checked)

**return;**

**endIf**

set Pixel as Checked;

call **FourPointRecurions** for pixel directly North;

call **FourPointRecurions** for pixel directly South;

call **FourPointRecurions** for pixel directly East;

call **FourPointRecurions** for pixel directly West;

**return;**

**endIf**

The key difference here is that we are having to check if the Pixel is within the *textureMip* and then keep track of whether the pixel has been checked or not (via the *bool[]*, *pixelCheck*).

One other thing I had to be weary of is the fact *textureMip* was a flatten array thus couldn’t distinguish between *x* and *y* values. This meant this this would have account a lot. Because *textureMip* ordered it’s elements from *left* to *right*, *bottom* to *up*, *texture.width* had to be used a lot to find the coordinate values:

Shape.Push(new Vector2(currentMipPos % textureWidth, Mathf.Floor(currentMipPos / textureHeight)))

The variable *currentMipPos* was used to keep try of which pixel we were on in the array.

### Pseudo-Code / Code: Linear Implementation

This Pseudo code was based of the code from Karim Oumghar. [43]

**FourWayLinear**(Shape) :

Instantiate new temp data structure dS;

Add Pixel to the end of dS;

**while** (dS is **not** empty)

Set p equal to the first element of dS;

Remove first element from dS;

**if** (p is within Texture)

**if** (p is selected colour)

**if** (p has **not** been checked yet)

Add p coordinates to Shape;

**endIf**

Add pixel directly North of p;

Add pixel directly South of p;

Add pixel directly East of p;

Add pixel directly West of p;

Set p to Checked;

**endIf**

**endIf**

**endWhile**

**return** Shape;

## SpanFill.cs

Span Fill has the same alterations from the algorithms discussed above.

### Pseudo-Code: Scan Line

This Pseudo code was based of the code from Lode Vandevenne [41] and Karim Oumghar [43].

**ScanLine**(Shape) :

Set bools spanAbove and spanBelow to false;

Instantiate new temp data structure dS;

Add Pixel to the end of dS;

**while** (dS is **not** empty)

Set p equal to the first element of dS;

Remove first element from dS;

Set int x1 to p.x;

**while** (x1 >= 0 **&&** x1 position on span line is **not** Checked **&&** is the same colour)

x1--;

**endWhile**

x1++;

while (x1 < pictureWidth **&&** x1 position on span line is **not** Checked **&&** is the same colour)

Add Pixel(x1, p.y) to Shape and set Pixel(x1, p.y) to Checked;

**if** (!spanAbove && p.y > 0 = 0 **&&** x1 position on span line is **not** Checked **&&** is the same colour)

Add Pixel(x1, p.y - 1) to dS;

Set spanAbove to true;

**endIf**

**elseIf** (spanAbove **&&** p.y > 0 **&&** x1 position on span line is Checked **&&** is the same colour)

Set spanAbove to false;

**endIf**

**if** (!spanBelow && p.y < pictureHeight - 1 **&&** x1 position on span line is **not** Checked **&&** is the same colour)

Add Pixel(x1, p.y + 1) to dS;

Set **spanBelow** to true;

**endIf**

**elseIf** (!spanBelow **&&** p.y < pictureHeight - 1 **&&** x1 position on span line is Checked **&&** is the same colour)

Set spanBelow to false;

**endIf**

x1++;

**endWhile**

**endWhile**

**return** Shape;

## WalkBasedFilling.cs

The was not a whole lot of implementation of Walk Based Fill that I could find, with other flood fill algorithms in mazes being more popular. Thus, this code was harder to implement because the sources provided which in turn lead me to altering some parts of the pseudo-code.

### Pseudo-Code: Painter / Walk Based Filling

This Pseudo code was based off the pseudo-code discussed in *Space-efficient region filling in raster graphics* [39] and provided from the Wikipedia page on Flood Fill Algorithms [44] with some major alterations.

Firstly, is the addition of Case 0. Where the cursor will just continue forward until it hits a boundary (a case that is not Case 0) as specified in Chapter 2.6. Furthermore, the condition of the first *if* statement in *Navigate()* also includes the condition of *count* not equalling both 4 and 0.

Secondly, an *if* statement in both Case 1 and Case 2 were added. I check if the front corner pixels (front-left and front-right) are full, indicating a 1 pixel wide bottle neck, and that a mark has not been placed down (*mark* is -1 or null). Once in, the cursor moves forward without painting until it can’t move forward anymore. When it reaches this dead end, it turns around and continues to paint as before following the RHR.

Finally, I changed the code to reflect one of multiple methods and not one with multiple *goto* implicitly baked in.

**Painter**(Shape):

set cur to starting pixel

set cur-dir to default direction

clear mark and mark2 (set values to null)

set backtrack and findloop to false

set finished to false

set mainLoop to false

set paint to false

**While**(!finished)

**if**(mainLoop)

**MainLoop**()**;**

**endIf**

Set mainLoop to true

**if**(paint)

**MoveForward**();

**endIf**

**Navigate();**

**endWhile**

**return;**

**MainLoop**():

move forward

**if** right-pixel is inside then

**if** backtrack is true **&&** findloop is false **&&** either front-pixel or left-pixel is inside then

set findloop to true

**end if**

turn right

**MoveForward**(Shape):

**if**(paint)

add to Shape;

**endIf**

move forward

**Navigate**():

set count to number of non-diagonally adjacent pixels filled (front/back/left/right ONLY)

**if** count is **not** 4 **&& not** 0 then

**do**

turn right

**while** front-pixel is inside

**do**

turn left

**while** front-pixel is not inside

**end if**

**switch** count

case 0

**Paint();**

**MoveForward();**

case 1

**if** backtrack is true then

set findloop to true

**else if** findloop is true then

**if** mark is null then

restore mark

**end if**

**else if** front-left-pixel and back-left-pixel are both inside then

clear mark

set cur

**Paint**();

**else if** front-left-pixel and front-right-pixel are both **not**  inside **&&** mark equals -1

**do**

**MoveForward**();

**while** front pixel is inside **&&** count equals 2

set count to number of non-diagonally adjacent pixels filled (front/back/left/right ONLY)

**if** (count is not 2)

**TurnAround**();

break;

**endIf**

**endDoWhile**

**end if**

**end case**

**case 2**

**if** back-pixel is **not** inside then

**if** front-left-pixel is inside then

clear mark

set cur

**Paint**();

**else if** front-left-pixel and front-right-pixel are both not inside **&&** mark equals -1

**do**

**MoveForward**();

set count to number of non-diagonally adjacent pixels filled (front/back/left/right ONLY)

**while** front pixel is inside && count equals 2

**if** (count is **not** 2)

**TurnAround**();

break;

**endIf**

**endDoWhile**

**end if**

**end if**

**else if** mark is **not** set then

set mark to cur

set mark-dir to cur-dir

clear mark2

set findloop and backtrack to false

**else**

**if** mark2 is **not** set then

**if** cur is at mark then

if cur-dir is the same as mark-dir then

clear mark

turn around

set cur

Paint();

**else**

set backtrack to true

set findloop to false

set cur-dir to mark-dir

**end if**

**else if** findloop is true then

set mark2 to cur

set mark2-dir to cur-dir

**end if**

**else**

**if** cur is at mark then

set cur to mark2

set cur-dir to mark2-dir

clear mark and mark2

set backtrack to false

turn around

set cur

**Paint**();

**else if** cur at mark2 then

set mark to cur

set cur-dir and mark-dir to mark2-dir

clear mark2

**end if**

**end if**

**end if**

**end case**

**case 3**

clear mark

set cur

**Paint**();

**end case**

**case 4**

add to Shape

set cur

set Finished to true

done

**end case**

**end switch**

## PerimeterCheck.cs

# Results and Evaluation (30% ~2400 – 3600 Words)

With the wide number of algorithms and variations on said algorithms the amount of tests able to experimented was great. The two main categories of test were:

* Algorithm Speed: how fast the algorithm took to map out all the pixels of a given test.
* Memory Use: - The amount of memory used while the algorithm was running.

## Algorithm Speed

### Circle Test

### Square Test

### Test Town

### Edge Case Tests

## Memory Use

### Circle Test

### Square Test

### Test Town

### Edge Case Tests

# Conclusion (10% ~800 – 1200 Words)

* Eight Way

# Form/References (5%)

# References

|  |  |
| --- | --- |
| [1] | R. Eeps, “The Rise of Open World Games,” *TheGamer,* p. 1, 19 August 2019. |
| [2] | S. Biswas, “Cyberpunk 2077: Former CD Projekt Red Dev Opens Up About Crunch Culture Within the Studio,” *Essentially Sports,* 16 October 2020. |
| [3] | H. Cryer, “Cyberpunk 2077 bugs: all the weird and wonderful glitches we’ve seen so far,” *gamesrader+,* 14 December 2020. |
| [4] | C. Salge, M. Cerny Green, R. Canaan and J. Togelius, “Generative design in minecraft (GDMC): settlement generation competition,” *AMC Digital Library,* 2018. |
| [5] | N. Olsson and E. Frank, “Procedural city generation usign Perlin Noise,” Blekinge Insitute of Technology, Karlskrona, 2017. |
| [6] | G. Kelly and H. McCabe, “A Survey of Procedural Techniques for City Generation,” *ITB Journal,* p. 29, 2006. |
| [7] | G. Kelly and H. McCabe, “An Interactive System for procedural city generation,” Blanchardstown, 2008. |
| [8] | W. Gaisbauer, W. L. Raffe and H. Hlavacs, “Procedural Generation of Video Game Cities for Specific Video Game Genres Using WaveFunctionCollapse (WFC),” Association for Computing Machinery, Barcelona, 2019. |
| [9] | R. Olszewski, M. Cegiełka, U. Szczepankowska and J. Wesołowski, “Developing a Serious Game That Supports the Resolution of Social and Ecological Problems in the Toolset Environment of Cities: Skylines,” *International Journal of Geo-Information,* pp. 1-20, 2020. |
| [10] | Astral Dawn Studios, *Dodo Video Game! - Dodo Alone: DevLog #1,* Windsor: YouTube, 2020. |
| [11] | S. Hargreaves and D. Capello, *floodfill.cpp,* 2020. |
| [12] | K. Hwanhee, L. Seongtaek , L. Hyundong, H. Teasung and K. Shinjin, “Automatic Generation of Game Content using a Graph-based Wave Function Collapse Algorithm,” *IEEE Symposium on Computational Intelligence and Games, CIG,* 26 September 2019. |
| [13] | D. Plans and D. Morelli, “Experience-driven procedural music generation for games,” *IEEE Transactions on Computational Intelligence and AI in Games,* vol. 4, no. 3, pp. 192-198, 2012. |
| [14] | J. Freiknecht and W. Effelsberg, “A Survey on the Procedural Generation of Virtual Worlds,” *Multimodal Technologies and Interactio,* vol. 1, no. 4, 2017. |
| [15] | M. Brown, *How the Nemesis System Creates Stories,* Game Maker's Toolkit, 2021. |
| [16] | T. Short and T. Adams, Procedural generation in game design, CRC Press, 2017, p. 1. |
| [17] | M. Hendrik, S. Meijer, J. Van Der Velden and A. Iosup, “Procedural content generation for games: A survey,” *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM),* vol. 9, no. 1, pp. 1-22, 2013. |
| [18] | O. Stålberg, *Organic towns from square tiles,* Paris: IndieCade Europe, 2019. |
| [19] | C. Raux, T.-Y. Ma, R. Lemoy and N. Ovtracht, “Investigating land-use and transport interaction with an agent-based model,” in *The 13th World Conference on Transport Research*, Rio de Janerio, 2013. |
| [20] | R. Easton and P. Nabokov, Native American Architecture, Oxford University Press, 1989, pp. 76 - 80. |
| [21] | E. W. Burgess and R. E. Park, in *The City*, Chicago, University of Chicago Press, 1925, pp. 71-78. |
| [22] | Wodins, “Urban Development and the "Concentric Ring Theory".,” July 2012. [Online]. Available: https://wodinskeep.files.wordpress.com/2012/07/burgess.png. [Accessed 20 April 2021]. |
| [23] | E. M. Horwood and R. R. Boyce, Measurement of Central Business District Change and Urban Highway Impact, Seattle: University of Washington, 1959. |
| [24] | SunanneKN and 11gardir, “File:Core frame model.svg,” 28 October 2009. [Online]. Available: https://commons.wikimedia.org/wiki/File:Core\_frame\_model.svg. [Accessed 20 April 2021]. |
| [25] | H. Hoyt, The structure and growth of residential neighborhoods in American cities, US Government Printing Office, 1939. |
| [26] | Cieran 91 and SuzanneKn, “File:Hoyt model.svg,” 18 February 2008. [Online]. Available: https://commons.wikimedia.org/wiki/File:Hoyt\_model.svg. [Accessed 20 April 2021]. |
| [27] | C. D. Harris and E. L. Ullman, “The nature of cities,” *The Annals of the American Academy of Political and Social Science,* vol. 242, no. 1, pp. 7-17, 1945. |
| [28] | SuzanneKn, “File:Multiple nuclei model.svg,” 2009 October 2009. [Online]. Available: https://commons.wikimedia.org/wiki/File:Multiple\_nuclei\_model.svg. [Accessed 20 April 2021]. |
| [29] | S. Groenewegen, R. M. Smelik, K. J. de Kraker and R. Bidarra, “Procedural City Layout Generation Based on Urban Land Use Models.,” in *Groenewegen 2009 Procedural*, 2009. |
| [30] | T. Lechner, P. Ren, B. Watson, C. Brozefski and U. Wilenski, “Procedural modeling of urban land use,” in *Lechner 2006 Procedural*, 2006. |
| [31] | X. Lyu, Q. Han and B. de Vries, “Procedural modeling of urban layout: population, land use, and road network,” *Transportation research procedia,* vol. 25, pp. 3333--3342, 2017. |
| [32] | I. Elshamarka and A. B. S. Saman, “Design and implementation of a robot for maze-solving using flood-fill algorithm,” *International Journal of Computer Applications,* vol. 56, no. 5, 2012. |
| [33] | E.-M. Nosal, “Flood-fill algorithms used for passive acoustic detection and tracking,” 2008. |
| [34] | W. Newman and R. F. Sproull, Principles of Interactive Computer Graphics, McGraw-Hill Education (ISE Editions); International 2 Revised ed edition (1 Jun. 1979), 1979, p. 253. |
| [35] | M. P. Baker and D. Hearn, *Computer Graphics, C Version,* Pearson, 1996, pp. 129 - 130. |
| [36] | S. A. Ray, “Tint fill,” *SIGGRAPH '79: Proceedings of the 6th annual conference on Computer graphics and interactive techniques.,* p. 276–283, 1979. |
| [37] | J. Dunlap, “Queue-Linear Flood Fill: A Fast Flood Fill Algorithm,” Code Project, 15 November 2006. [Online]. Available: https://www.codeproject.com/Articles/16405/Queue-Linear-Flood-Fill-A-Fast-Flood-Fill-Algorith. [Accessed 2021 April 30]. |
| [38] | freeCodeCamp, “Boundary Fill Algorithm,” 25 January 2020. [Online]. Available: https://www.freecodecamp.org/news/boundary-fill-algorithm/#:~:text=Boundary%20fill%20is%20the%20algorithm,a%20stack%2Dbased%20recursive%20function.. |
| [39] | D. Henrich, “Space-efficient region filling in raster graphics,” *The Visual Computer,* vol. 10, no. 4, pp. 205-215, 1994. |
| [40] | Unity, “Texture2D.GetPixels32,” Unity, [Online]. Available: https://docs.unity3d.com/ScriptReference/Texture2D.GetPixels32.html. |
| [41] | L. Vandevenne, “Lode's Computer Graphics Tutorial - Flood Fill,” 2004 . [Online]. Available: https://lodev.org/cgtutor/floodfill.html. |
| [42] | J. R. Shaw, “QuickFill: An Efficient Flood Fill Algorithm,” Code Project, 12 March 2004. [Online]. Available: https://www.codeproject.com/Articles/6017/QuickFill-An-Efficient-Flood-Fill-Algorithm. |
| [43] | K. Oumghar, “Flood FIll algorithm (using C#.Net),” 29 December 2015. [Online]. Available: https://simpledevcode.wordpress.com/2015/12/29/flood-fill-algorithm-using-c-net/. |
| [44] | “Flood Fill,” Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Flood\_fill#Pseudocode. |
| [46] | Lee, “NYC Buildings over Time,” 5 May 2010. [Online]. Available: http://tier1dc.blogspot.com/2010/05/nyc-buildings-over-time.html. |

# Table of Figures

[Figure 1.1 City diagram proof of concept. 3](file:///C:\Users\Jack\Downloads\Dissertation%20Write%20Up%20-%20Jack%20Hopkins.docx#_Toc71155224)

[Figure 2.1 Evolution of NYC [46] 6](file:///C:\Users\Jack\Downloads\Dissertation%20Write%20Up%20-%20Jack%20Hopkins.docx#_Toc71155225)

[Figure 2.2 Evolution of the shape of the city (first line) and of the price of land as a function of the distance to the centre (second line) during a simulation. [19] 7](#_Toc71155226)

[Figure 2.3 Flood Fill across pixel spans for a 4-connected area. 10](file:///C:\Users\Jack\Downloads\Dissertation%20Write%20Up%20-%20Jack%20Hopkins.docx#_Toc71155227)

[Figure 3.1 Voxel Style Map which based Cube's Y value according to the luminoncity of the pixel (white would like to a higher Y value). 12](#_Toc71155228)

[Figure 3.2 On the left is the texture that was inputted. On the right is the output in Unity. The brown items are the houses. 13](file:///C:\Users\Jack\Downloads\Dissertation%20Write%20Up%20-%20Jack%20Hopkins.docx#_Toc71155229)

[Figure 3.3 Meshes being placed on the perimeter of the shape. 13](#_Toc71155230)

[Figure 3.4 An example of Perimeter fill working on a pixel art character I drew. This was a joke I sent to a few friends at the time, but in hindsight it is a very good example of showing the perimeter fill working. 14](file:///C:\Users\Jack\Downloads\Dissertation%20Write%20Up%20-%20Jack%20Hopkins.docx#_Toc71155231)

[Figure 3.5 Colours Array in Unity Inspector 15](#_Toc71155232)

[Figure 3.6 100x100px test of different shapes and colours 15](#_Toc71155233)

[Figure 3.7 Showing Flood Fill algorithms working on multiple shape types and colours. Notice how there is a different Knight mesh on each colour. There is a different Mesh aligned for each colour. 16](file:///C:\Users\Jack\Downloads\Dissertation%20Write%20Up%20-%20Jack%20Hopkins.docx#_Toc71155234)

[Figure 3.8 Full Town Plan 17](#_Toc71155235)

[Figure 3.9 Populated town in Unity. Notice the different buildings within different sectors. 17](#_Toc71155236)

[Figure 3.10 UML diagram of the code base 18](file:///C:\Users\Jack\Downloads\Dissertation%20Write%20Up%20-%20Jack%20Hopkins.docx#_Toc71155237)

[Figure 3.11 BuildingGenerator.cs in the Unity Inspector tab 19](#_Toc71155238)