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

By Jack Hopkins [180246647]

Project Supervisor: Dr Gary Ushaw

Number of Words: 12,352.

ABSTRACT

With market demand for open world video games increasing, the ability to create and prototype settlements for these games in a shorter timeframe is strongly valued. To retain creative control over the settlement's procedural generation, a diagram (bird's eye view) of the settlement is entered 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 it such as: Four Way – both recursive and linear implementations, Span Fill and Walk Based Fill.

CONTENTS

1 INTRODUCTION

In the current landscape of AAA video game releases, many are 3D open-world titles [1] such as, Skyrim, Witcher 3, GTA, Legend of Zelda: Breath of the Wild and many more. This also includes smaller studios and independant developers. However, the bigger these worlds get, the more man-hours are needed to produce quality environments in a such a time frame. This can result in the now infamous crunch culture, an example of which was CD Projeckt Red's *Cyberpunk2077,* [2] when the game's scope became so large that on launch day, the product was riddled with 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]. Other researchers have tried to address the issue using Perlin Noise [5], fractals, L-systems, tiling system, Voronoi texture basis [6] [7], and wave function collapse. [8]

However, instead of purely random generated settlements, what if we wanted to create settlements from a plan/diagram that would add constraints and structure? (Fig. 1) This would allow designers to have more control over the worlds they are creating. The aim here is to hit a middle ground where developers can still create interesting and unique settlements while also greatly speeding up the process and reducing man hours.

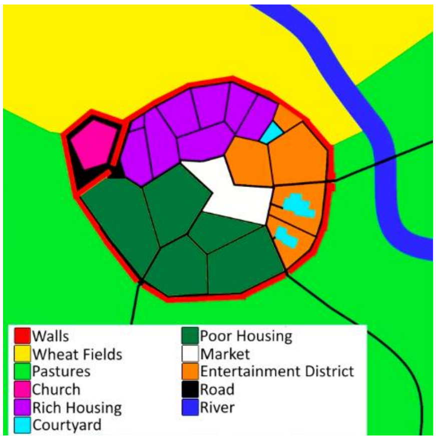


Figure 1.1 An early city diagram as a proof of concept.

Furthermore, the cost-effective nature of this proposal addresses a need within smaller/independent video game studios, who have fewer resources.

Furthermore, this tool could be used to quickly prototype and experiment with town layouts, making it useful for all types of game developer.

This project will go beyond existing works by focusing specifically on video game settlement creation. I will not be attempting to use video game graphical engines to create a city to simulate real world phenomena, [9] or use settlement creation in a specific video game. I will be using a predesigned map to guide the layout of the city rather than randomly generated one [6].

Personally, I am 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 specifically on using Flood-Fill algorithms to map out the sectors of the map. The 4 different Flood-Fill Algorithms (FFAs) that were used were:

- Four Way:
  - Recursive implementation (FWR).
  - Linear implementation (FWL).
- Walk-Based Filling [Fixed Memory Method] (WBF).
- Span Filling (SF).

And a Brute Force Algorithm (BF).

These models will be compared in 2 different categories:

- Memory use.
- Initial settlement creation speed.

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.

As we proceed through this dissertation, you will see that SF will is the best in most situations. However, there are many variables to be consider and each individual algorithm, even BF, is best suited for certain contexts.

This project will be using technologies such as:

- the Unity Engine as it is nice video game development tool that takes care a much of the level things that are not relevant to this project.
- Trello as a task and sprint managing tool.
- GitHub for version control.

1.1 Original Aims & Objectives

1.1.1 Objectives

1.1.1.1 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.

1.1.1.2 Research 4 Flood Fill algorithms.

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

1.1.1.3 *Implement Flood Fill algorithms.*

Program the project with the ability to easily select which algorithm to use. The algorithms will also be implemented in an efficient manner in order to produce results that are representative of the algorithm's ability.

1.1.1.4 *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.).

1.2 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.

2 BACKGROUND TECHNICAL MATERIAL

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 the format of diagram used for the settlements to be generated from.

2.1 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 my 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 to look at 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 Wave Function Collapse. [18]

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

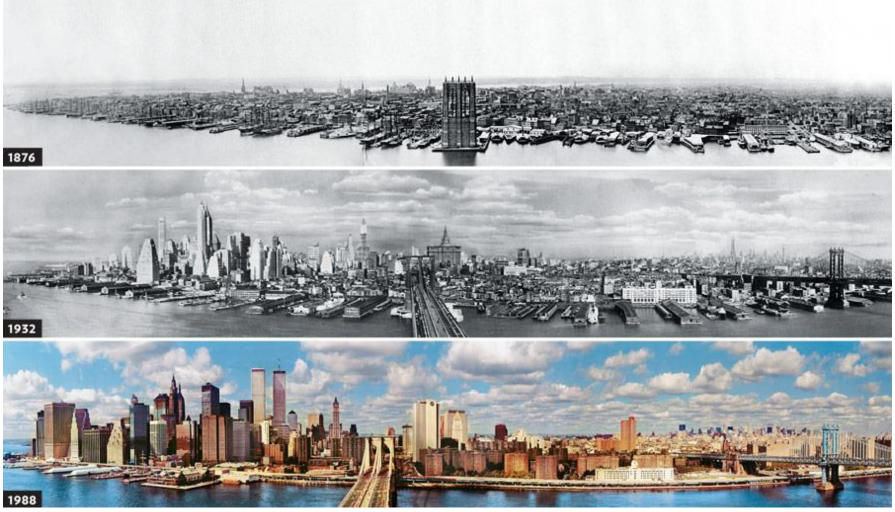


Figure 2.1 Evolution of NYC [53]

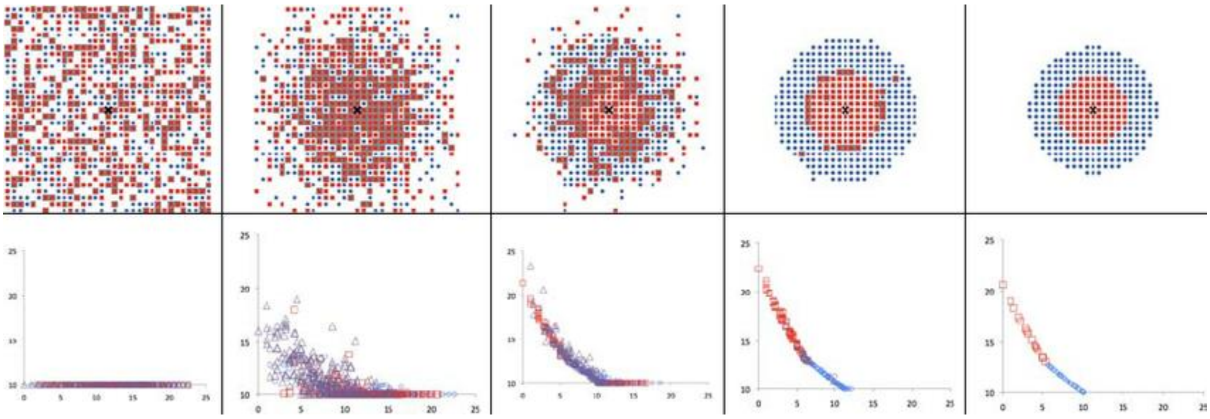


Figure 2.2 Evolution of the shape of the city (first line) and of the price of land as a function of the distance to the centre (second line) during a simulation. [19]

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 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 focussed 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.

2.2 LAND USE MODELS (LUMS)

Regarding the design of the diagram, it was based on the design of Land Use Models (LUMs) which are abstractions of complex city layouts. However, the simplification of each settlement's layout varies on the type of LUMs 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]

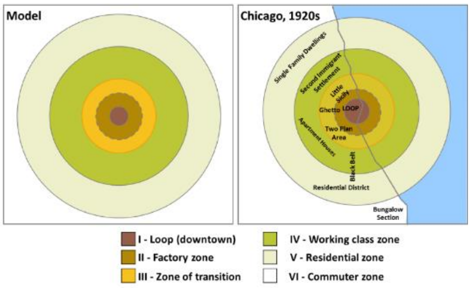


Figure 2.3 Burgess Model

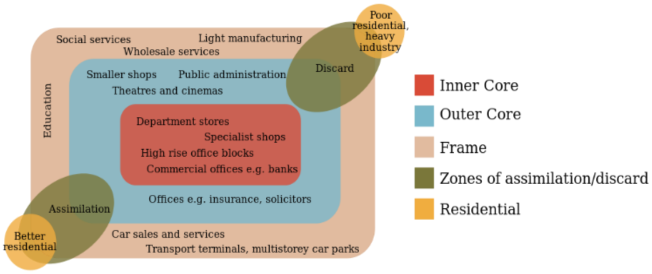


Figure 2.4 Core Frame Model

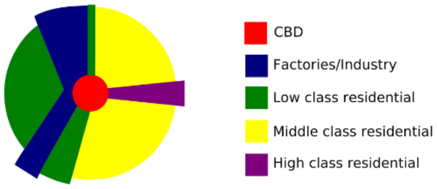


Figure 2.4 Sector Model

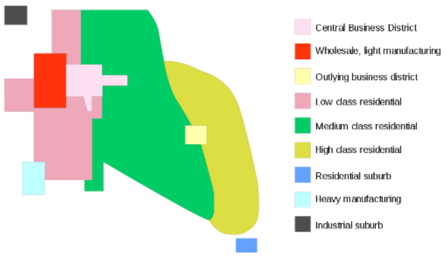


Figure 2.5 Multiple Nuclei Model

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

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 discretion 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 on 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 as 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 the results of my findings, I instead was trying to create a product that would be usable by developers in the future. That's not exactly what a dissertation was designed for.

2.3 Flood Fill Algorithms

Eventually, I landed on examining the use 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 when 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 of where flood fill algorithms have been applied are maze-solving [32] and passive audio detection either tracking a four-dimensional source(x, y, z, time) or a signal detection when applied to spectrograms [33].

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

2.4 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 does 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 (FWR) or linearly (FWL) 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]

2.5 Span Filling / Scanline

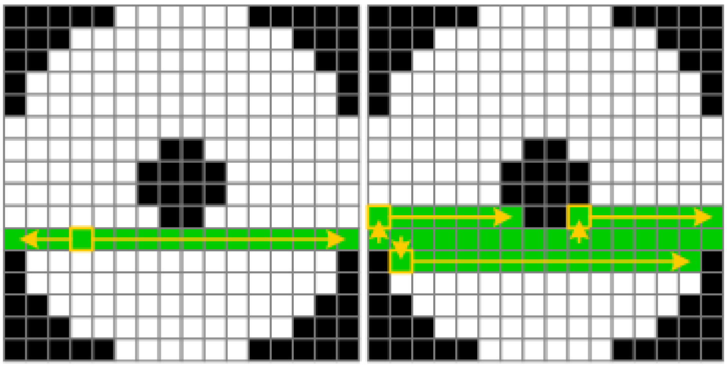
Span Filling (SP), also referred to as Scanline Fill, is an optimisation of the previous Four Way algorithm. Instead of filling pixel by pixel, it fills a whole span at a time.

Initially, it was described by Alvy Ray Smith (who referred to it as Tint Fill) like this:

> *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 as such:

Starting with a seed point, the algorithm fills left and right of it until it hits an 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 use of spans within the context of Boundary Fill - essentially Flood Fill when you are only trying to find the boarders of a polygon. [38] It is mostly thought of as the state-of-the-art flood fill algorithm, sometimes being up to 7 times faster then your basic four way methods. [39] [40].

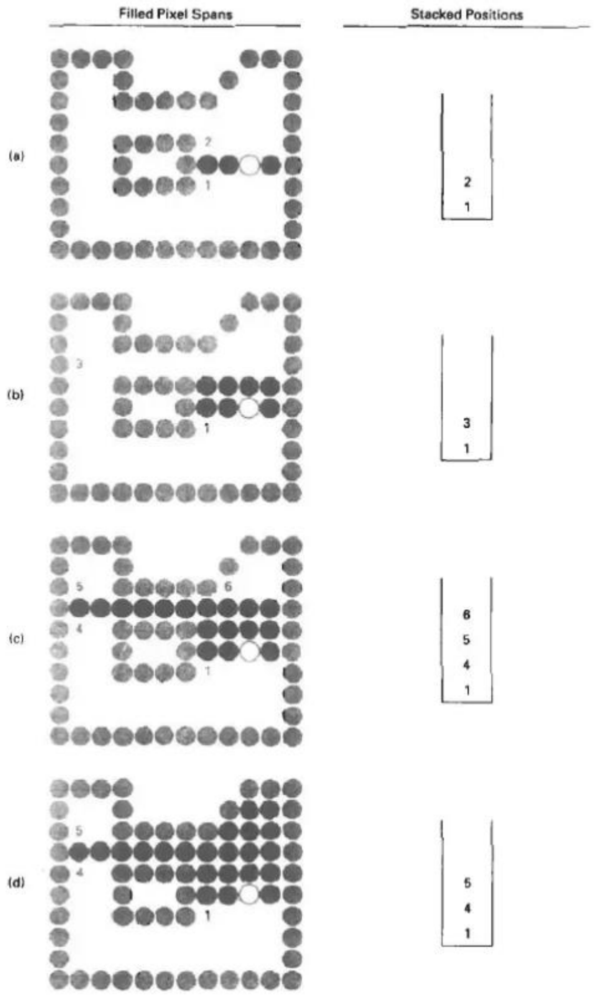


Figure 2.3 Flood Fill across pixel spans for a 4-connected area.

(a) 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.

(b) Filled pixel span on the first scan line above the initial scan line and the current contents of the stack.

(c) Filled pixel spans on the first two scan line and the current contents of the stack.

(d) Completed pixel spans for the upper-right portion of the defined region and the remaining stacked positions to be processed.

[35]

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

Walk Based Filling (WBF), or what I like to refer to as: 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. [41]*

Essentially, from the initial seed point, the cursor (or the current pixel being examined) 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 directly 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.

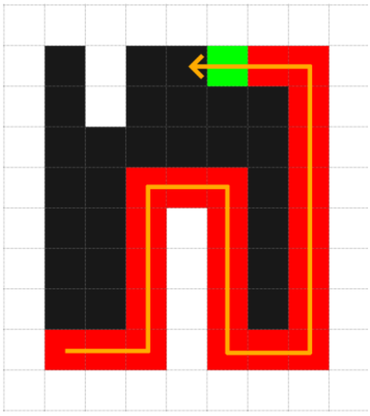


Figure 2.4 Example of following the RHR. Green pixel is the cursor. Red are the painted pixels.

This leads to the cursor finding itself in one of these conditions:

0. Zero boundary pixels are filled.
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.

Add Images:

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

Case #0: Move forward until you find a boarder. Does not paint until boarder is found.

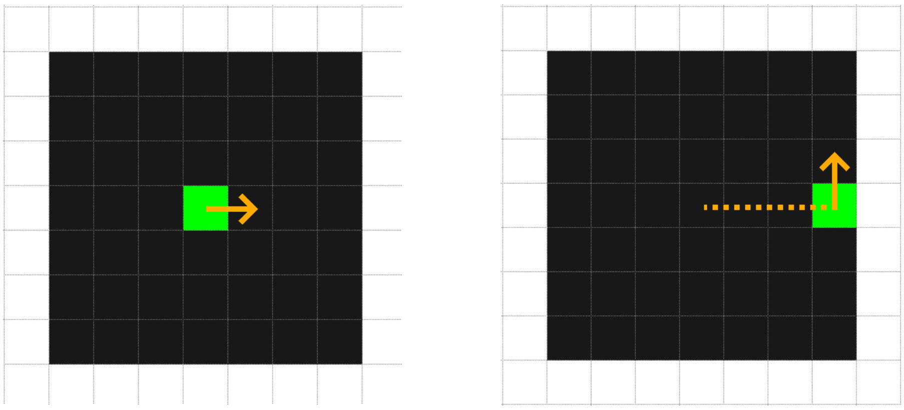


Figure 2.5 Left: Iteration 1. Right: Iteration 4; starts painting.

Case #1: Check the front 8-corners (front left and front right) to see whether they are available or not. If both are available, 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.

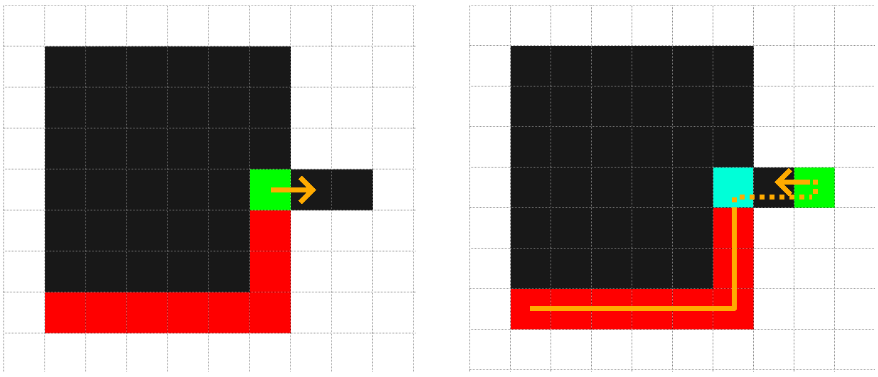


Figure 2.6 Algorithm avoiding painting itself into a corner as seen in this example with a dead-end.

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 in 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 encounters the mark again 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 encounters another case 2, then it places done a second mark instead of moving the first.

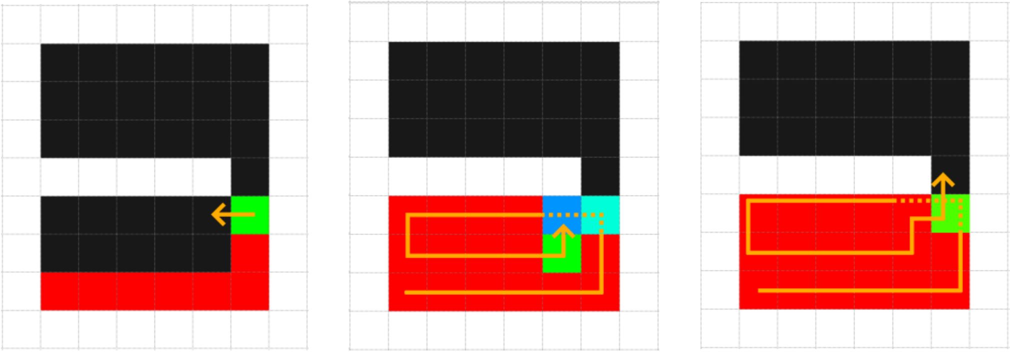


Figure 2.7 Scenario when 2 different marks must be placed down.

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

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. [41] This algorithm, while focusing on a fixed memory solution, has to sacrifice speed since it can often be caught in a loop before closing them, examining the same pixels multiple times in the process.

3 IMPLEMENTATION

This chapter will detail the journey of the project's implementation from start to finish.

3.1 OVERVIEW

At the start of my implementation, I had to select how I wanted to approach the project:

- 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, I ended up deciding against it. The project eventually moved towards the idea of using pre-existing meshes to create a settlement. These were tools that Unity could use without any additional coding. Furthermore, I have much more experience with Unity as a technology then with OpenGL.

Unity is also currently a much more favoured video game development engine for indie programmers which is one of the main demographics my project designed for. The GMTK Jam 2020, the largest game Jam ever (a total of 5477 games) [42], had 62.2% of the games submitted using Unity. [43]

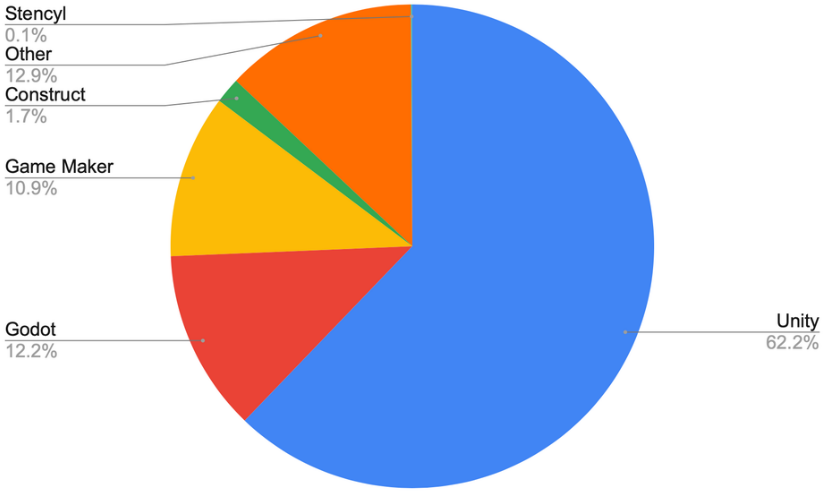


Figure 3.1 Breakdown of the popularity of different game engines in the GMTK Jam 2020

When starting my project, I came across the `Color` class and it's `GetPixel()` function within the UnityEngine package. I initially used this when I was trying to use monochrome textures to create height maps. This was before I had landed on the LUM style of diagram (Figure 3.2). This was then represented by cubes floating in the air and ended up looking like some sort of voxel map.

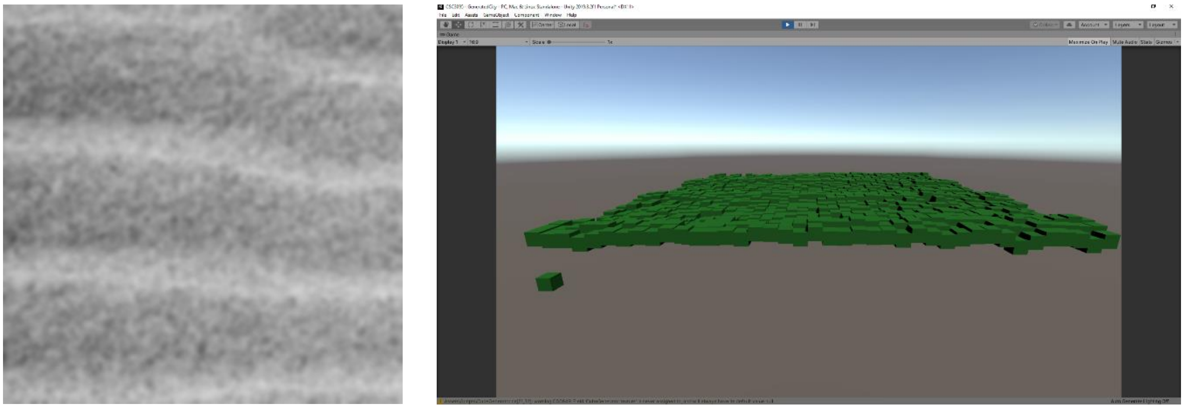


Figure 3.2 On the left is a height map (grayscale picture of sand) and on the right is a Voxel Style Map which based Cube's Y value according to the luminance of the pixel (white would like to a higher Y value).

Next, I started 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 textures with black lines across them. The aim was to stop the houses spawning on top of these black lines. (Figure 3.3)

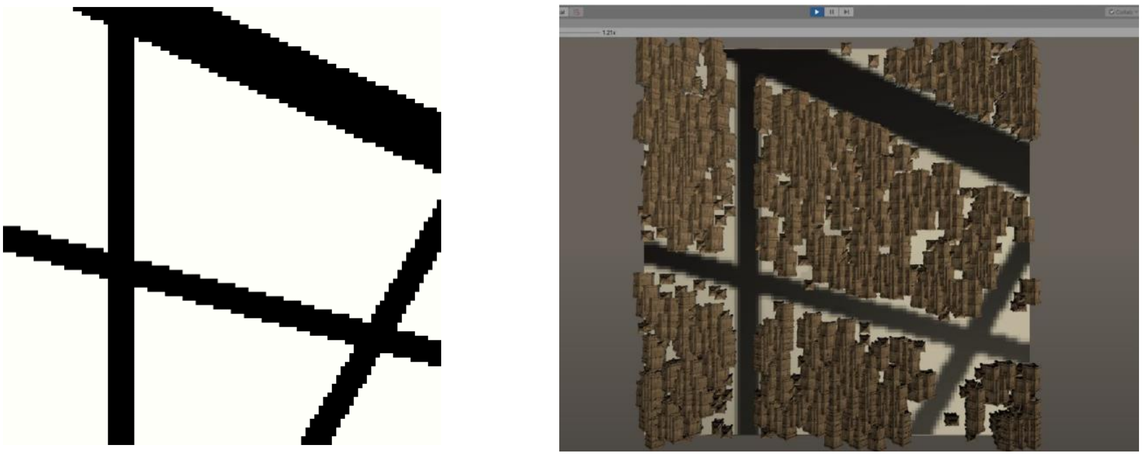


Figure 3.3 On the left is the texture that was passed in. On the right is the output in Unity. The brown items are the houses.

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

This then led me to finally finding about flood fill algorithms. From here, I implemented the simple four-way algorithm (FWR). I found it to be a little slow and came into conflict with the dreaded "Stack Overflow". Therefore, I pivoted to implementing the Linear version of the four-way method (FWL) for the time being. I then adapted the algorithm so that it would only spawn meshes on the perimeter of the shape (Figure 3.4). This functionally was not needed in the end since I later realised that allowing the meshes to be placed anywhere within the sector, made more sense. I was implementing a Boundary Fill algorithm [38], instead of a proper flood fill algorithm.

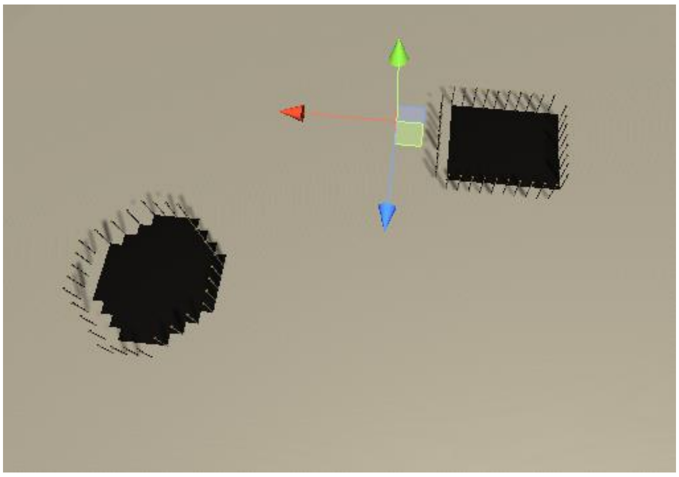


Figure 3.4 Meshes being placed on the perimeter of the shape.

Between implementing the FWL and coming to conclusion to not use just the perimeter, I had developed a method of perimeter checking on my own. FWL was currently inefficient due to how *FloodFill.cs* was set up. You do not manually pass in an initial seed value as you typically do with most flood fill algorithms and instead find it automatically by iterating through every pixel in the texture. Therefore, I theorised that my own algorithm could be more efficient. This was what is now referred to as Perimeter Fill (PF).

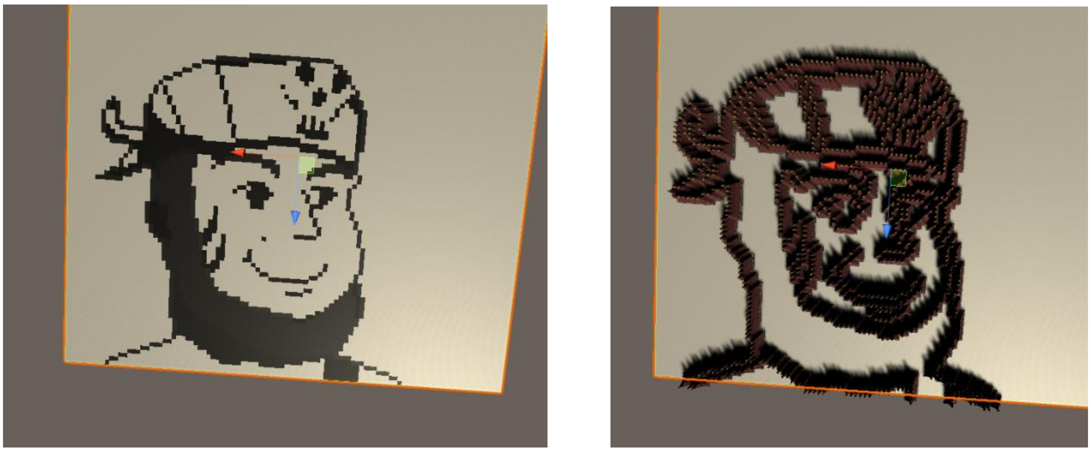
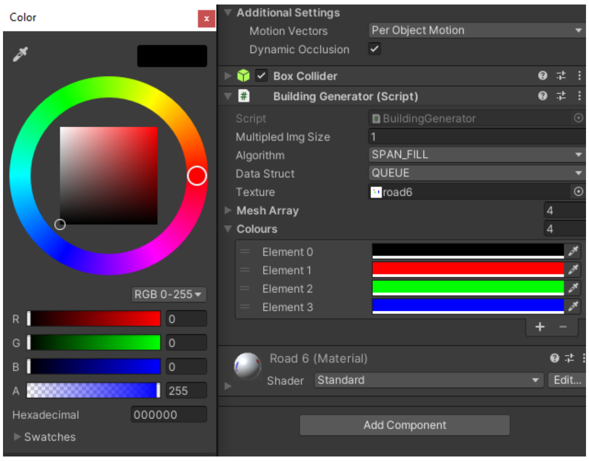


Figure 3.5 An example of Perimeter fill working on a pixel art character I drew. This was a joke I sent to a few friends at the time, but in hindsight it is a very good example of showing perimeter fill working.

From this, I started implementing the rest of the flood fill algorithms, first starting with SF as it seemed like a simple and naturel progression from FWR and FWL. Instead of adding pixel by pixel, it added row by row. Furthermore, it should be much faster in theory because it did not have to check each pixel up to 4 times and instead read a pixel a maximum of 3 times, still inefficient but an improvement. [44]

With these two algorithms under my belt, I started looking into more complicated shapes then a simple line. This is when, as a joke, I took a picture of a pixel art character I had drawn and then got it to place meshes around its boarders, effectively taking a picture out of building parts. (Figure 3.5)

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 their colours of choice. This is where Unity's Engine came in super useful, not only could you input your precise colours into the array using either RGB or HSV, but you could also use the colour picker tool. (Figure 3.6).



Color

RGB 0-255

R 0

G 0

B 0

A 255

Hexadecimal 000000

Swatches

Additional Settings

Motion Vectors: Per Object Motion

Dynamic Occlusion: ✓

✓ Box Collider

Building Generator (Script)

Script: BuildingGenerator

Multipled Img Size: 1

Algorithm: SPAN_FILL

Data Struct: QUEUE

Texture: road6

Mesh Array: 4

Colours: 4

Element 0 ⬛

Element 1 🟥

Element 2 🟩

Element 3 🟦

Road 6 (Material)

Shader: Standard

Figure 3.6 Colours Array in Unity Inspector

The first test of multiple colours had 4 RGBA values:

- Black [0,0,0,255]

- Red [255,0,0,255]

- Green [0,255,0,255]

- Blue [0,0,255,255]

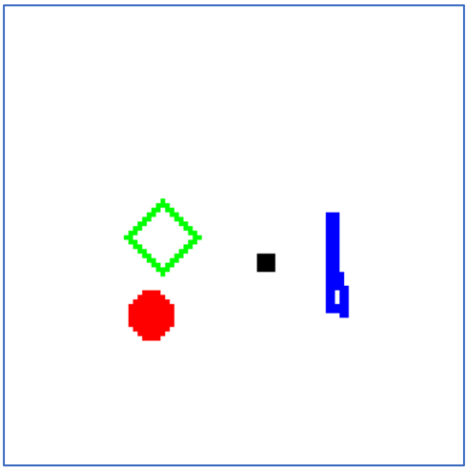


Figure 3.7 100x100px test of different shapes and colours

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

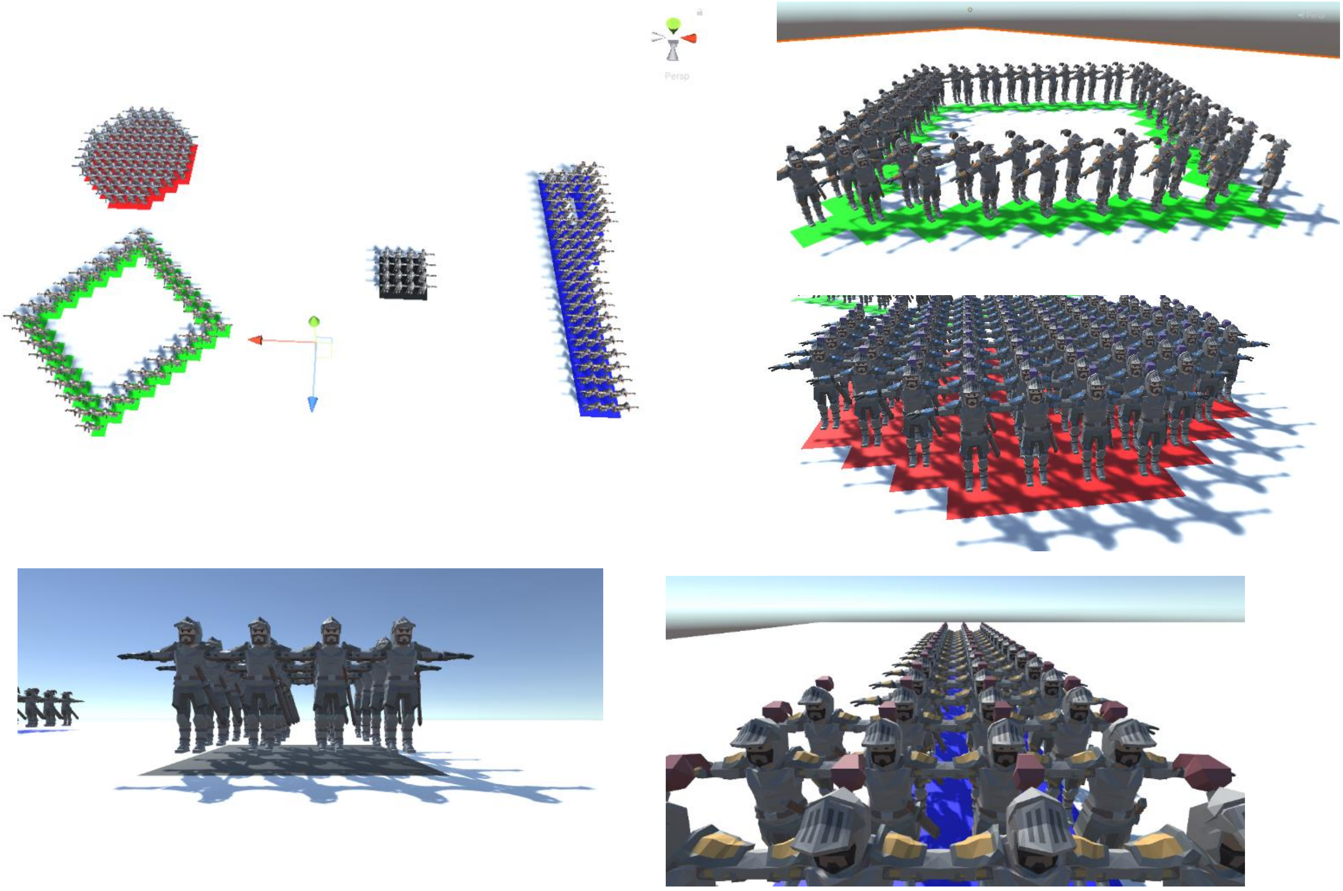


Figure 3.8 Showing Flood Fill algorithms working on multiple shape types and colours. Notice how there is a different Knight mesh on each colour. There is a different Mesh aligned for each colour.

However, the methods implement thus far were quite memory intensive. This is when I started to implement the WBF method. This would be by far that hardest to implement since its logic was not simple. It had multiple cases to take into consideration and more complex pseudo-code which I had to change from my initial sources. Furthermore, due to its complexity and many while loops, it was the hardest to debug. Many a time with Unity freezing on start-up; pressing me force quit 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. (This is further discussed in 3.7.1).

I then created a proper test town with 6 different colours. All colours represented different areas of town, whether it be roads, poor housing, entertainment districts, walls, etc. and tested it to see if all the algorithms worked. [See more in chapter 4]

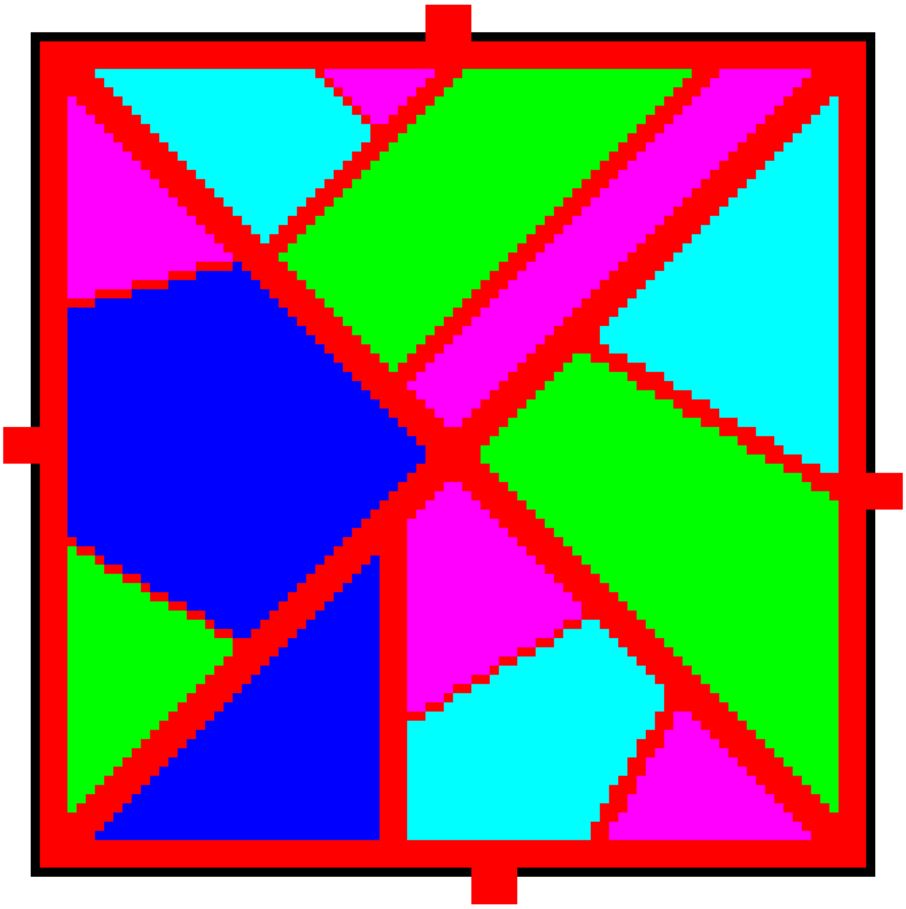


Figure 3.9 Full Town Plan

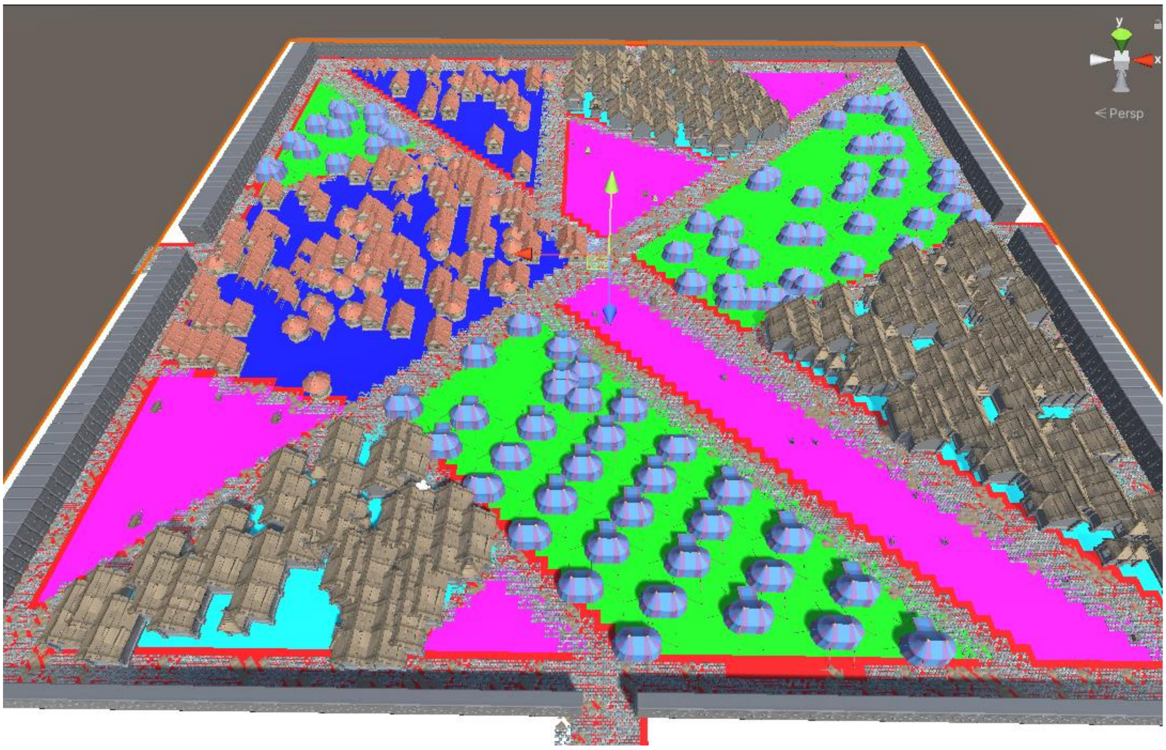


Figure 3.10 Populated town in Unity. Notice the different buildings within different sectors.

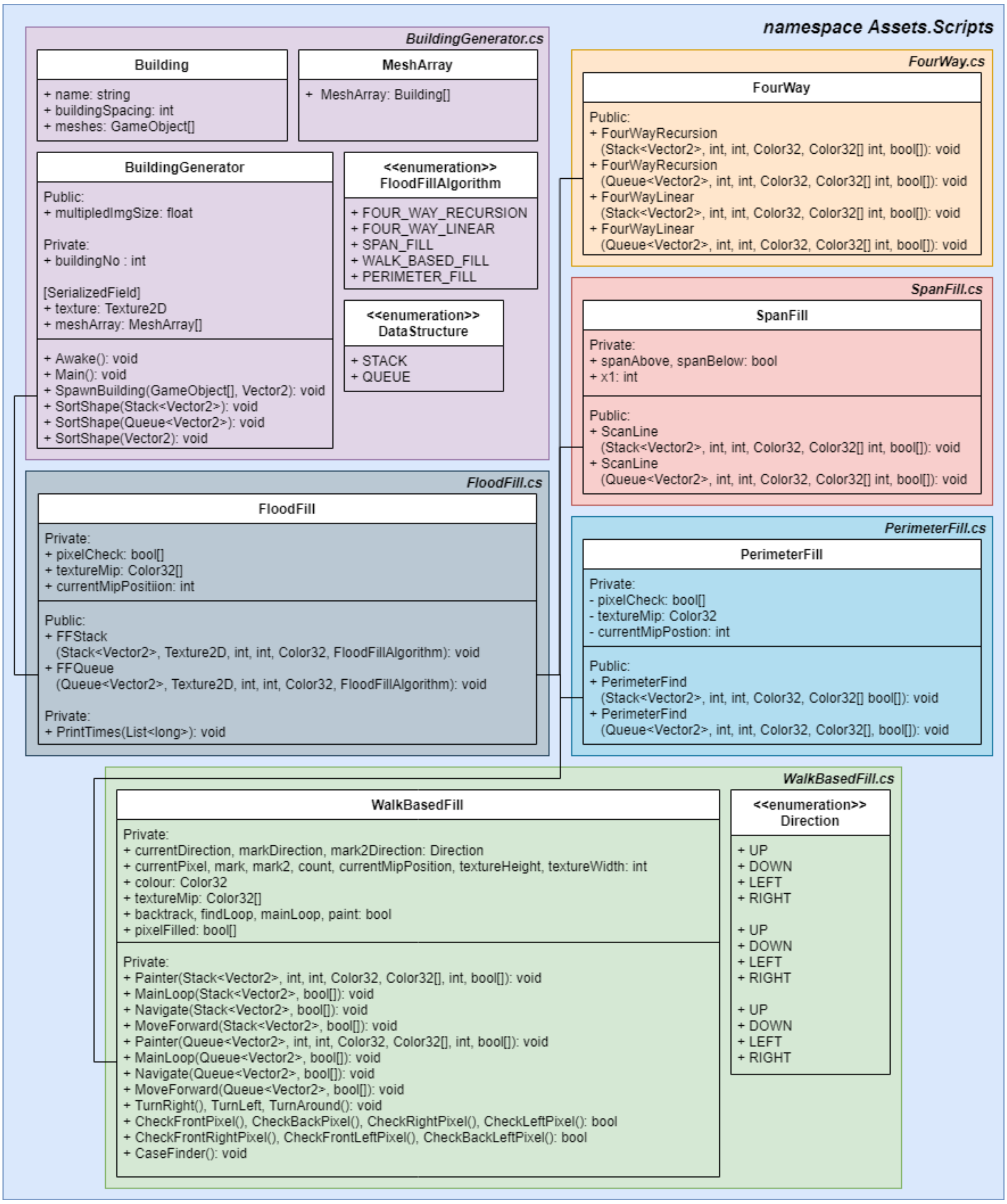

namespace Assets.Scripts
BuildingGenerator.cs
Building
+ name: string
+ buildingSpacing: int
+ meshes: GameObject[]
MeshArray
+ MeshArray: Building[]
BuildingGenerator
Public:
+ multipledImgSize: float
Private:
+ buildingNo : int
[SerializedField]
+ texture: Texture2D
+ meshArray: MeshArray[]
+ Awake(): void
+ Main(): void
+ SpawnBuilding(GameObject[], Vector2): void
+ SortShape(Stack<Vector2>): void
+ SortShape(Queue<Vector2>): void
+ SortShape(Vector2): void
<<enumeration>>
FloodFillAlgorithm
+ FOUR_WAY_RECURSION
+ FOUR_WAY_LINEAR
+ SPAN_FILL
+ WALK_BASED_FILL
+ PERIMETER_FILL
<<enumeration>>
DataStructure
+ STACK
+ QUEUE
FloodFill.cs
FloodFill
Private:
+ pixelCheck: bool[]
+ textureMip: Color32[]
+ currentMipPositiion: int
Public:
+ FFStack
(Stack<Vector2>, Texture2D, int, int, Color32, FloodFillAlgorithm): void
+ FFQueue
(Queue<Vector2>, Texture2D, int, int, Color32, FloodFillAlgorithm): void
Private:
+ PrintTimes(List<long>): void
FourWay.cs
FourWay
Public:
+ FourWayRecursion
(Stack<Vector2>, int, int, Color32, Color32[] int, bool[]): void
+ FourWayRecursion
(Queue<Vector2>, int, int, Color32, Color32[] int, bool[]): void
+ FourWayLinear
(Stack<Vector2>, int, int, Color32, Color32[] int, bool[]): void
+ FourWayLinear
(Queue<Vector2>, int, int, Color32, Color32[] int, bool[]): void
SpanFill.cs
SpanFill
Private:
+ spanAbove, spanBelow: bool
+ x1: int
Public:
+ ScanLine
(Stack<Vector2>, int, int, Color32, Color32[] int, bool[]): void
+ ScanLine
(Queue<Vector2>, int, int, Color32, Color32[] int, bool[]): void
PerimeterFill.cs
PerimeterFill
Private:
- pixelCheck: bool[]
- textureMip: Color32
- currentMipPostion: int
Public:
+ PerimeterFind
(Stack<Vector2>, int, int, Color32, Color32[] bool[]): void
+ PerimeterFind
(Queue<Vector2>, int, int, Color32, Color32[], bool[]): void
WalkBasedFill.cs
WalkBasedFill
Private:
+ currentDirection, markDirection, mark2Direction: Direction
+ currentPixel, mark, mark2, count, currentMipPosition, textureHeight, textureWidth: int
+ colour: Color32
+ textureMip: Color32[]
+ backtrack, findLoop, mainLoop, paint: bool
+ pixelFilled: bool[]
Private:
+ Painter(Stack<Vector2>, int, int, Color32, Color32[], int, bool[]): void
+ MainLoop(Stack<Vector2>, bool[]): void
+ Navigate(Stack<Vector2>, bool[]): void
+ MoveForward(Stack<Vector2>, bool[]): void
+ Painter(Queue<Vector2>, int, int, Color32, Color32[], int, bool[]): void
+ MainLoop(Queue<Vector2>, bool[]): void
+ Navigate(Queue<Vector2>, bool[]): void
+ MoveForward(Queue<Vector2>, bool[]): void
+ TurnRight(), TurnLeft, TurnAround(): void
+ CheckFrontPixel(), CheckBackPixel(), CheckRightPixel(), CheckLeftPixel(): bool
+ CheckFrontRightPixel(), CheckFrontLeftPixel(), CheckBackLeftPixel(): bool
+ CaseFinder(): void
<<enumeration>>
Direction
+ UP
+ DOWN
+ LEFT
+ RIGHT
+ UP
+ DOWN
+ LEFT
+ RIGHT
+ UP
+ DOWN
+ LEFT
+ RIGHT

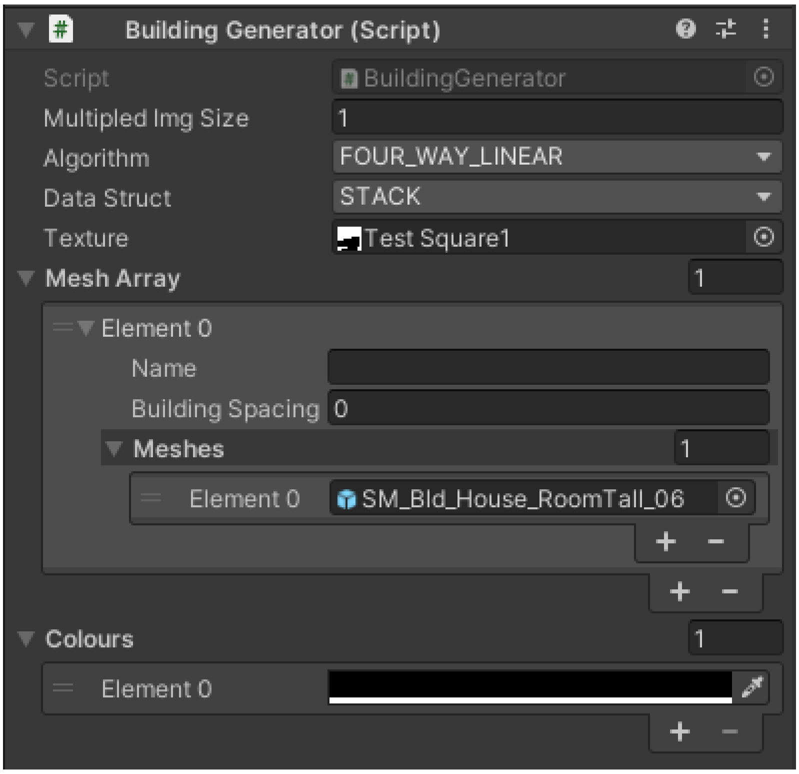
Figure 3.11 UML diagram of the code base

3.2 BuildingGenerator.cs

As seen in the UML diagram above (Figure 3.11) the main script is *BuildingGenerator.cs*. This script is attached to the scene object that is representing the stage that the settlement will be spawn on. 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 (FWR, FWL, SP, WBF, BF). This is done via an Enum.

- Select which data structure you would like to use (Stack or Queue). This is also done via an Enum.
- An 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.
- An array of Colours that correspond to each sector.



Building Generator (Script)

Script: BuildingGenerator

Multipled Img Size: 1

Algorithm: FOUR_WAY_LINEAR

Data Struct: STACK

Texture: Test Square1

Mesh Array: 1

Element 0

Name:

Building Spacing: 0

Meshes: 1

Element 0: SM_Bld_House_RoomTall_06

Colours: 1

Element 0

Figure 3.12 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. It iterates through every colour in the *Color32[]* finding the pixels of the selected colour and add it to either a *Stack<Vector2>* or *Queue<Vector2>*. *SortShape()* is then called, followed by *SpawnBuilding()*. *SpawnBuilding()* spawns a single mesh and within each *Building* object there is the variable, *buildingSpacing*, 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 instead of the whole area. Thus, this algorithm is not optimised and instead it was an attempt to get the job done when using *PerimiterCheck()*.

Pseudo-Code:

```
sortShape(Vector2[] shape) :
    for each Pixel i
        set yAxis to 0
        set diagonal to 0
        if (next i coord is (x+1 OR x-1, y))
            continue to next For Loop iteration
        endIF

        for each Pixel j
            if (next j coord is (x+1 OR x-1, y))
                Swap j and i in the Array
            endIf

            if (next j coord is (x, y+1 OR y-1))
                set yAxis to j
                continue to next For Loop iteration
            endIf

            if (next i coord is (x+1 OR x-1, y+1 OR y-1))
                set diagonal to j
            endIf
        endForLoop

        if (next i coord is (x, y+1 OR y-1))
            continue to next For Loop iteration
        endIf

        if (next i coord is (x, y+1 OR y-1))
            Swap j and i in the Array
            continue to next For Loop iteration
        endIf

        if (next coord is (x+1 OR x-1, y+1 OR y-1))
            continue to next For Loop iteration
        endIf

        if (next coord is (x+1 OR x-1, y+1 OR y-1))
            Swap j and i in the Array
            continue to next For Loop iteration
        endIf
    endForLoop
```

This algorithm was needed for when each building wasn't spawn right next to each other (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. [45]*

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 spawn.

3.3 FloodFill.cs

From *BuildingGenerator.cs* (within *SpawnBuildingFloodFill()*), *FloodFill.cs* (either *FFStack()* or *FFQueue() depending of the data structure you want to use*) 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 initial seed assigned as the pixel where the mouse / paint bucket is clicked upon [34]. 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 to called *Texture2D.GetPixels32()* once then *Texture2D.GetPixel()* repeatedly. [45]

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 merely 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, in 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.

3.4 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) BF, surprisingly, seemed to compete quite well with the other algorithms, even outperforming them in certain scenarios. The reasoning why was because the initial seed (pixel) had to be found automatically and was not inputted manually (see chapter 3.3). If all the shapes had seeds that were entered in manually then it would be unlikely that BF compete.

3.4.1 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
```

3.5 FourWay.cs

As shown in Chapter 2.3, you can have either a recursive implementation or a linear implementation of the four-way algorithm. FWL is almost always preferable because FWR often leads to you blowing your stack, leading to a *StackOverflow* error. I implemented both because I wanted to test this out, mainly because in my research the pseudo code given was often 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 linearly. [39] [46]

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).

3.5.1 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. Therefore, I could not distinguish between *x* and *y* values. 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 track of which pixel we were currently inspecting.

3.5.2 Pseudo-Code / Code: Linear Implementation

This Pseudo code was based of the code from Karim Oumghar. [47]

```
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;
```

3.6 SPANFILL.CS

Span Fill has the same alterations from the algorithms discussed above. As discussed in chapter 2.5, SF works by creating spans above and below the current line being examined. This code uses two bools *spanAbove* and *spanBelow* to keep track if there is already a span either above or below the current pixel, making sure that the cursor does not go out of bounds.

3.6.1 Pseudo-Code: Scan Line

This Pseudo code was based on that from Lode Vandevenne [39] and Karim Oumghar [47].

```
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;
```

3.7 WalkBasedFilling.cs

There was not a whole lot of implementation of WBF 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 were of less quality which lead me to altering some parts of the pseudo-code.

3.7.1 Pseudo-Code: Painter / Walk Based Filling

This Pseudo code was based off that being discussed in *Space-efficient region filling in raster graphics* [41] and provided from the Wikipedia page on Flood Fill Algorithms [48] with some major alterations.

Firstly, is the addition of Case 0. Where the cursor will just continue forward until it hits a boundary (i.e. 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, instead of just *count != 4*.

Secondly, an *if* statement in both Case 1 and Case 2 was added. I check if the front corner pixels (front-left and front-right) are available. If both are not available, while the pixel directly in front is, this indicate there is a pixel wide bottle neck. Then a mark has not been placed down (*mark* is -1 or null) is placed. Once in, the cursor moves forward without painting until it cannot 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* statements implicitly baked in to it.

```
Painter(Shape):
    set cur to starting pixel
    set curDir 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 adja-
                              cent 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 markDir to curDir
                clear mark2
                set findloop and backtrack to false
            else
                if mark2 is not set then
                    if cur is at mark then
                        if curDir is the same as markDir then
                            clear mark
                            turn around
                            set cur
                            Paint();
                        else
                            set backtrack to true
                            set findloop to false
                            set curDir to markDir
                        end if
                    else if findloop is true then
                        set mark2 to cur
                        set mark2Dir to curDir
                    end if
                else
                    if cur is at mark then
                        set cur to mark2
                        set curDir to mark2Dir
                        clear mark and mark2
                        set backtrack to false
                        turn around
                        set cur
                        Paint();
                    else if cur at mark2 then
                        set mark to cur
                        set curDir and markDir to mark2Dir
                        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
```

3.8 PERIMETERCHECK.CS

Perimeter fill is very simple to understand. You check every pixel in the texture (one by one) and when you meet a pixel that has an equal colour value to the colour you are looking for, it then checks the four pixels around it (like four-way) to see if any of them 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.

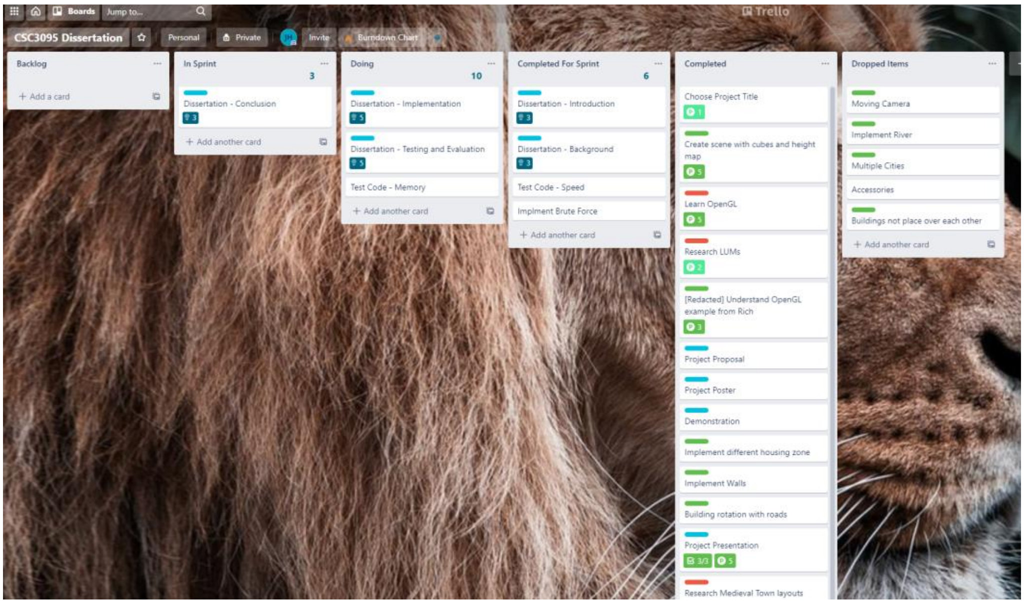
3.8.1 Pseudo-Code: PerimeterCheck:

```
PerimeterCheck(Shape):
      For every pixel in texture
            if pixel is not colour
                  if North pixel is with bounds && North pixel is colour &&
                  and not checked
                        add to Shape
                        set pixel to checked
                  endIf
                  if South pixel is with bounds && North pixel is colour &&
                  and not checked
                        add to Shape
                        set pixel to checked
                  endIf
                  if East pixel is with bounds && North pixel is colour &&
                  and not checked
                        add to Shape
                        set pixel to checked
                  endIf
                  if West pixel is with bounds && North pixel is colour &&
                  and not checked
                        add to Shape
                        set pixel to checked
                  endIf
            endIF
            set pixel to checked
      endFor
```

3.9 OTHER TECHNOLOGIES / ASSETS USED

3.9.1 Trello

Over the course of the project, I used Trello for project management. It allowed me to work in an agile way with a Canban-Sprint style of workflow (bi-weekly sprints). The point system allowed me to measure my velocity to see my current progress and rate of work.



Boards Jump to...

CSC3095 Dissertation Personal Private Invite Burndown Chart

Trello

Backlog

+ Add a card

In Sprint 3

- Dissertation - Conclusion 3

+ Add another card

Doing 10

- Dissertation - Implementation 5
- Dissertation - Testing and Evaluation 5
- Test Code - Memory

+ Add another card

Completed For Sprint 6

- Dissertation - Introduction 3
- Dissertation - Background 3
- Test Code - Speed
- Implment Brute Force

+ Add another card

Completed

- Choose Project Title 1
- Create scene with cubes and height map 5
- Learn OpenGL 5
- Research LUMs 2
- [Redacted] Understand OpenGL example from Rich 3
- Project Proposal
- Project Poster
- Demonstration
- Implement different housing zone
- Implement Walls
- Building rotation with roads
- Project Presentation 3/3 5
- Research Medieval Town layouts

Dropped Items

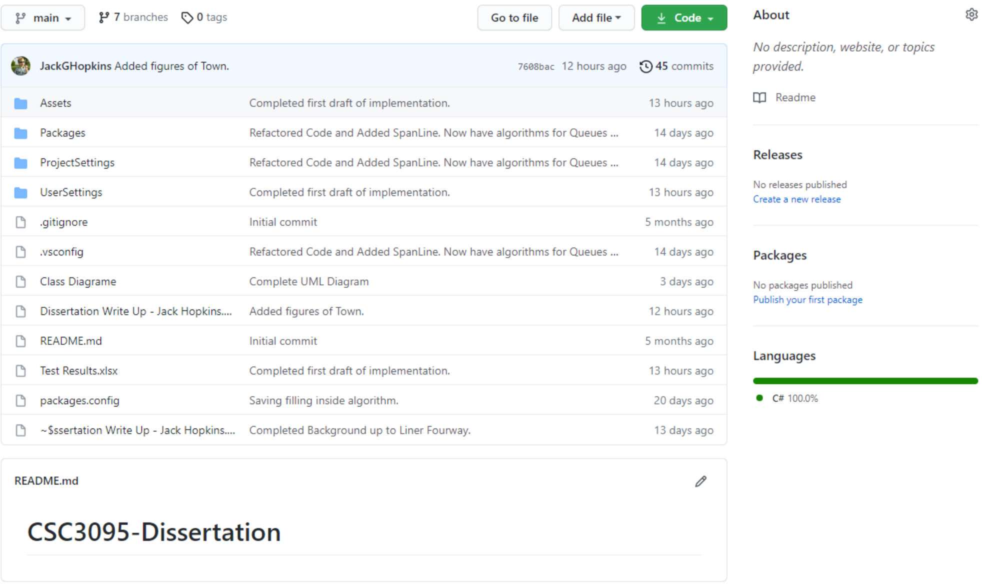
- Moving Camera
- Implement River
- Multiple Cities
- Accessories
- Buildings not place over each other

+ Add another card

Figure 3.13 Picture of my Trello board

3.9.2 GitHub (Version Control)

I was working from many different stations, whether it be at home, or on campus at either the Urban Science Building or Fredrick Douglas. Github and Git Bash allow me to seamlessly update my directories and work on multiple parts of the project at once using different branches and pull requests. Furthermore, it allowed me to go back a take pictures of my progress once I started writing my dissertation. It acted as a contingency plan if I lost my work for whatever reason.



main 7 branches 0 tags Go to file Add file Code

JackGHopkins Added figures of Town. 7608bac 12 hours ago 45 commits

Assets	Completed first draft of implementation.	13 hours ago
Packages	Refactored Code and Added SpanLine. Now have algorithms for Queues ...	14 days ago
ProjectSettings	Refactored Code and Added SpanLine. Now have algorithms for Queues ...	14 days ago
UserSettings	Completed first draft of implementation.	13 hours ago
.gitignore	Initial commit	5 months ago
.vsconfig	Refactored Code and Added SpanLine. Now have algorithms for Queues ...	14 days ago
Class Diagrame	Complete UML Diagram	3 days ago
Dissertation Write Up - Jack Hopkins....	Added figures of Town.	12 hours ago
README.md	Initial commit	5 months ago
Test Results.xlsx	Completed first draft of implementation.	13 hours ago
packages.config	Saving filling inside algorithm.	20 days ago
~$ssertation Write Up - Jack Hopkins....	Completed Background up to Liner Fourway.	13 days ago

README.md

CSC3095-Dissertation

About

No description, website, or topics provided.

Readme

Releases

No releases published
Create a new release

Packages

No packages published
Publish your first package

Languages

C# 100.0%

Figure 3.14 GitHub repository

3.9.3 Polygon Knights Asset pack

This was the asset pack I used during my assessment. I used their breath of meshes to create multiple sectors [49] allowing me to focus entirely on the coding and research aspect of the project.

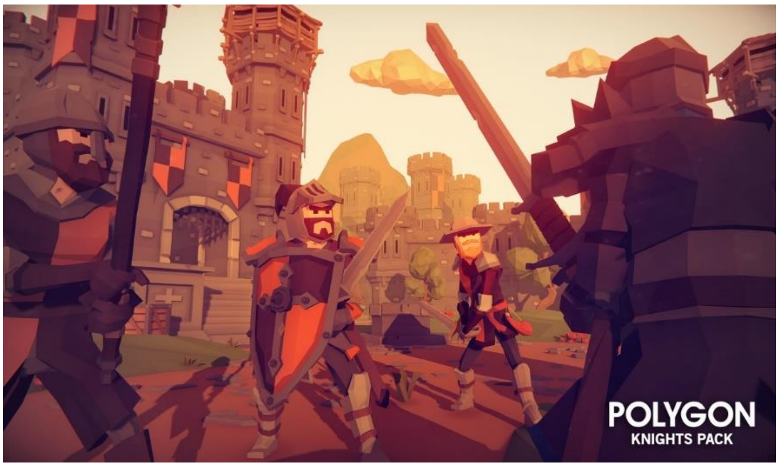


Figure 3.15 POLYGON Knights picture demonstration. [49]

4 RESULTS AND EVALUATION

With the wide number of algorithms, the number of different possible tests were many. 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.

These two different categories should give us a comprehensive look at each algorithm's strengths, as suggested by previous literature (see chapter 0).

4.1 STACK VS QUEUE

The first main test that needed to be carried out, was on which data structure would be used to hold the shape's pixels. The main to contestants, as stated in chapter 2.4, was the stack and queue data structures. Here are some of the results:

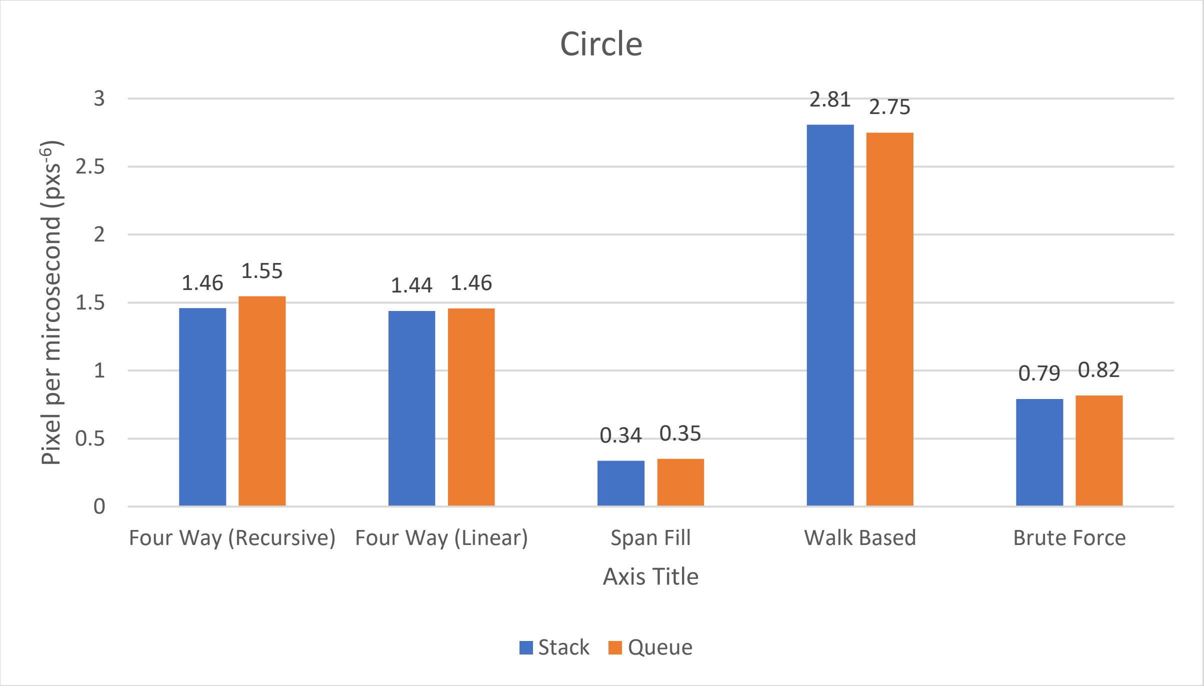


Figure 4.1 Stack vs Queue - Circle Test

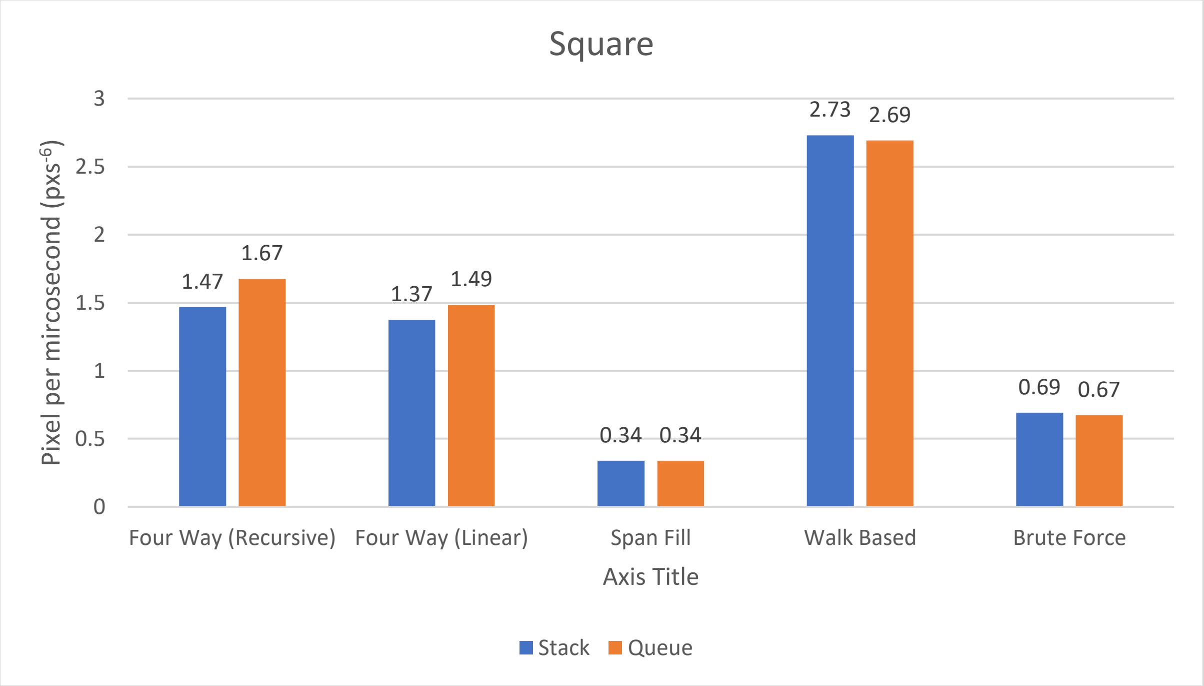


Figure 4.2 Stack vs Queue - Square Test

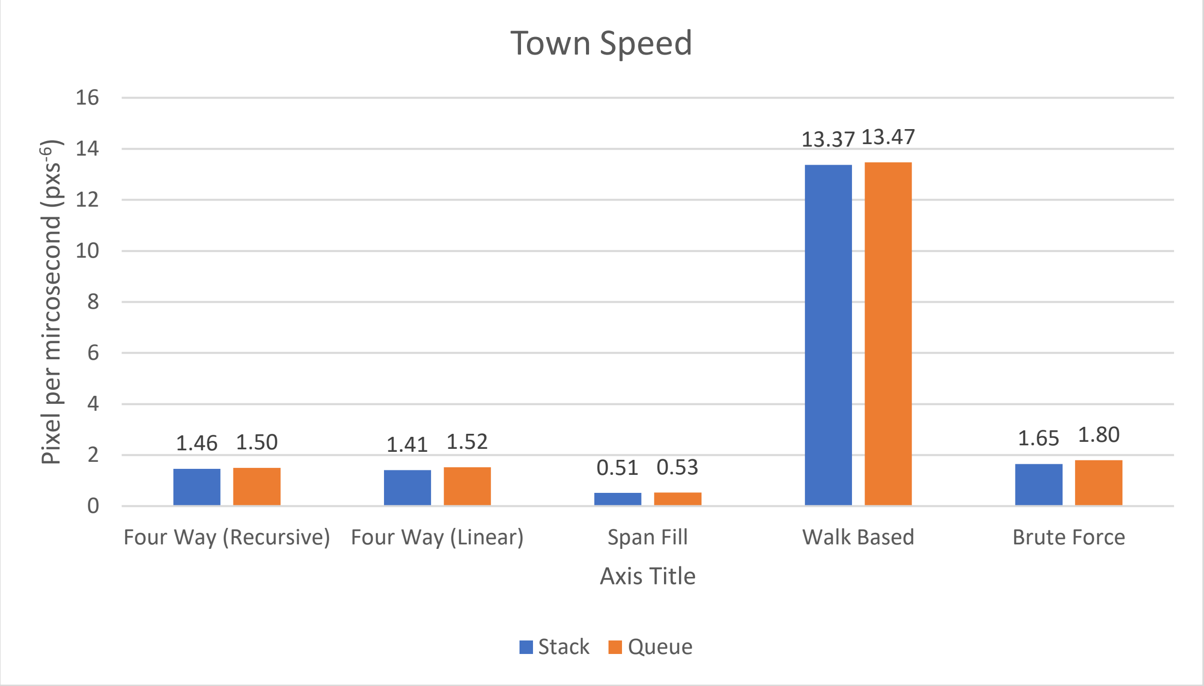


Figure 4.3 Stack vs Queue - General Town Test

The results here a little inconsistent. Supposedly there should be very little difference in speed, however, aside from WBF, stack seems to perform consistently better.

According to the *Microsoft Documentation*:

> *Use Queue<T> if you need to access the information in the same order that it is stored in the collection. Use Stack<T> if you need to access the information in reverse order. [50]*

This makes sense since stacks operates on the "last-in, first-out" principle (LIFO) while queues operate on "first-in, first-out" (FIFO). This could explain the differences in the FWL and SF, as both pop / dequeue variables in a temporary data structure many times. Thus, operating on LIFO could possibly be beneficial, though by how it would be, I do not know.

Since this is not supported by the existing literature, and FWR and BF don't need to pop / dequeue any variables, I think it's much more likely the difference in values are due to either the implementation of the *Stack<T>* and *Queue<T>* in C#, some random error, or that queues need to maintain 2 pointers (one for the first element and one for the last), while stacks only have one (one for the first element) so when reassigning the pointers, it could cost additional time.

Furthermore, the difference is marginal at best with the greatest speed improvement being FWRin Figure 4.3 with an increase of 13%. The other speed improvements are all less than 10%.

From this point on, all the results will be ones using stacks as their data structure. This will allow for the diagrams and figures to be more readable and compact. However, in the accompanying *TestResults.xlsx* there are additional results.

4.2 Algorithm Speed Tests

Testing the speed of the algorithms was quite simple. I used the Stopwatch class to record the time of the algorithm, starting and stopping it on each side of the method call. Once a shape was complete, I would put the time into a *list<long>* using the *stopwatch.ElapsedMilliseconds* property*.* For each of these tests I had them run 1000 times. This was so:

- I could be sure I ironed out any unforeseen delays, errors, or discrepancies.
- to get enough significant figures for my results. Since C#'s Stopwatch class could only give values to the nearest millisecond.

All times are in *microseconds*, *µs,* or s^{-6}.

4.2.1 Circle and Square Test

The Circle test is one where we run the different flood fill algorithm over 16 different circles in increasing sizes. The diameters are – from smallest to largest – 5, 10, 15, 20, 25, 30, 40, 50, 75, 100, 125, 150, 200, 250, 300, 400 pixels. They are all the same colour, RGBA[0,0,0,255].

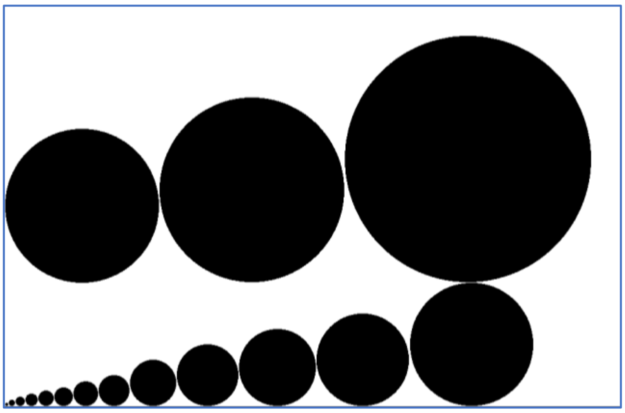
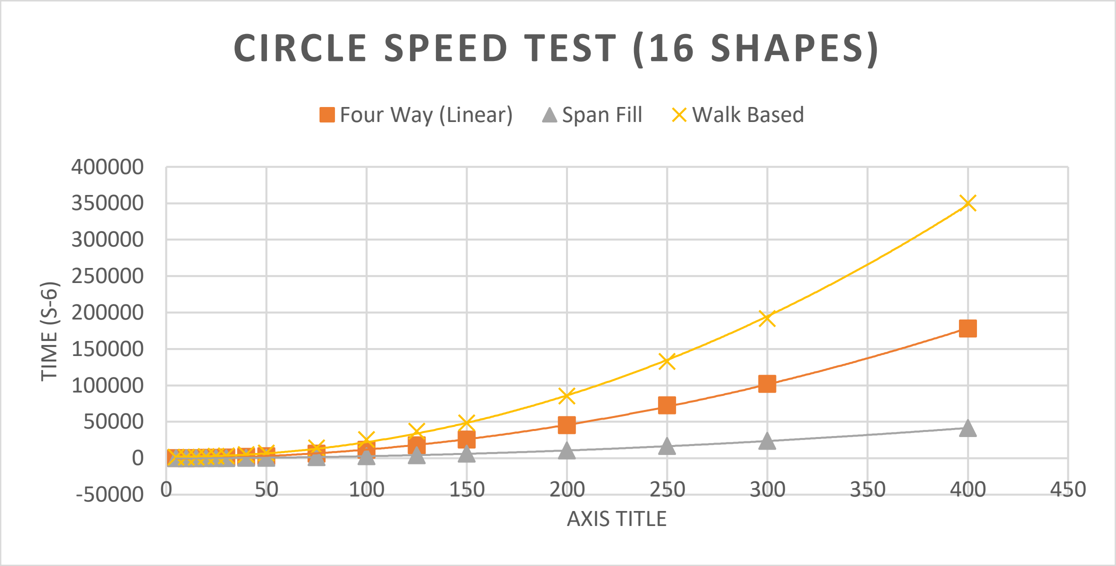
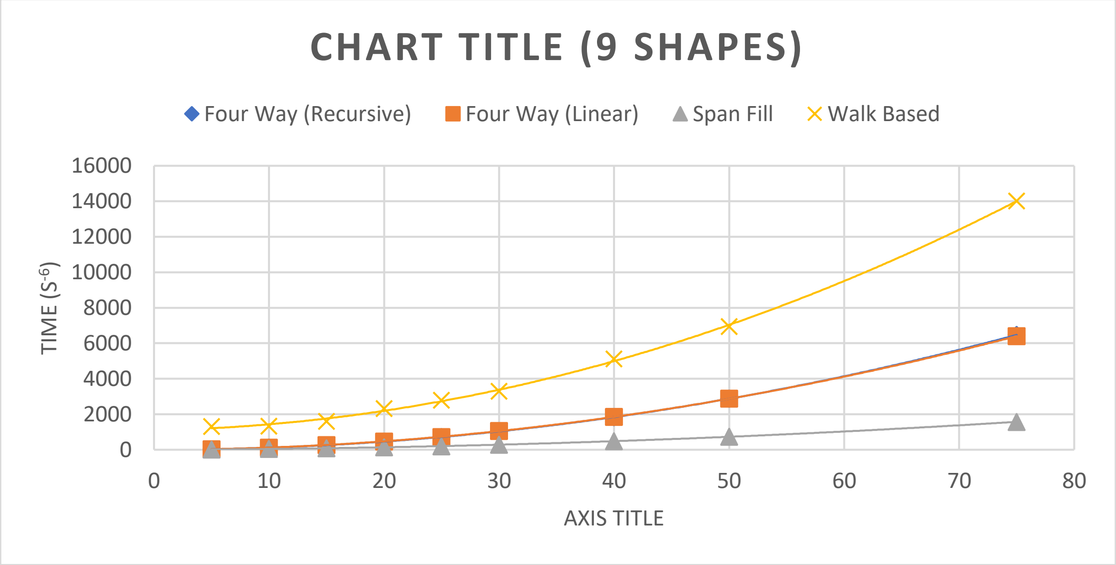


Figure 4.4 Circle Test texture. There are 16 circles in total which vary in size.

Circle Size (Diameter in pixels)	No. Pixels	Four Way (Recursive)	Four Way (Linear)	Span Fill	Walk Based
5	21	34	34	16	1319

10	80	121	130	41	1332
15	177	252	260	87	1594
20	316	455	471	136	2329
25	489	709	718	197	2785
30	716	1044	1057	288	3295
40	1268	1836	1848	479	5127
50	1976	2878	2886	735	6944
75	4418	6468	6399	1569	14017
100	7860	STACK OVERFLOW	11395	2760	25720
125	12277	STACK OVERFLOW	17879	4226	37000
150	17693	STACK OVERFLOW	25560	6085	48581
200	31436	STACK OVERFLOW	45344	10716	85172
250	49100	STACK OVERFLOW	72511	16624	132857
300	70711	STACK OVERFLOW	102041	23663	191729
400	125691	STACK OVERFLOW	177941	41602	350475





The trendline equations for the graph above:

Algorithm	Recursive	Linear	Span Fill	Walk Based Fill
Trendline Equation	$y = 1.151x^2 - 0.0661x$	$y = 1.1209x^2 + 1.3408x$	$y = 0.2552x^2 + 1.8083x$	$y = 1.4678x^2 + 73.739x$

The Square test is the exact same as the Circle test, but this time with squares. The diameters of the circles are the same as the lengths of each respective square.

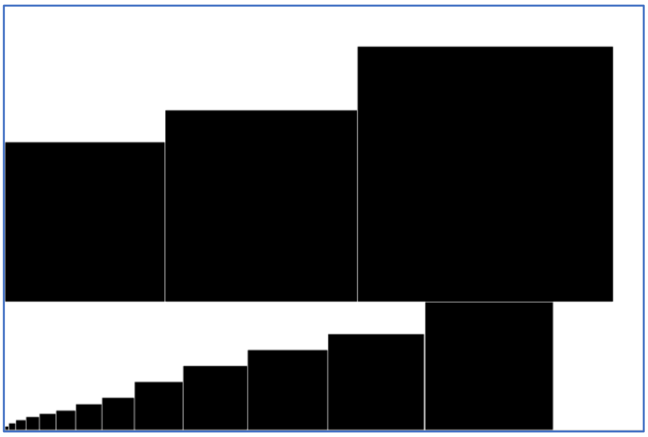
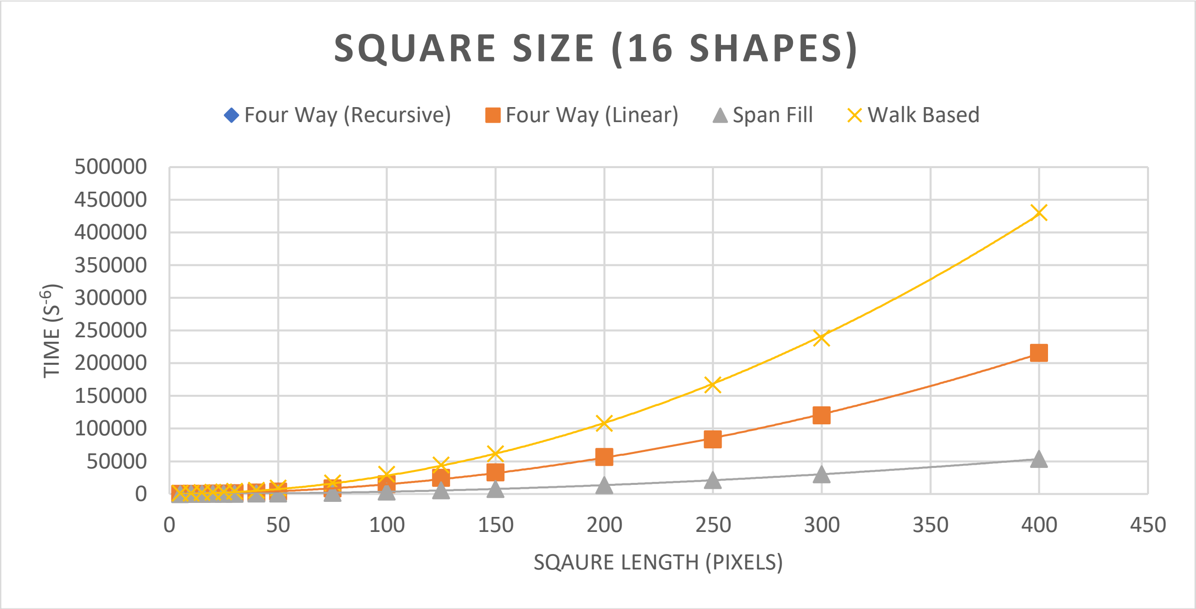
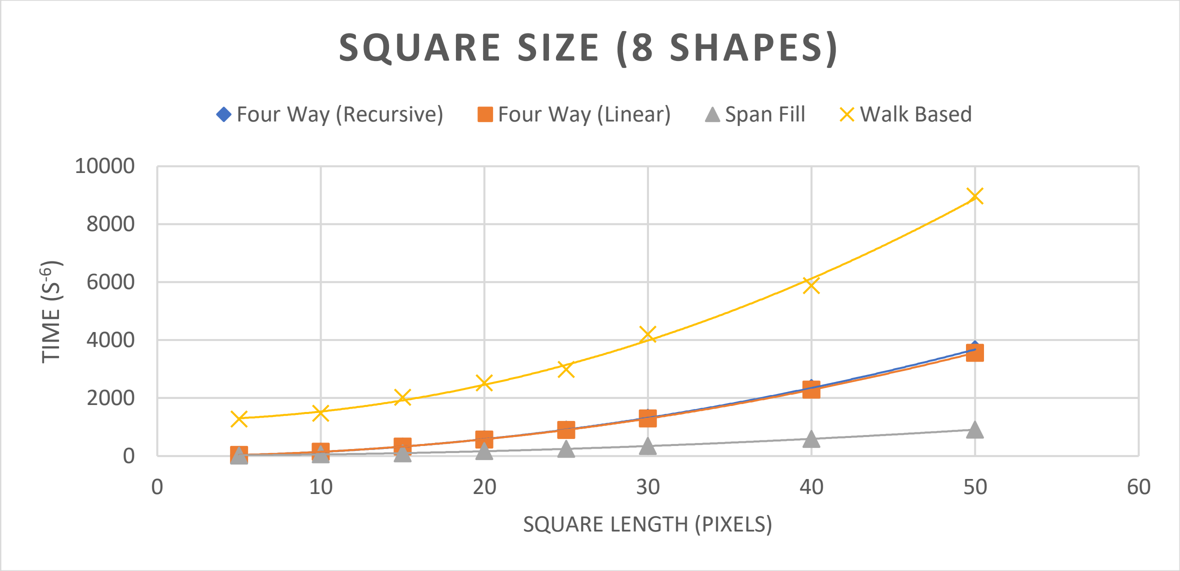


Figure 4.5 Square Test texture. There are 16 squares in total which vary in size.

Square Size (length)	No. Pixels	Four Way (Recursive)	Four Way (Linear)	Span Fill	Walk Based
5	25	38	36	22	1275
10	100	149	146	57	1478
15	225	325	331	99	2030
20	400	582	574	168	2527
25	625	913	896	249	2986
30	900	1317	1300	345	4202
40	1600	2352	2289	593	5884
50	2500	3680	3563	911	8977
75	5625	STACK OVERFLOW	8792	1983	16789
100	10000	STACK OVERFLOW	15152	3498	30270
125	15625	STACK OVERFLOW	24574	5422	44203
150	22500	STACK OVERFLOW	33267	7729	61639
200	40000	STACK OVERFLOW	56608	13559	108003
250	62500	STACK OVERFLOW	83682	21126	166962
300	90000	STACK OVERFLOW	120171	30163	238452
400	160000	STACK OVERFLOW	215834	53488	430454





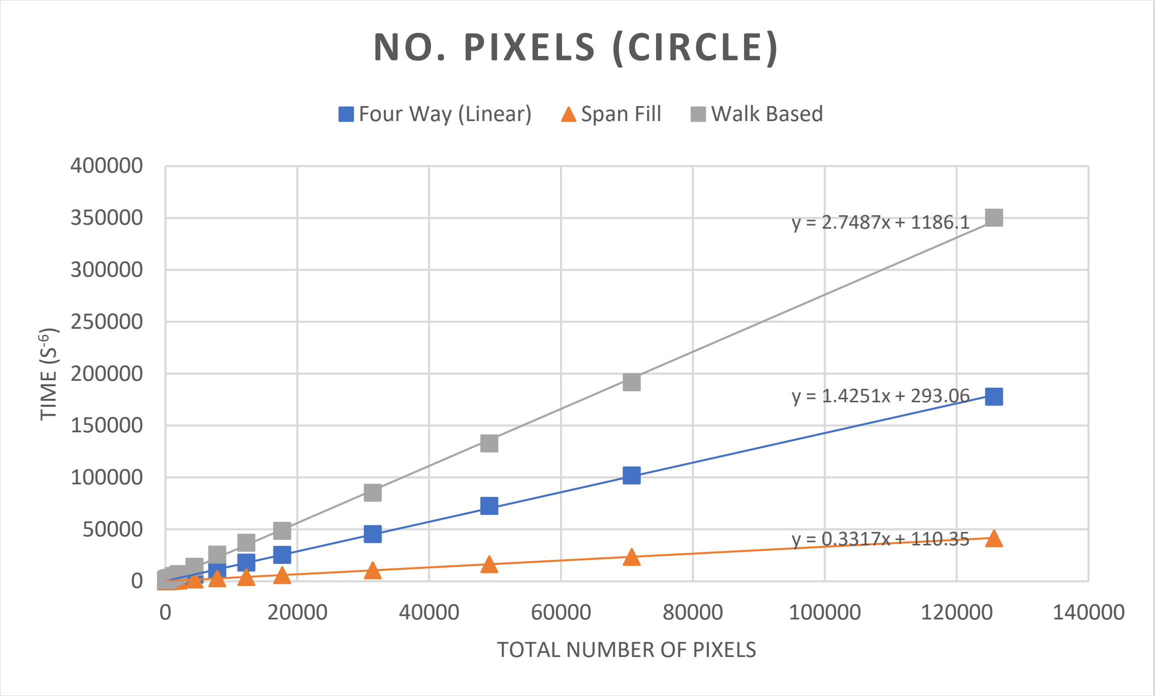
Trendline equations for the graph above:

	Recursive	Linear	Span Fill	Walk Based Fill
Trendline Equation	$y = 1.4811x^2 - 0.463x$	$y = 1.2925x^2 + 19.422x$	$y = 0.3291x^2 + 2.0237x$	$y = 2.6319x^2 + 17x$

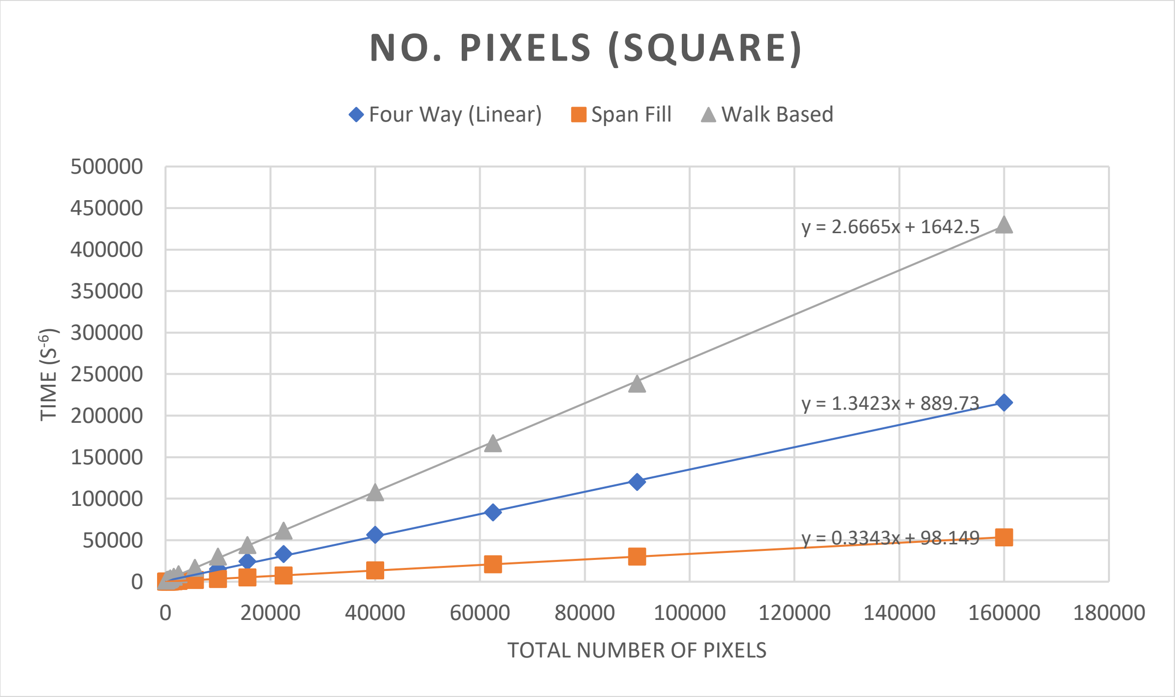
Analysis:

As the size of the diameter or length of the shape increases, so does the time it takes to flood fill it. These relationship between these two variables noticeably follows a polynomial trendline for every algorithm.

If we were to plot out the time taken for the flood fill algorithm to execute to the number of pixels, we would get something like this:



As you can see, they are directly proportional to on another. And this doesn't change with shape either:

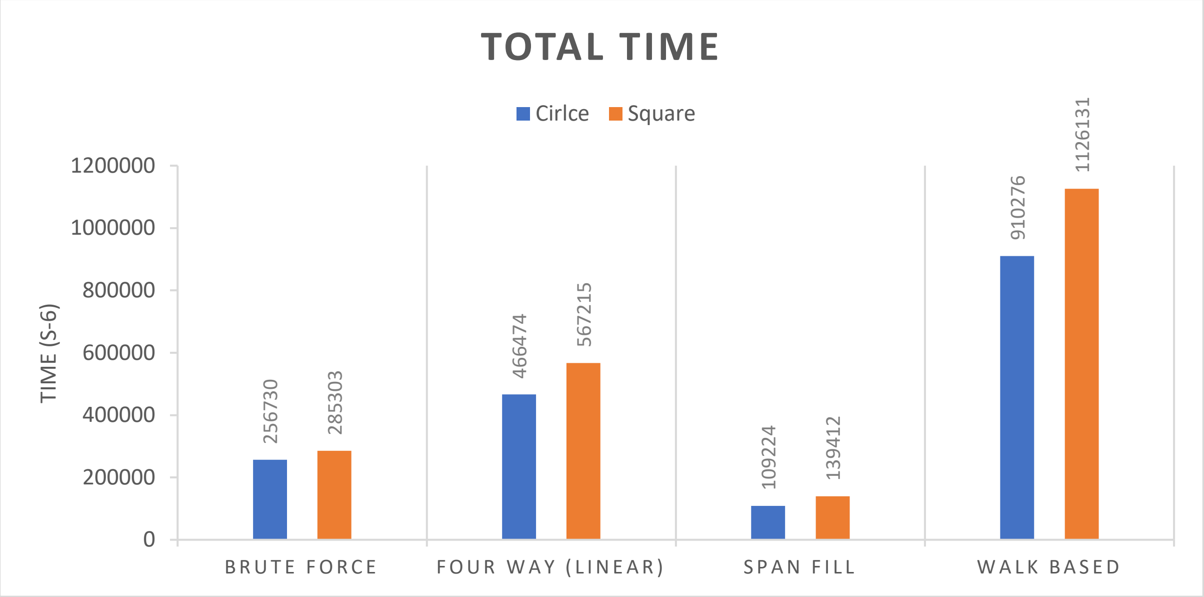


Furthermore, we can clearly see that SF is by far and away the best algorithm in terms of speed. With it being on average 4.16x faster (average of both circle and square) then FWL and 8.13x faster then WBF.

There was not a significant difference in results between FWR and FWL, which was to be expect because both run on the same logic. However, it became clear that its quite common for a *StackOver-*

flow error to occur once you get past ~4400 pixels. Thus, it would seem unreasonable to use this algorithm in any circumstance as of yet, because the linear implementation is just as fast, much more reliable, and scalable.

When it comes to attempting a brute force approach it was impossible to gain results for individual shapes since the BF I wrote does not have the fidelity to scan these as individual shapes. To compare BF to the rest, we can chart their performance by totalling all their times together:



Interestingly, as alluded to earlier in chapter 3.4, BF does very well. It was both 1.9x faster than FWL and 3.75x faster then WBF. This is mostly likely because BF only has to check each pixel once, while the other algorithms have to check every pixel more than once (see chapter 3.3) and checking each coloured pixel (black pixels in this case) up to 4 times each.

However, this is the price you pay for having the additional fidelity to segment different shapes. This functionally, while not useful on this project could be in the future if this piece of software was expanded upon as it offers more control to the developers (e.g. changing a certain shape's buildings without changing the buildings on the rest of the shapes of that colour).

However, SFis still faster than BF despite the automatic initial seed finding, with SF being 2.2x faster than BF. This is most likely due to span filling out a line at a time thus only revisiting most pixels around 3 times [36] and since these shapes are very simple, that would not happen many times.

4.2.2 Test Town

In this test I used the generic town layout created early (Figure 3.9). This test was mainly to get results for the most common use case of this project, which was the settlement creation. This test also had the benefit of testing out a total of 6 different colours. On top of that, there were many different types of polygon within the town, from triangles to pentagons.

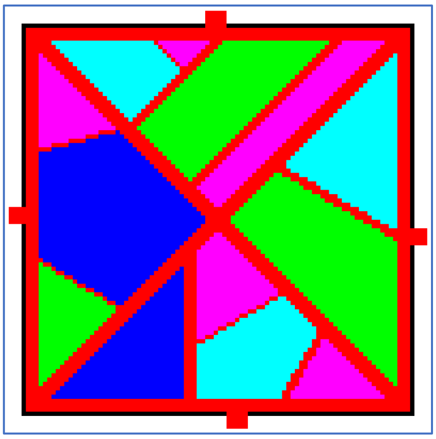
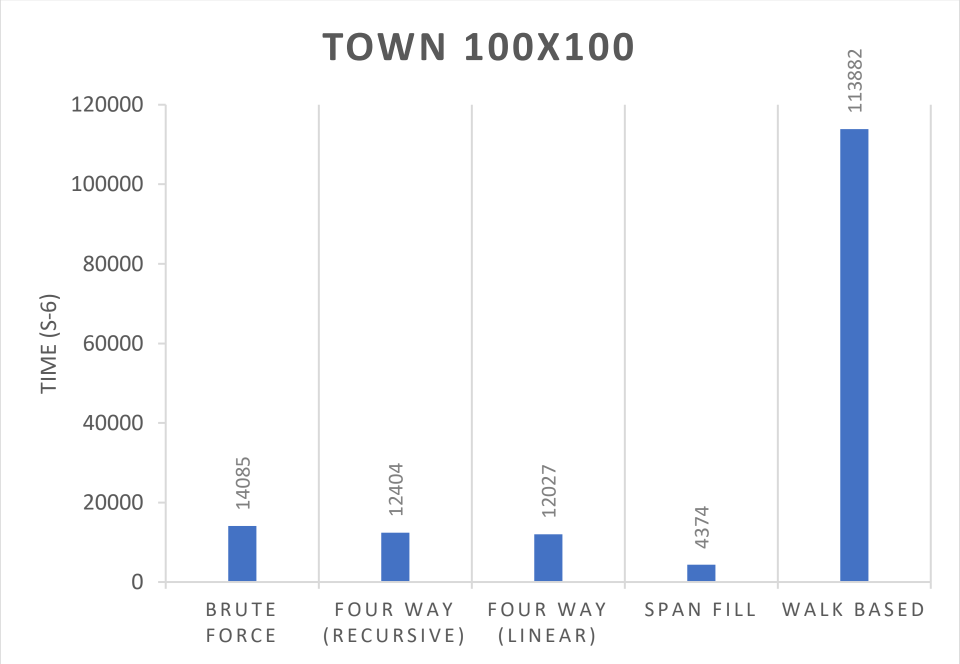


Figure 4.6 Same generic town shown in figure 3.9

	No. Pixels	Brute Force	Four Way (Recursive)	Four Way (Linear)	Span Fill	Walk Based
100x100 Town	8518	14085	12404	12027	4374	113882



As expected, SF performed the best via - 2.75x the speed of FWL. However, BF was noticeably slower - 3.22x slower then SF and 1.17x slower then FWL. This could because of the additional 6 colours. Could be because BF must process the same texture once for every colour (6 in this case). Furthermore, because of the size of the shapes, relative to the canvas, were quite large it meant FWR, FWL and SF were more efficient and didn't have to re-check pixels as many times – this is better shown in chapter 4.2.3.

Finally, WBF performed significantly worse when viewing pixels per microsecond (px/s^{-6}) (see chapter 4.2.4) being 4.9x slower than its px/s^{-6} for the *square* test. This is most likely due to the algorithm getting caught in some loop, walking around for a long time without painting. This is most likely because the shapes were less orthodox and regular so it is possible one of them cause this algorithm quite a few problems.

Furthermore, since some of the roads (red lines) are one pixel thick and do not connect via four-connect region but instead eight-connected region could cause a slow-down. This is because they are processed as different shapes. However, this would also affect both four-way algorithms and SF to, because more individual shapes seem to affect the performance of these algorithms much more then WBF - see chapter 4.2.3.

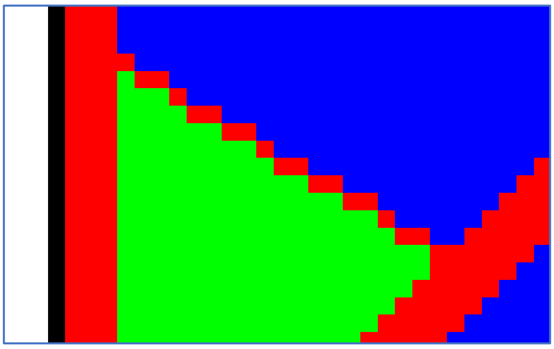


Figure 4.7 Example of road where many of the pixels are not part of the same four-connected region.

While this test may not be as precise at outlining a specific situation, it is a very good example of your average use case. Most developers will be using textures like this one.

4.2.3 Checkers Test

The 100x100 was to test how each algorithm would handle many small individual shapes. This should be stressful on all algorithms apart from BF as the checkers nullify the strengths and efficiency of the typical flood fill algorithms and make their logic redundant as each shape is only 1 pixel big.

The two textures used here only differ in the amount of space between each shape. In the 25% checker texture, one pixel/shape is placed every 4 pixels while in the 50% checker texture one pixel/shape is place every other pixel.

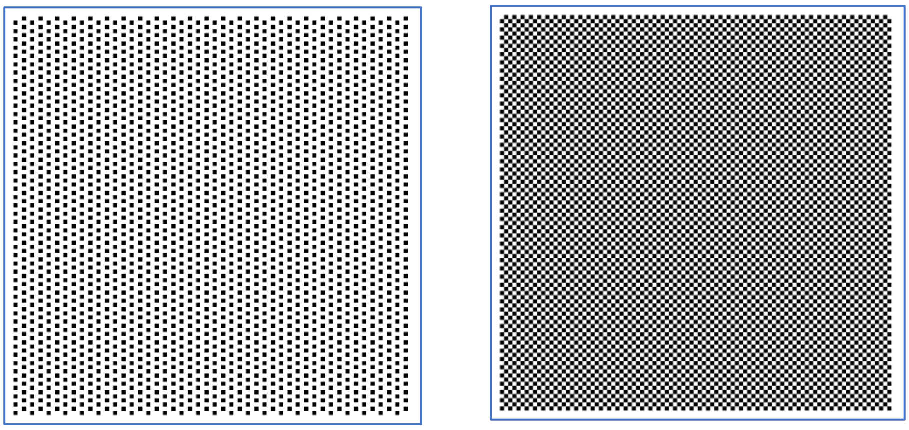
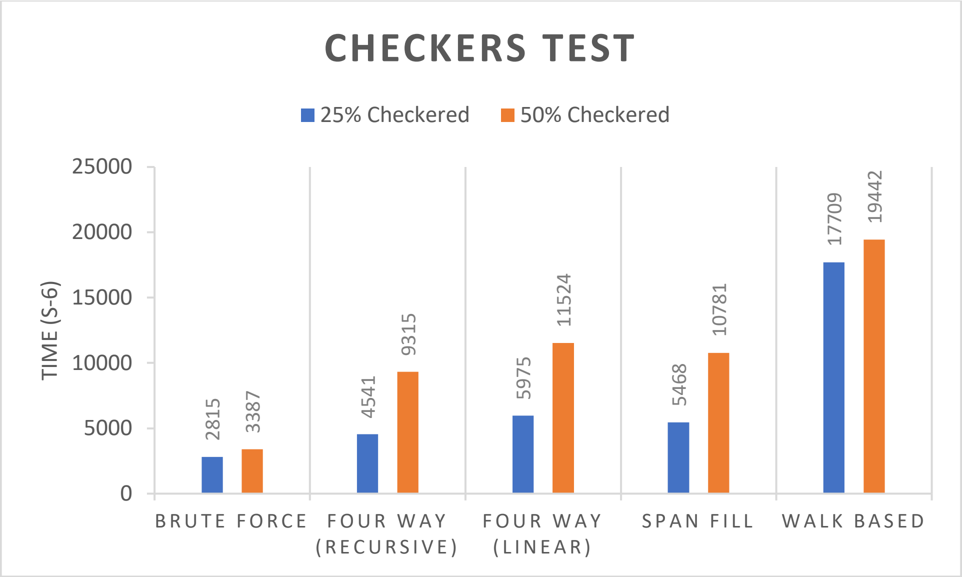


Figure 4.8 Left: 25% checker texture. Right: 50% checker texture.

	No. Pixels	Brute Force	Four Way (Recursive)	Four Way (Linear)	Span Fill	Walk Based
25% Checkers	2304	2815	4541	5975	5468	17709
50% Checkers	4560	3387	9315	11524	10781	19442



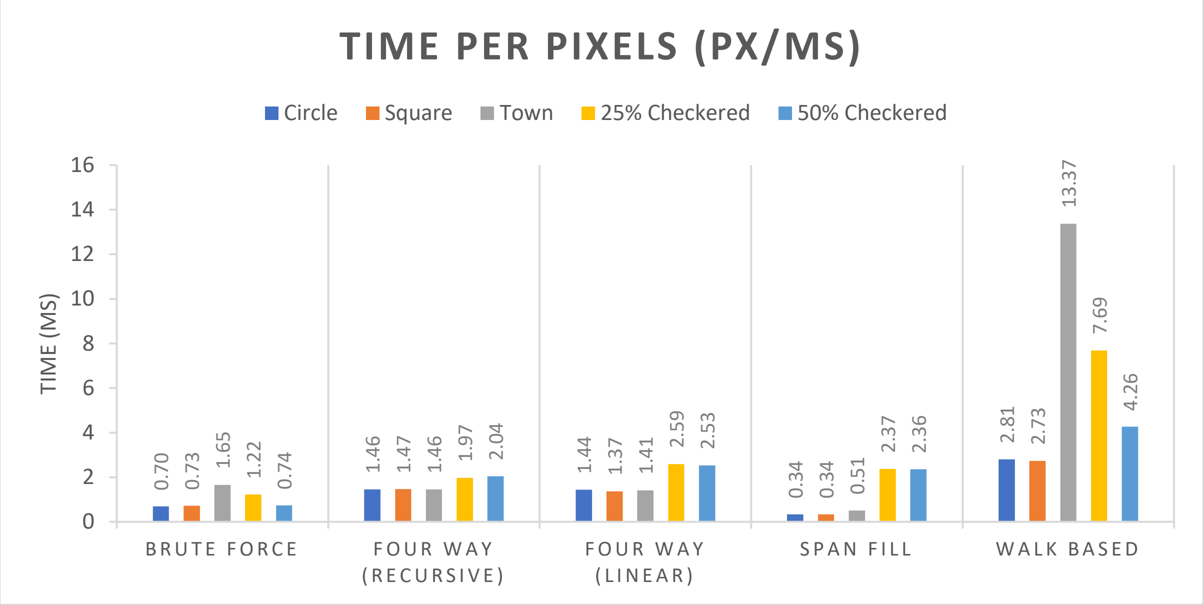
These results are mostly uprising with BF perform the best on average:

- 2.18x faster than four way recursive.
- 2.76x faster than four way linear.
- 2.57x then span fill.
- 6.02x faster then walk based.

This is most likely due to the reason stated early. However, interesting, the FWR performed marginally better than FWL (on average 1.28x faster). As for the reason, I am not sure why exactly. Perhaps it is because the linear implementation must create a temporary data structure both popping and pushing points on to it. Since the recursively called function instantly returns if the adjacent pixels are not part of the shape, it does not instantiate or use additional unnecessary data structures.

WBF is another point of intrigue as its 50% and 25% are very close in speed, particularly when compared to the other algorithms with 50% only being 9% slower than 25% (compare that with the next closest being BF where the different 50% is 55% slower than 25%). This perhaps suggests that smaller canvases give WBF performance issues and the amount of shapes is much less significant factor in its speed.

4.2.4 Algorithm Speed: Summary



This is probably the most interesting graph as helps give us and overall view of which algorithm is best suited for each set of circumstances.

As it should be quite apparent, the two algorithms you should look to be using (when it comes to speed) are BF and SF.

SF should be used in most circumstances, as it is the fastest when it comes to scaling the size of shapes, as seen with the circle and square tests, and when it comes to a texture with multiple colours, as seen with the town test.

However, when SF confronts textures with a high number of small shapes, it may be wise to use BF instead as its logic's strengths remain useful in these circumstances while SF's logic become an unnecessary burden.

FWR may be best in a circumstance where the texture has both a high number of colours but also a high number of small shapes. However, unless the texture remains small, you may encounter a *stackoverflow* error.

4.3 Memory Use Tests

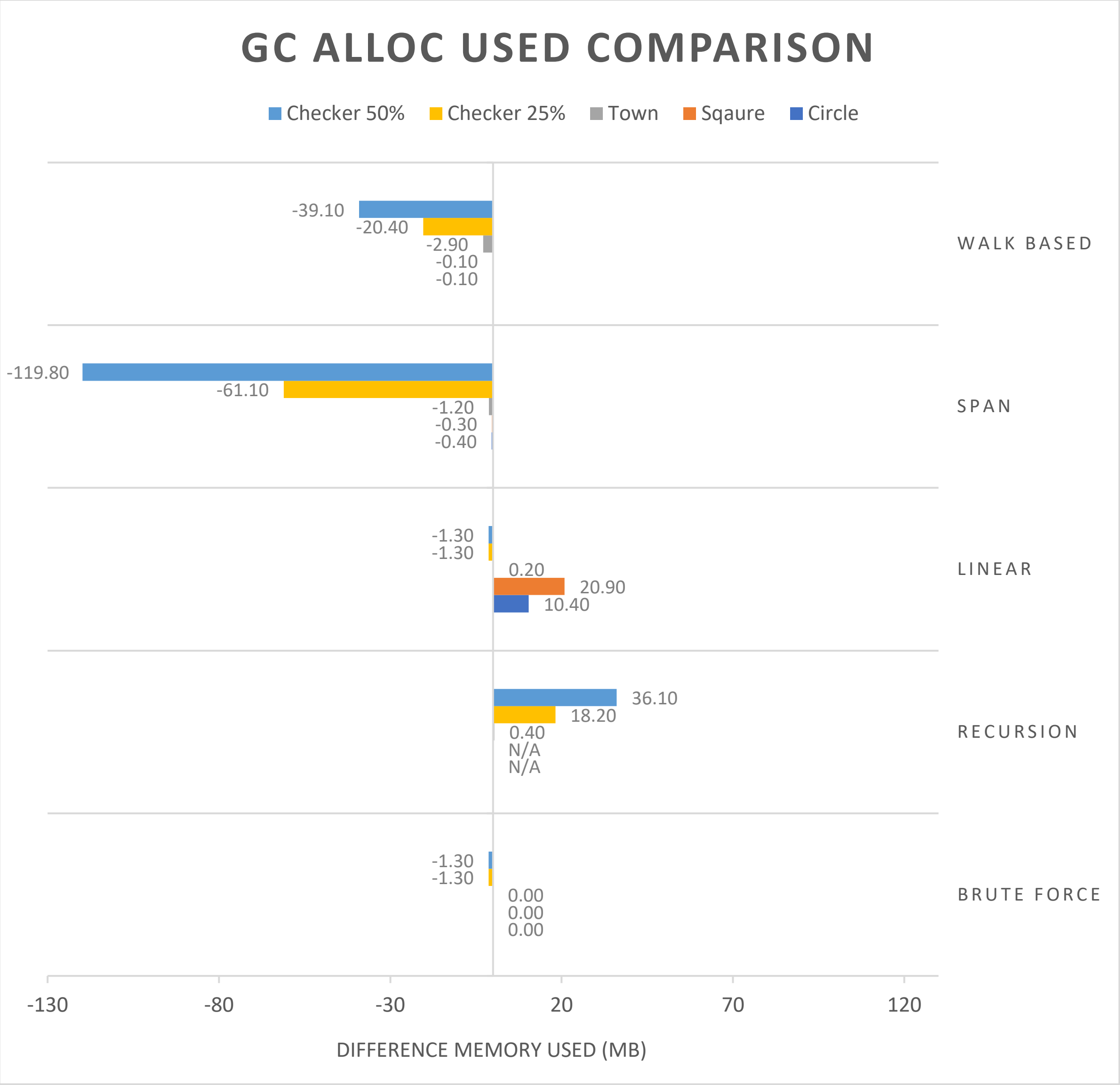
To gain a more compressive look at the performance of each algorithm, I also examined the amount of memory used while each algorithm was running. This could further help us understand when to use one algorithm over another depending on the hardware, or what could happen to a computer when this was performed on a much larger scale. Furthermore, specific algorithms (i.e. WBF) main strength is that it is a fixed-memory algorithm.

However, due writing this in C# and Unity, the measuring of peak memory use would be difficult because C# has a garbage collector. Thus, the tests performed in this section could use some improvements in future works. This is because Unity's profiler was only able to give me the total GC Alloc during the run time of the algorithm. The garbage collector (GC) only deals with things that are no longer reference, thus the size and fluctuations of temporary data structures (such as those in FW and SF) are not shown.

Each of these tests were conducted 5 times and the mean average of the both the GC Alloc memory and time passed were used.

4.3.1 GC Alloc Used Comparison

We can still pull some interesting data from this. Mainly in how it displays some differences in using stacks or queues as data structures:



These calculations were done be subtracting the memory use of the queues from those of the stack. So, a + value shows that using a stack data structure used more memory and a - value shows a queue used more memory. As you can see both span fill and walk base favour using a stack as op-posed to a queue. Sometimes it is marginal at best as seen with the circle and square test, other times is it very noticeable like the checker tests.

On the other hand, Linear only slightly prefers stacks when it comes to the checker tests but strongly prefers queues during the circle and square test.

The reason for these discrepancies is not entirely obvious. Since these results were averages and consistent with each other (no anomalies) is it hard to suggest it is to do with an error with the testing method. Perhaps, these tests could be done on another set of hardware to confirm. Another possibility could be to do with the memory access patterns of each data structure.

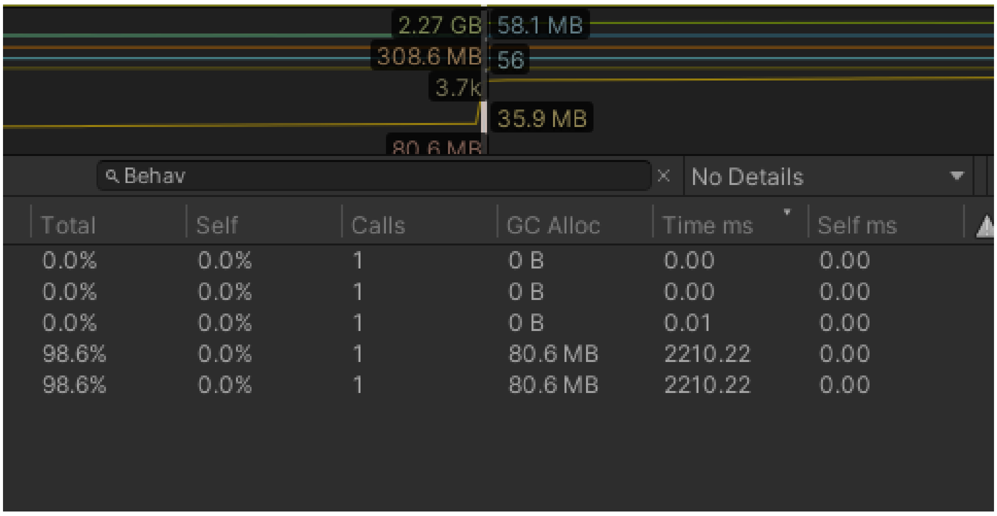
For example, in the case of SF, using a stack data structure was always less memory intensive. This could be because the initial seed is always the most bottom-left pixel. So, when the add new positions to the temporary stack the seed for the span below will always get popped first. This is unlike the queue, where the seed span above will always get dequeued first.

However, this type of explanation falters for algorithms such as WBF. This is because they do not have any removal of elements from its data structures, and thus the differences between Queue and Stack, theoretically, is nullified.

Once again, these inconsistencies could be due to the how the GC works. Since it is an automated process, you cannot control when it happens:

> *The garbage collector's optimizing engine determines the best time to perform a collection based on the allocations being made. [51]*

This method of data collection can be made slightly more accurate as, according to the Unity Profiler, there was only one spike GC Alloc when these methods were called:

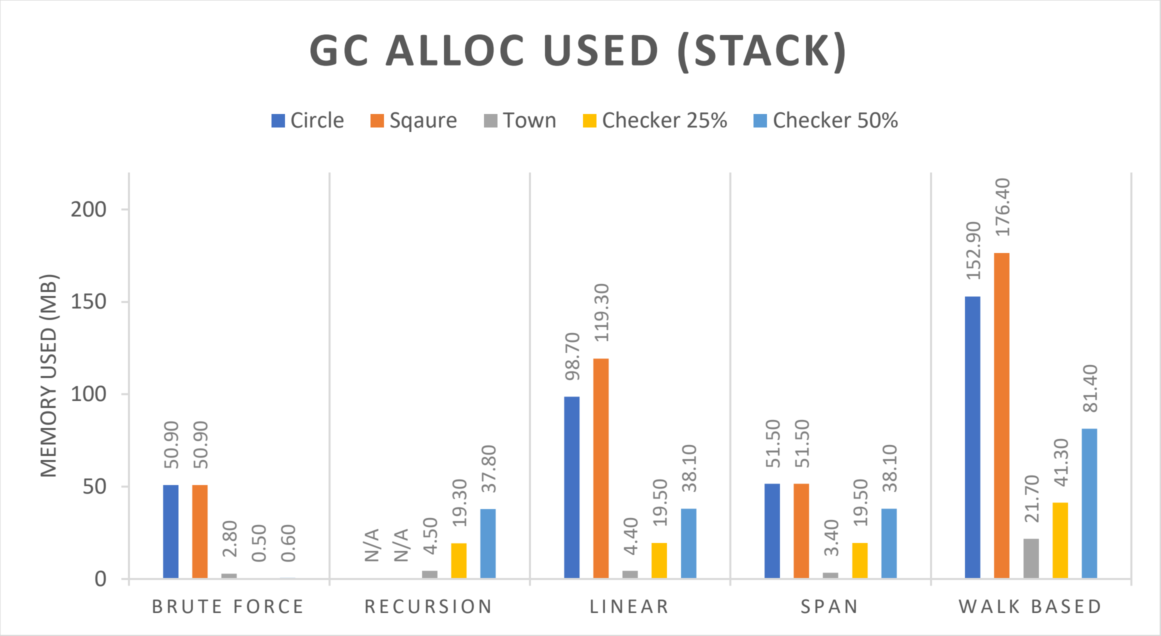


Total	Self	Calls	GC Alloc	Time ms	Self ms
0.0%	0.0%	1	0 B	0.00	0.00
0.0%	0.0%	1	0 B	0.00	0.00
0.0%	0.0%	1	0 B	0.01	0.00
98.6%	0.0%	1	80.6 MB	2210.22	0.00
98.6%	0.0%	1	80.6 MB	2210.22	0.00

Figure 4.9 Unity Profiler displaying GC Alloc

4.3.2 GC Alloc Used

Now that we have compared the data structures to each other on the total memory use, now let's focus more on the individual algorithms:



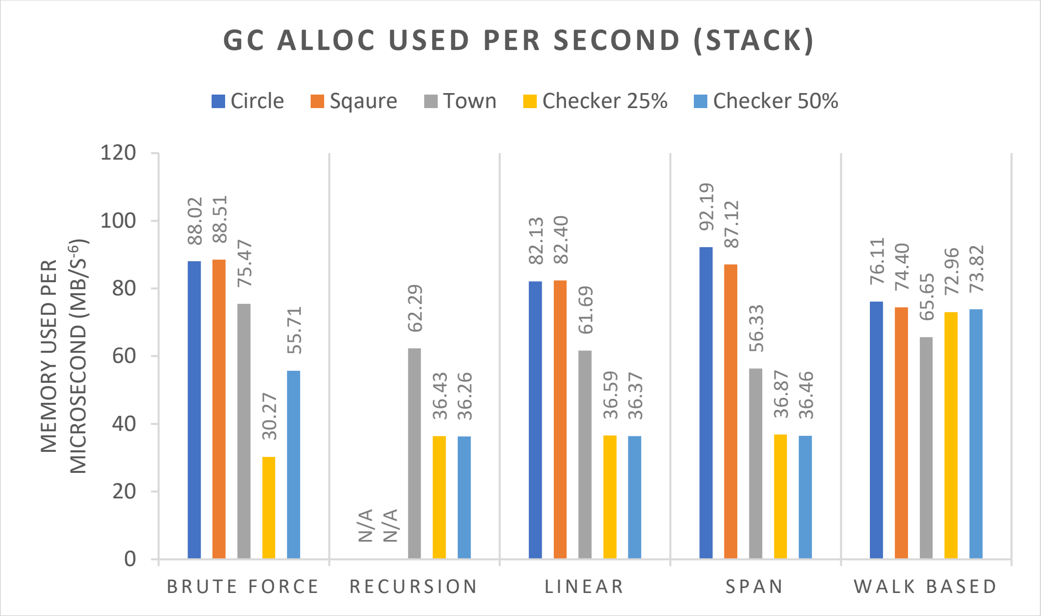
Unsurprisingly, BF uses the least memory overall. There are not as many variables instantiated within the algorithm, as is just iterates through every single pixel. This is most evident in the Checkers 50% texture where BF is 63x, 63.5x, 63.5x, 135.67x more efficient then FWR, FWL, SF and WBF respectively.

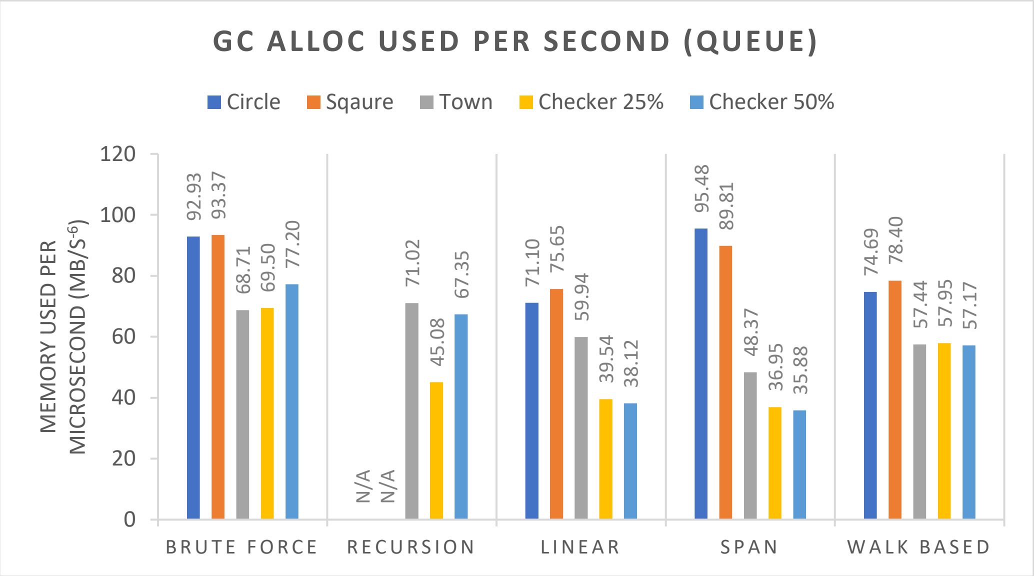
WBF, when looking at this graph at face value, is the worst in every category. But this is where the measuring of GC Alloc is unhelpful. Since we cannot control when the GC will happen, and we can't see the peak memory use of the other algorithm, many of the short comings of this fixed fill algorithm are possibly exaggerated. This is because, the temporary data structure within FWL and SF deallocates it's old elements when it pops/dequeues them, and the garbage collector may possibly clean up earlier on while the algorithm is running. For WBF, there are many additional variables to keep track up, such as the current position of the cursor, marks, mark's directions and case count.

4.3.3 GC Alloc Used per Microsecond

One other perspective the data gathered offers, could be to calculate how many megabits per second (µs) each algorithm had. Once again, there were some inconsistencies between data structures, but the board picture can still give us something to analysis.

This method of showing the data allows us to get a more accurate view of the average memory demand of each algorithm since they do not have a consistent run time between them.





Overall, the results were quite comparable. SF and FWL (and sometime FWR depending on the data structure) being the best for memory for the smaller textures (Town and Checkers). While WBF and FWL best one the larger textures (Circle and Square). This shows a much more favourable picture towards WBF in particular, because it's strengths as an algorithm, that being its fixed memory solution, have been overshadowed by its lower performance in speed.

5 Conclusion

In conclusion, I think the project was a good foundation to a potential product in the future to help with game development. With more focus on allowing for more sensible building placement, allowing for it to use height maps so that settlements can be generate upon hills or within valleys, etc. it could become a useful tool within the workflow of many game developers.

I completed my initial aim because I created a projected that used flood fill algorithms to generate settlements from a diagram. I learnt a lot about Unity 3D, such as learning how to use the profiler, and C#. I learnt about 4 different types of flood fill algorithm, learning how to implement them, their history, and their pseudocode. Furthermore, I was able to successfully analyse their speed based perform. However, I will admit, the testing of memory performance was not entirely successful.

I was happy with the breadth of algorithms implemented and tested with many showing different strengths in different situations, including a brute force approach. As expected, Span Fill was much faster than the other algorithms, particularly the Four Way ones. The implementation of these algorithms could also be improved as the method by which you find the initial seed could possibly be optimised, or there is a better solution to that problem.

Unfortunately, the data gathering method for memory used wasn't quite comprehensive enough, possibly rewriting this in C++ - using an engine like Godot or Unreal Engine – would allow me to get better memory control, to allow me to experiment further.

Overall, I am very happy with this project.

6 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] A. R. Smith, “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] L. Vandevenne, “Lode's Computer Graphics Tutorial - Flood Fill,” 2004 . [Online]. Available: https://lodev.org/cgtutor/floodfill.html.

[40] A. Milazzo, “A More Efficient Flood Fill,” 10 May 2015. [Online]. Available: http://www.adammil.net/blog/v126_A_More_Efficient_Flood_Fill.html.

[41] D. Henrich, “Space-efficient region filling in raster graphics,” *The Visual Computer,* vol. 10, no. 4, pp. 205-215, 1994.

[42] M. Brown, *The Best Games from GMTK Game Jam 2020,* Game Maker's Toolkit, 2020.

[43] u/negavolt, “Engine Usage Breakdown for GMTK Jam 2020 - Godot at 12.2%,” 24 July 2020. [Online]. Available: https://www.reddit.com/r/godot/comments/hx3xd5/engine_usage_breakdown_for_gmtk_jam_2020_godot_at/.

[44] K. P. Fishkin and B. A. Barsky, “203 - AN ANALYSIS AND ALGORITHM FOR FILLING PROPAGATION,” Berkeley, 2015.

[45] Unity, “Texture2D.GetPixels32,” Unity, [Online]. Available: https://docs.unity3d.com/ScriptReference/Texture2D.GetPixels32.html.

[46] 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.

[47] 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/.

[48] Twaling, “Flood Fill#Pseudocode,” Wikipedia, 4 February 2012. [Online]. Available: https://en.wikipedia.org/wiki/Flood_fill#Pseudocode.

[49] Synty Studios, *POLYGON Knights - Low Poly 3D Art by Synty,* Unity Asset Store, 2017.

[50] “Queue<T> Class,” Microsoft, [Online]. Available: https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.queue-1?view=net-5.0.

[51] Microsoft, “Fundamentals of garbage collection,” Microsoft Docs, [Online]. Available: https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals.

[53] Lee, “NYC Buildings over Time,” 5 May 2010. [Online]. Available: http://tier1dc.blogspot.com/2010/05/nyc-buildings-over-time.html.

7 APPENDIX

7.1 TABLE OF FIGURES