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 1.1

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).

Find papers on 4 different things:

* Procedural Generation (Broad)
* City Generation
* Flood Fill diagrams
* Urban City Diagrams (sidenote)
* Types of Settlements (sidenote)

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 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 necessarily seem 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 [19]

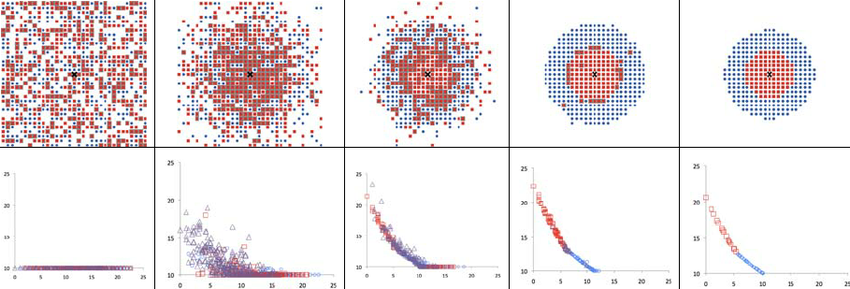


Figure 2.2 Evolution of the shape of the city (first line) and of the price of land as a function of the distance to the center (second line) during a simulation. [20]

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 like native American settlements.

This led my down the path of analysing different city types in general [21] 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:

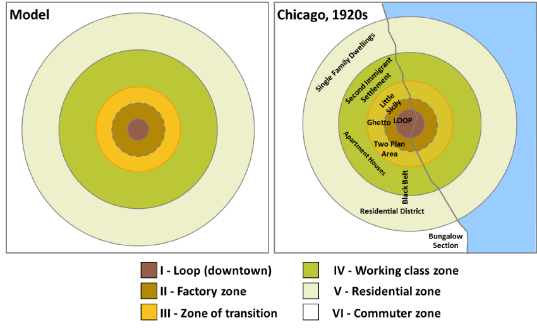
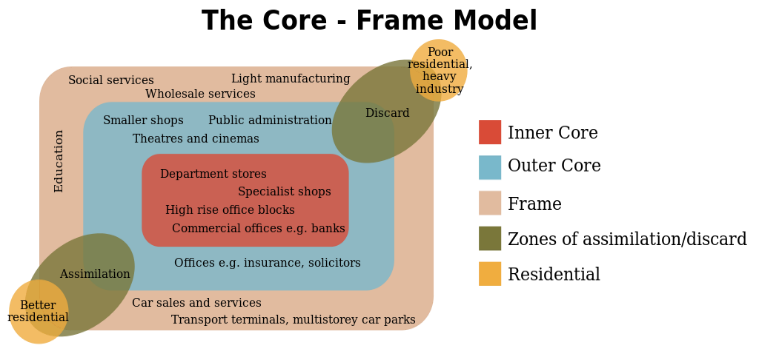
* Concentric Zone / Burgess Model [22] (Figure 2.3) [23]
* Core Frame Model [24] (Figure 2.4) [25]
* Sector/Hoyt Model [26] (Figure 2.5) [27]
* Multiple Nuclei Model [28] (Figure 2.6) [29]

Figure 2.4 Core Frame Model

Figure 2.3 Burgess Model

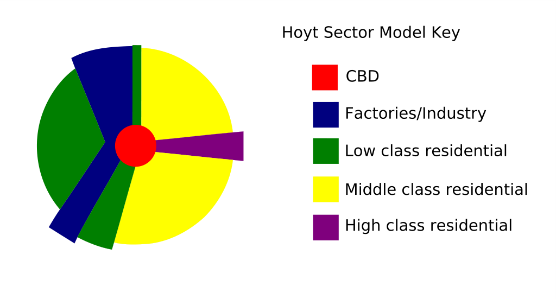
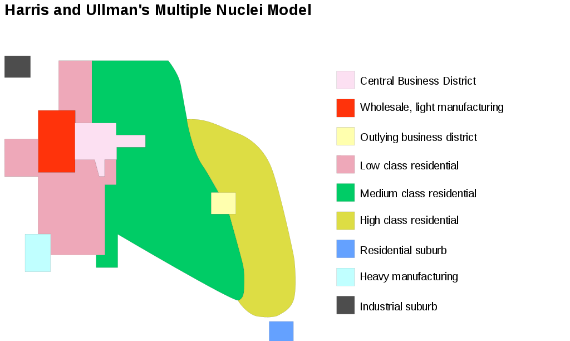
These models were inspirational in design only and in the end a less abstract approach was taken. (see Figure 1.1) This led to common structures such as roads and walls to be manually imputed. Furthermore, the labelling of the sectors and what buildings spawn withing them is up to the digression of the user. If the user wanted, they could use a procedurally generated urban LUM for a more realistic modern day city layout. [30] [31]

Figure 2.5 Multiple Nuclei Model

Figure 2.4 Sector Model

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.

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. There a many different types of flood fill algorithms and different methods of implementing them. So, I took some of the most well known 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* [32] 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.

The Pseudo Code has been adapted for this project. But the principles remain the same.

### Recursive Implementation



### Linear Implementation



## 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. [33]

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.

[34]

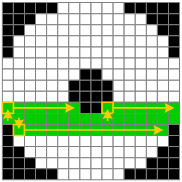
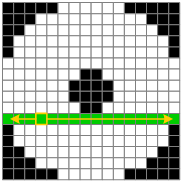


Figure 2.1 First, filling in horizontally from seed (left picture), then find a new seed in the lines above and below (right picture).



## 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. [35]

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 #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.

**Case #5:** Move right along the x axis until you find a boarder. Don’t paint until boarder is found.

This was algorithm was first published by Dominik Henrich in 1994. [35]

# 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

To start off my implementation, I had to select how I wanted to approach the project: to either attempt to implement it at a lower level, with technologies such as OpenGL, or take a more high level approach, with technologies such as the Unity Engine. Initially, implementation was attempted in OpenGL with C++ but my lack of experience with such tools the aim of the project was eventually found to not fit well. The project eventually moved towards the idea of using pre-existing meshes to create a settlement, which were tools that the Unity provided already. Furthermore, I had much more experience with Unity as a technology then with OpenGL.

## BuildingGenerator.cs

Figure 3.1 UML diagram of the code base

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:

* Set the height and width of the diagram the settlement will be based on.
* The settlement size Magnifier.
* Select which data structure you would like (Stack or Queue)
* Select which algorithm you would like the settlement drawn by (Four Way, Span Fill, Walk Based Fill, Perimeter Fill)
* Array of Buildings/Meshes you wish to spawn in certain sectors.
* Array of Colours that correspond to each sector.

**EXAMPLE HERE Image here**

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 were 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()*. **<INSERT CODEBITE>**

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. [36]

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.

**<INSERT CODEBITE>**

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 chosen flood fill algorithm will be called.

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. [36]

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[currentMipPosition]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.

## FourWay.cs

### Stack Storage

### Queue Storage

## SpanFill.cs

## WalkBasedFilling.cs

## PerimeterCheck.cs

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

# Conclusion (10% ~800 – 1200 Words)

# 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. Hendrikx, 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] | Lee, “NYC Buildings over Time,” 5 May 2010. [Online]. Available: http://tier1dc.blogspot.com/2010/05/nyc-buildings-over-time.html. |
| [20] | 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. |
| [21] | R. Easton and P. Nabokov, Native American Architecture, Oxford University Press, 1989, pp. 76 - 80. |
| [22] | E. W. Burgess and R. E. Park, in *The City*, Chicago, University of Chicago Press, 1925, pp. 71-78. |
| [23] | 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]. |
| [24] | E. M. Horwood and R. R. Boyce, Measurement of Central Business District Change and Urban Highway Impact, Seattle: University of Washington, 1959. |
| [25] | 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]. |
| [26] | H. Hoyt, The structure and growth of residential neighborhoods in American cities, US Government Printing Office, 1939. |
| [27] | 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]. |
| [28] | 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. |
| [29] | 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]. |
| [30] | 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. |
| [31] | T. Lechner, P. Ren, B. Watson, C. Brozefski and U. Wilenski, “Procedural modeling of urban land use,” in *Lechner 2006 Procedural*, 2006. |
| [32] | 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. |
| [33] | S. A. Ray, “Tint fill,” *SIGGRAPH '79: Proceedings of the 6th annual conference on Computer graphics and interactive techniques.,* p. 276–283, 1979. |
| [34] | 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]. |
| [35] | D. Henrich, “Space-efficient region filling in raster graphics,” *The Visual Computer,* vol. 10, no. 4, pp. 205-215, 1994. |
| [36] | Unity, “Texture2D.GetPixels32,” Unity, [Online]. Available: https://docs.unity3d.com/ScriptReference/Texture2D.GetPixels32.html. |