Title Page:

Procedural Generation of a 3D City

Alexander Durkin
1103501
BSc (Hons) Computer Games Technology

Honours Dissertation

April 2015

University of Abertay Dundee
Permission to copy

Author: Alexander Durkin
Title: Procedural Generation of a 3D City
Degree: BSc (Hons) Computer Games Technology
Year: 2015

(i) I certify that the above mentioned project is my original work
(ii) I agree that this dissertation may be reproduced, stored or transmitted, in any form and by any means without the written consent of the undersigned

Signature............................................................................................................................................

Date........................................................................
I. Abstract

Open world games are now more popular than ever. They can come in all kind of genres and settings but the majority of these games include huge, detailed cities for players to explore. These model cities can often take several years to develop and at great cost. The purpose of this project was to research procedural generation techniques that could be, or have previously been, used to assist with the creation of a three-dimensional city. This project aimed to produce an algorithm capable of making a city structure which could be used as the foundations of a game level to help speed up the overall development process of an open world city game.

The main objectives for the project were to find existing and effective techniques for creating city components via procedural generation and to build an application, in Unity, that could take input from the user and produce varied outcomes each time it is ran. The main components to be included were the road network and the positioning of buildings around the city. The program was split into two main stages, the first stage controls the construction of the road network – three methods of generating the road network have been implemented. A ‘random’ method, a grid style method and a method based on Voronoi diagrams. All three can be used in one scene which allows for effective comparison between the methods. The second stage allows the user to set the maximum number of buildings in the scene. The user can watch the entire process in real time and explore the scene by manoeuvring the camera. Procedural generation techniques have been applied at each stage where necessary.

Not all procedural generation methods that were discovered during the research stage were implemented into the final project but the application does achieve different, interesting outcomes each time it is executed. The research and results from this project indicate that procedural generation techniques can be applied effectively to some areas of city level design such as the road network but it is difficult to create a great deal of realism without some manual alterations from the user.
II. Foreword

I would like to thank my university lecturers for their excellent tutoring throughout my time at university.

I would also like to thank Ruth Falconer and Henry Fortuna for their guidance over the course of this project.

I would also like to thank my family and friends for their invaluable support.
III. Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>I. Abstract</td>
<td>i</td>
</tr>
<tr>
<td>II. Foreword</td>
<td>ii</td>
</tr>
<tr>
<td>III. Table of Contents</td>
<td>iii</td>
</tr>
<tr>
<td>IV. List of Figures</td>
<td>v</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2. Literature Review</td>
<td>4</td>
</tr>
<tr>
<td>2.1. Open World Games</td>
<td>4</td>
</tr>
<tr>
<td>2.2. Previous Procedural Generation Projects</td>
<td>7</td>
</tr>
<tr>
<td>2.2.1. CityBuilder</td>
<td>8</td>
</tr>
<tr>
<td>2.2.2. CityGen</td>
<td>9</td>
</tr>
<tr>
<td>2.2.3. Undiscovered Worlds</td>
<td>10</td>
</tr>
<tr>
<td>2.2.4. CityEngine</td>
<td>11</td>
</tr>
<tr>
<td>2.3. Road Networks Summary</td>
<td>13</td>
</tr>
<tr>
<td>3. Methodology</td>
<td>14</td>
</tr>
<tr>
<td>3.1. City/Zone Areas</td>
<td>15</td>
</tr>
<tr>
<td>3.2. Road Networks</td>
<td>15</td>
</tr>
<tr>
<td>3.2.1. Random Network</td>
<td>15</td>
</tr>
<tr>
<td>3.2.2. Grid Network</td>
<td>16</td>
</tr>
<tr>
<td>3.2.3. Voronoi Network</td>
<td>17</td>
</tr>
<tr>
<td>3.3. Roads</td>
<td>19</td>
</tr>
<tr>
<td>3.4. Buildings</td>
<td>19</td>
</tr>
<tr>
<td>3.5. User Interface</td>
<td>20</td>
</tr>
<tr>
<td>4. Results</td>
<td>21</td>
</tr>
<tr>
<td>4.1. City/Zone Areas</td>
<td>21</td>
</tr>
<tr>
<td>4.2. Random Road Network</td>
<td>21</td>
</tr>
<tr>
<td>4.3. Grid Road Network</td>
<td>23</td>
</tr>
<tr>
<td>4.4. Voronoi Road Network</td>
<td>24</td>
</tr>
<tr>
<td>4.5. Roads</td>
<td>25</td>
</tr>
<tr>
<td>4.6. Buildings</td>
<td>27</td>
</tr>
<tr>
<td>4.7. User Interface</td>
<td>29</td>
</tr>
</tbody>
</table>
4.8. Output...........................................................................................................................................30
4.9. Test................................................................................................................................................34
5. Discussion..........................................................................................................................................35
6. Conclusion and Future Work...........................................................................................................36
7. References.........................................................................................................................................38
8. Bibliography........................................................................................................................................41
IV. List of Figures

Figure 1: Screenshot of a world in Minecraft ........................................................................ 2
Figure 2: Subversion City Generator .................................................................................. 3
Figure 3: Google Street map of London vs The Getaway map .............................................. 5
Figure 4: Google Street map of San Francisco vs Driver: San Francisco map ....................... 5
Figure 5: GTA 4 map vs street map of New York ................................................................. 6
Figure 6: Screenshot from GTA 5 ....................................................................................... 7
Figure 7: Grid, Organic and Mixed road networks from CityBuilder ................................... 8
Figure 8: Screenshot of CityGen ......................................................................................... 10
Figure 9: Screenshot of Undiscovered Worlds ................................................................... 11
Figure 10: CityEngine Manhattan map vs real Manhattan map ......................................... 12
Figure 11: Screenshot of CityEngine .................................................................................. 12
Figure 12: Several steps of an L-system ............................................................................. 13
Figure 13: Voronoi diagram ............................................................................................... 14
Figure 14: Voronoi Diagram with connecting circles ......................................................... 18
Figure 15: Buildings used in this project ......................................................................... 20
Figure 16: Early screenshot of the random method ............................................................ 22
Figure 17: Latest screenshot of the random method ........................................................... 22
Figure 18: Early screenshot of the grid method .................................................................. 23
Figure 19: Latest screenshot of the grid method ................................................................. 24
Figure 20: Voronoi Diagram problem ................................................................................ 24
Figure 21: Overlapping problem ....................................................................................... 25
Figure 22: Overlap solution ............................................................................................. 26
Figure 23: Close road problem ......................................................................................... 26
Figure 24: Close road solution ........................................................................................ 27
Figure 25: Building rotation problem ............................................................................. 28
Figure 26: Building rotation fixed ................................................................................... 29
Figure 27: Screenshots of UI.................................................................30
Figure 28: GTA 4 road network and grid method...............................31
Figure 29: GTA 5 road network and random method.........................32
Figure 30: Crackdown 2 road network and Voronoi method............32
Figure 31: Disconnected road............................................................33
Figure 32: Connecting networks problem.........................................33
Figure 33: Building rotation problem in grid network.......................34
Figure 34: Network generation time bar graph..................................34
Figure 35: Random method vs Voronoi method...............................35
1. Introduction

Simply put, procedural generation creates content governed by a set of rules in an algorithm as opposed to it being created manually. It has been a technique used in games for decades now. One of the first popular programs to use procedural generation was The Game of Life (1970) which simulated the random growth of cell populations over a 2D grid. In those days, when data storage space was very limited, procedural generation was mostly used to generate content in real-time to save space. Several games used procedural generation to create levels and textures while the game was running as opposed to having pre-made versions stored in memory. One of the most ground-breaking games of this time was *Elite* (1984). The game was a trading and combat simulator, set in space, which featured 8 galaxies each containing 256 planets that would all be procedurally generated as the player started up the game. Details of the galaxies such as the location and name of planets and the price of trade items were generated from information taken from a sequence of numbers produced from a in a random number generator that was based on the Fibonacci sequence (Knott 1996). This purely came about because of the restricted size of computer memory at the time of development.

Nowadays, the demand for massive amounts of content in games is continuing to grow and thanks to advancements in computer technology, games can now be much more finely detailed by storing massive amounts of content in memory. But large teams of programmers and artists are required to produce all of this content, usually over the course of several years. This is where procedural generation techniques can be useful. Procedural generation can be applied in games for many uses such as creating geometry, audio, animations, 3D textures and triggering in-game events. Making these features procedurally can save time and add an unlimited amount of variation instead of just saving storage space. Seemingly infinite, randomised worlds can be constructed as seen in the hugely popular *Minecraft* (2009), see Figure 1.
No Man’s Sky (2015), set to be released later this year, looks to expand on the phenomenon that was Elite. It is a space exploration game which is made up of an almost unlimited variation of planets, spaceships, creatures, trees, etc. The developers, Hello Games, spent a whole year creating a game engine specifically for the purpose of producing this procedural content. Lead developer, Sean Murray, was asked just how many planets will be in the final game to which he replied, “If you were to discover one every second, it would take 584 billion years” (Silver 2015). Elite could have been around the same size but the games publisher, Acornsoft, requested that it be made smaller for the audience’s sake (Spufford 2003). The content in both of these games was achieved by very small teams with the help of procedural generation.

As seen in these examples, and many other projects, procedural generation is often used to create natural, organic elements such as plants, mountains or planets but rarely man-made structures, such as cities. Large cities are very common in AAA games and a lot of time and money is spent on developing them – this consuming process could be made much simpler with the application of procedural generation techniques. There are some examples of cities being created with such techniques for smaller projects such as Introversion Software’s Subversion (not dated) and Structure Procedural System (Marco Corbetta and Miragoli Gianluca 2009). Subversion began in 2006 and looked promising but development was suspended in 2011 and no further progress has been made since, Figure 2 shows an early screenshot of one of Subversion’s procedurally generated cities.
Structure Procedural System was supposedly a tech demo, created by the head programmer and technical manager of Crytek, Marco Corbetta, and 3D artist Miragoli Gianluca for Crytek’s upcoming game Crysis 2. The demo showed off a large, procedurally generated city that allowed the player to enter every building and destroy its whole interior - very impressive but this level of detail was not included in the final game.

Steps have been made in the right direction, but no AAA game has ever included a procedurally generated city. This project aims to expand on established procedural generation techniques that could speed up the development process of creating city models by constructing road networks and correctly positioning buildings.

The research question for this project was:

*How can procedural generation techniques be used to create a road network, shape city zones and position buildings to help develop a 3D city model for use as part of a game world?*

The main objectives were:

- *Research existing techniques for creating city components via procedural generation.*
- Create a program that is capable of taking user input and procedurally generating a road network, defining zones and positioning buildings in a 3D city with different outcomes depending on the input.

- Evaluate the program by comparing the outputs to other procedurally generated cities and manually produced models used in popular games.

- Suggest improvements that could be made to the program and key areas of research that should be investigated further to find the most efficient way of procedurally generating a 3D city with minimal input from the user.

2. Literature Review

This section will look at popular games which are set in large cities and how procedural generation could have possibly improved their development process. It will also detail recent projects, based around the procedural generation of model cities, and describe any features that were adapted for use in this project.

Procedural generation has been around for a long time in 2D form but its popularity grew immensely in the 1980s thanks to the work done by Gardner (1984), Perlin (1985) and Peachey (1985) on 3D texturing. Since then, many more uses have been found for procedural modelling techniques and they have an almost unlimited number of potential uses within video game development. Although, since they create seemingly random outputs they are commonly used to create naturally occurring matter, like terrain and plant life or effects like shading and texturing but are seldom used to create artificial, manmade structures.

2.1 Open World Games

Designing a city for a game can be a very expensive and time consuming task. A lot of thought has to go into the planning of a city’s layout. This may be why many game developers choose to base their in-game cities off of famous locations from around the world such as:
The Getaway which was set in London, see Figure 3.

Driver: San Francisco which was set in a replica of northern San Francisco, as seen in Figure 4.
And GTA 4 which had areas based on central New York, see Figure 5.

Would game developers still choose to set their games in these real world replicas if they had software which could create a completely original world using procedural generation? Cities recreated for games are rarely exact copies of their real life counterparts, they may share identical main roads, iconic buildings and points of interest but the majority of each city, mainly the layout of the streets, are completely different in the game version – most likely because real life road networks are too boring for a game. This was confirmed by Nicolas Guerin, world level design director for Assassins Creed Unity, who said “It’s a better Paris than the actual Paris for gameplay... We have to build a game playground first, and on top of that make a cool city that’s visually striking and historically accurate as well” (Webster 2014). It is the roads in these areas that procedural generation could be used to help with development by not only creating content quickly but by making interesting designs and layouts that the development team might not have come up with on their own.
Open world games with cities that are not set in the real world, e.g. *Just Cause*, *Crackdown*, etc. could definitely benefit from the use of procedural generation. Much less time, and money, would have to be spent on designing and constructing an entire model city, street by street, building by building, if the developers could have one generated for them, at least as a foundation to begin working on rather than starting from scratch.

Aaron Garbut, Rockstar art director, said this about Los Santos - the main city in *GTA 5*, “In terms of raw man-hours, this has been way in excess of any project we’ve done at Rockstar before…It’s all handcrafted, all unique, and we’ve gone over it all again and again and again to make sure there’s enough layering of detail…” (J.Bernstein 2013). This suggests that cities/games of this calibre can only be achieved by companies as big as Rockstar. One day, with the help of procedural generation, something of this size could be realised by smaller development teams. Figure 6 shows a screenshot of the city of Los Santos in *Grand Theft Auto 5*, could this eventually be possible via procedural generation?

![Figure 6: Screenshot from GTA 5](image)

### 2.2 Previous Procedural Generation Projects

There have already been several detailed studies researching the use of procedural generation techniques for creating 3D cities that helped with the development of this project.
Here are a few of the most significant studies that were found:

### 2.2.1 CityBuilder

CityBuilder (Lechner, et al 2004) is a city generating system that uses agents which simulate the decisions made by an urban development team to create a 3D city model over a supplied terrain height map. The road network is constructed by two types of agents, extenders and connectors. Extenders randomly move across the terrain until an area with no current road network is found, they then tracks back, to the existing road network that they started from, on a path that can be influenced by terrain height and the type of network set by the user – either grid or organic. The discovered path is then proposed as a new possible network to a developer agent which runs a series of tests to check if it is acceptable. The connector agents randomly explore the existing road network and attempt to build small roads to new areas within a set radius of the current network.

Buildings are generated within the existing road networks based on decisions made by the developer agents. These agents will place building types based on data collected from the nearby areas. Residential buildings are placed away from any industrial areas or dense areas of the road network and close to any bodies of water. Industrial developers aim to place buildings away from residential areas and commercial developers look for areas with plenty of road access.

Figure 7 shows three examples of possible road networks generated in CityBuilder.
An adaption of the idea of checking for possible roads within a certain radius of the current network, used by the connector agents in CityBuilder, was used in the random and Voronoi methods of this project.

2.2.2. CityGen

Citygen (G.Kelly and H.McCabe 2007) is an interactive city building program that can generate buildings and roads over an uneven terrain map. The system is broken down into three parts, primary road generation, secondary road generation and building generation. The primary road generation works by placing nodes across the terrain and then several paths to these roads are sampled. The sampling takes into account the elevation of the terrain which is not a factor in this project as the terrain used is completely flat. Once all the nodes have been reasonably connected by small road segments, “city cells” are then extracted and stored. The city cells are the areas bound by the primary roads. Each city cell begins generating its own set of secondary roads within its bound area. The secondary roads begin growing parallel to each small road section in the primary roads and are extended by a “growth based algorithm similar to [L-systems]”, they are then snapped to nearby road segments when they come within a set proximity.

Building lots are identified as areas enclosed by roads and each lot gets rotated to become perpendicular to its nearest access road and algorithmically subdivided to reach a target size. Buildings are positioned differently in the lots depending on their type which can be residential, industrial or commercial. The buildings themselves are then procedurally generated within the subdivisions – another feature that does not apply to this project as it uses pre-generated building models. Figure 8 shows an example of a city being created in Citygen.
The idea of using nodes to position roads in Citygen formed the main basis of the road network scripts created for this project and the method of positioning buildings by orientating the lots depending on any access roads was also applied.

2.2.3 Undiscovered Worlds

Undiscovered Worlds (S. Greuter et al 2003) was a study that proposed a framework for creating a procedurally generated world in real-time and used a massive city as an example. The road network used in the project is a simple, repeating, grid shape layout as the generation and placement of the buildings was the main focus. As the user moves around the world, building spaces are filled with procedurally generated building objects as they come into view of the camera – this is called view frustum filling. When a building goes out of view it is removed from the world but its data is cached in memory so that it can be put back immediately if the player returns. This efficient use of memory and procedural generation allows the world to be “extremely large, consisting of 4x10^18 geometrically different buildings; about 600,000,000 times the world’s current human population.” Figure 9 shows a screenshot of Undiscovered Worlds.
This is an interesting framework which certainly adds to the evidence of the enormous amounts of content that procedural generation can be used to quickly create but the focus of the Undiscovered Worlds project was on handling the optimisation of this large capacity of content. These optimisation techniques could have been implemented if necessary but were not a major requirement for this application. The simple, grid style road network in Undiscovered Worlds formed the foundation of the final version of the grid network created in this project.

### 2.2.4 CityEngine

Perhaps the most technical and highly regarded procedural city generating application is CityEngine created by Y.Parish and P.Muller (2001). The system is split into smaller tasks like the previously mentioned projects. The main tasks are the creation of the road map, the division of building lots and the building generation. Image maps are required as input to control the city creation by identifying details such as areas of high population density, any land-water boundaries, terrain elevation, desired street patterns or zones (residential, commercial and mixed). An extended version of L-systems is then used to construct the road networks whilst taking into account the data provided by the image maps. The roads snap together in a similar fashion to those in Citygen (some elements of Citygen were based on CityEngine). CityEngine is capable of replicating real cities with amazing accuracy. Figure 10 shows a comparison between a road map procedurally generated by CityEngine, using image maps of Manhattan, and a real road map of Manhattan.
When the roads have finished generating, the city is subdivided into “blocks”. These blocks are then subdivided into building lots until a limit, which has been set by the user, is reached. All lots are then checked and any which are too small or are completely surrounded by other lots get deleted, the remaining lots are populated with buildings procedurally generated from an L-systems class and the buildings are procedurally textured. Figure 11 shows a screenshot from within a CityEngine product.
The downside of this project is the large amount of input required from the user, it appears as though the user still has to, effectively, design the city themselves in the form of several image maps that CityEngine can then work from. However, this does allow the user a great deal of influence over the final output which may be preferred in some cases. The idea of designating zones in the city as residential, commercial, etc. in CityEngine was adapted and used for this application.

2.3 Road Network Summary

The generation of the road network will be the most vital part of this project. In most of the studies looked at so far, the roads have been generated by some adaptation of L-systems (A.Lindenmayer 1968). L-Systems, effectively described at CGJennings.ca (2002), were originally developed to simulate the growth of bacteria and other simple organisms but, as seen earlier, can also be used to construct road networks and buildings in 3D city models. In their original form, they produce very repetitive patterns which can grow out of control very quickly, as seen in Figure 12, and so would had to have been heavily altered to become capable of producing a coherent road network which would have taken up too much time. For this reason, L-systems were not used in this project.

Voronoi diagrams (G.Voronoi 1907) were first described as a method of procedural generation by S.Worley (1996) when he used them as part of an algorithm to create an array of cells which could be used to produce textures such as skin or bark.
Voronoi diagrams are created by plotting a set of points on a plane and subdividing it into cells around these points. Each dividing line is placed exactly at the midpoint between a pair of neighbouring points.

J.Sun et al. (2002) conducted research on road generation in a virtual city and explained Voronoi diagrams as method that could be used to create road networks. The boundary lines between cells could be used as roads and the actual cells could represent building lots. Figure 13 shows an example of a Voronoi diagram.

![Voronoi diagram](image)

**Figure 13 Voronoi diagram**

### 3. Methodology

This section details the creation of the application. The main objective of the application was to implement procedural generation techniques discussed earlier in this document to generate variations of road networks and randomly position buildings across a city area. The goal was to have an application that could take input from the user and produce varying city models as a starting point to build upon rather than realistic looking final models.


3.1 City/Zone Areas

The initial idea for the city area was to have it made up of multiple zones such as suburban areas, industrial areas and the city centre. These areas would each have their own style of road network, the suburban areas could have an organic, irregular layout with lots of dead ends and cul-de-sacs whereas the industrial or city centre areas might look more structured with a straight roads in grid style layout.

The zones themselves were originally planned to be randomly shaped by procedural generation methods and then the roads would be created within the boundaries of the zones. The user would be able to set a rough size for the overall city area and each zone within it as well as the type of zone. The buildings that get placed in a zone would depend on its type. The city area is flat so all the roads, nodes and buildings are created on the same plane on the Y-axis.

3.2 Road Networks

Three styles of road network were to be implemented: Random, Grid and Voronoi. Each network would contain an array for the roads and an array for the nodes that the roads are drawn between. The Voronoi implementation would have to contain an extra array for the ‘Voronoi nodes’.

3.2.1 Random Network

The random road network was to function in a similar way to the Citygen project. The key parts of the Citygen road generation are designating a starting point and a destination and then sampling points in between to find a route for a road across possibly bumpy terrain. As the terrain used in this application is flat, the method was simplified to select a random number of nodes based on the overall size of the city area and randomly position these nodes across the area set by the user. When each node gets placed, its position needed to be checked against all other existing nodes to make sure no two nodes have been placed
too close together. If the new node is too close to another then it has to be assigned a new random position, this is repeated until a suitable location is found for every node. The node array is then rearranged to have the nodes in order of their Z coordinate from lowest to highest, this helps to produce more coherent networks.

The roads will then be drawn between the nodes. Rules are set in place to keep this function running as efficiently as possible, roads only get drawn between nodes within a set distance of each other, the current node only checks nodes ahead of it in the array to avoid redrawing earlier roads and each node has a maximum number of roads that can be drawn from it before the application moves onto the next node in the array.

Let iter1 = first node
Let iter2 = second node
If (iter1 > iter2)
{
    if (Distance(iter1, iter2) > MinRoadLength && Distance(iter1, iter2) < MaxRoadLength)
    {
        // Draw Road from iter1 to iter2
        CurrentNodesRoads++
    }
    If (CurrentNodesRoads == MaxNodeRoads)
        iter1++
}

A new pair of nodes get checked every frame so the user can watch the road network being created in real time.

3.2.2 Grid Network

The most effective plan for the grid layout was to set a number of nodes based on the size of the area and then to select a point near the bottom left corner of the set area and place the first node there and then to move across the X axis, placing nodes a set distance apart.
When the right edge of the area was reached, the next node is placed above the first node in the previous row and so on until a node's position exceeded the top right of the area.

```
If (X < TopRight.x && Z < TopRight.z)
{
    // Place node at X and Z
    X += PresetXgap
    Z += PresetZgap
}
```

```
If (X > TopRight.x)
    X = BottomLeft.x
If (Z > TopRight.z)
    // Finished
```

When the roads are being connected, each node only has to check for the node to its right and the node above it.

### 3.2.3 Voronoi Network

The Voronoi method seemed like it would be the most difficult to implement. While researching ways of implementing Voronoi diagrams, it became obvious that it is a complicated method to program for any use — not just as a road network. After lots of research and trial and error, an effective implementation was designed based on the diagram in Figure 14. The white circles connect three “Voronoi nodes” each and it is at the centre of these circles that road nodes are to be placed.
The function starts the same way as the random node method, by placing nodes randomly across the area. The array of nodes, referred to as Voronoi nodes, then get looped through and for each one, its two physically closest neighbouring nodes are found and stored. This is done by declaring a shortest distance variable and initialising it as infinity, then putting all of the Voronoi nodes into an array and going through them one by one, checking the distance to the current node each time. This distance is checked against the current shortest distance, if it is a lower value then it replaces the shortest distance and the next node gets checked until the end of the array is reached. At this point, the nodes with the current and second shortest distances are returned.

```plaintext
ShortestDistance = Infinity
SecondShortestDistance = Infinity
// Loop through all Voronoi nodes
{
    currentNodeDistance = Distance(OriginalNode, CurrentNode)
    If (CurrentNodeDistance < ShortestDistance)
    {
        SecondShortestDistance = ShortestDistance
        ShortestDistance = CurrentNodeDistance
        SecondClosestNode = ClosestNode
        ClosestNode = CurrentNode
    }
}
```
Each set of three nodes, the original one and its two closest neighbours, then get treated as if they are connected by a circle, the centre of this circle is where a road node needs to be placed. The centre is found by taking the line vectors of the current node and its two closest neighbours, getting the midpoint of each line, calculating perpendicular line vectors to the midpoints and then finding the coordinates at which the two perpendicular vectors intersect. This is explained in greater detail with sample code at Mathforum.org (1996).

### 3.3 Roads

The road objects will be created and named by a function that will be called every time a road is needed. The function will take the start and end node that the road is connecting and find the distance between them and the angle that they make with the X axis so that the road can be positioned and rotated appropriately. The start, end and angle of each road all get stored in separate arrays so that they can each be accessed by the building placer script when necessary. The start and end points are used when finding the closest road to current building being placed and the angle is copied to make the building parallel with its closest road.

### 3.4 Buildings

The buildings designed for the application are simple, premade objects that represent the types of buildings that are often found in in-game cities, e.g. skyscrapers, office buildings, factories/warehouses, small houses and apartment blocks as seen in Figure 15.
Placing the buildings around the road network is done by randomly selecting one of the buildings from a set (depending on the type of road network that the building is being placed in) and then randomly selecting a position within the set area. After being randomly placed onto the city map, the buildings will be checked for any collisions with roads or other buildings. If a collision occurs then the new building will be deleted. Once a building finds an acceptable position, the closest road to the building will be calculated and the building will copy its rotation so that it becomes parallel with it.

3.5 User Interface

A UI has been created to give the user several options to experiment with when creating a city. They can choose the size of the city area and can add up to three road networks—one of each type—which will then get connected together. When choosing the size and position of the networks, the user also gets to choose which type of zone each network is to form, which controls the selection of buildings that can be placed throughout this area. The user can set the maximum number of buildings that the algorithm can try to place. All variables have been given minimum and maximum values to prevent the user from accidently crashing the application.
4. Results

This section will cover the problems that were encountered during the implementation of the methods discussed so far and the solutions to these problems. It will also compare several screenshots of the output of this application with images of road networks in some open world games. Finally, results of the timed test that each road network was put through will be shown and explained.

4.1 City/Zone areas

The plan for the procedurally generated zone shapes proved difficult to implement as the chosen game engine, Unity, does not allow for mesh shapes to be customised very easily. To work around this, the user is asked to set the overall size of the city area and then to choose which type of zone they want and to define its size and position when they are creating the road networks. This does restrict the city to being only square or rectangular in shape but this did not affect the overall outcome of the project.

4.2 Random Road Network

The random method required some experimenting to achieve a desirable output. There are several variables that can have significant effects on the overall look of the network. The main ones being the number of nodes generated, the minimum distance apart that they have to be and the maximum length that the roads could be. An early, working implementation of the design can be seen in Figure 16.
It was thought that the roads looked too straight and unrealistic in this current state. The algorithm was later improved by increasing the number of nodes and reducing the minimum distance between nodes. An example of these changes can be seen in Figure 17.
4.3 Grid Road Network

Several methods were tried and tested to get the grid layout working properly. Some early implementations always produced a very basic and boring grid shape. See Figure 18.

![Figure 18: Early screenshot of the grid method](image)

To make the grid look less ‘perfect’, a percentage chance of drawing a road was added. Every time a road is drawn, the chance of the next road also getting drawn will drop slightly so that every so often there is a gap. When a gap occurs, the percentage will be reset to reduce the chance of any large gaps appearing. There was also a slight adjustment made to each node while being positioned across the X axis. A small, random number was added or subtracted to the nodes position to create a ‘loose’ grid effect. The same alterations made in the random road network function, e.g. nodes closer together, were made here to make the network more complex. The difference can be seen in the example in Figure 19.
4.4 Voronoi Road Network

Many problems were encountered whilst designing the Voronoi diagram method. The main problem was figuring out where to position each road as they had to be placed strategically around certain points to form Voronoi cells. In the project by Sun et al. (2002), the Voronoi cells were formed around points that signified largely populated areas. As there were no details along those lines in this project, randomly placed points were used instead. These points are the Voronoi nodes mentioned in the Methodology section. From here, the circle method described earlier was used. But getting the algorithm to connect appropriate road nodes proved just as difficult as placing them. How could the algorithm know to not connect to two red highlighted node or the two green ones in Figure 20?
In the end, this was only controlled by distance between the nodes. If they were close enough, then a road was drawn. This means that the Voronoi road network produced in this project is not a perfect Voronoi diagram but it does create an acceptable output.

Another problem was that, occasionally multiple road nodes may get placed outside of the set area or on top of an existing node. If this happens, the offending node is given a tag which tells Unity to avoid connecting any roads to it and to destroy it later.

4.5 Roads

There was a problem with the road objects early on in the project. When the roads are being drawn, the algorithm is only set to check if two nodes are close enough for a road and then it creates a road of appropriate length to connect the nodes and rotates it to line up with the nodes. This led to many roads being drawn on top of one another, as seen in Figure 21, this is something that a human level designer would be able to easily avoid but how could this be solved within an automated algorithm?

![Figure 21: Overlapping problem](image)

The solution was to add colliding boxes to the roads and if a collision was detected between a pair of roads then the larger one would be deleted. The colliding boxes on the roads had to be adjusted to make them slightly smaller than the roads themselves to allow for any
slight overlaps at the start/end of each road. FIGURE “” shows the altered size of the colliding box.

When it came to finding the closest road to a building, the program often made mistakes. Again, this was something that a level designer could easily see and do but was difficult to accomplish with an automated algorithm. The problem came about because the road position was taken from the bottom left corner of the road object, so a building could appear to be closer to a certain road but be rotated towards a different one because it was closer to its bottom left corner as seen in Figure 23. The building is closer to road “1” but has been rotated to match road “2” because it is closer to the bottom left corner of road “2”.

![Figure 22: Overlap solution](image)

![Figure 23: Close road problem](image)
To fix this, functions were added which calculated the distance between the building and the closest point, not just the bottom left corner, on every road. This algorithm was adapted from code posted as an answer to a question on Stackoverflow.com (2012). The road with the closest point overall gets set as the buildings closest road. The outcome of this can be seen in Figure 24.

![Figure 24: Close road solution](image)

### 4.6 Buildings

The buildings were now all rotated perfectly, but many were positioned far from any roads which did not look very realistic. The solution to this was to find each buildings forward vector and multiply it by the distance between the building and its closest point on a road (minus a small amount to position it just off of its parallel road). But this presented yet another problem that proved complicated for the algorithm to sort as seen in Figure 25.
Both of these buildings have the same closest road and have been rotated to be parallel with it but by doing so, the forward vector of the building on the right is now facing away from the road and will be moved in this direction instead of towards the road. To overcome this problem, code had to be added to each building's script which works out where the building is in relation to its closest point on a road, before it is moved, and its forward vector gets reversed if necessary. If the building is in a position where reversing its forward vector will not help, e.g. it is facing the correct direction on the X axis but not on the Z axis, then it gets deleted and a new building is created in a random position.

```cpp
If (BuildingAngle >= 0.0 && BuildingAngle < 90.0) // Building is in quadrant 1
{
    If (BuildingPosition.x > Point.x || BuildingPosition.z > Point.z)
    {
        If (BuildingPosition.x > Point.x && BuildingPosition.z > Point.z)
            // Reverse buildings forward vector
        Else
            // Delete building and place a new one
    }
}
// Repeat for quadrants 2, 3 and 4
```

Once a building is in its new position, it is checked again for any collisions. Figure 26 shows what the buildings looked like after this fix.
When the road networks had been implemented fully, the selection of buildings got added to a script and a random number generator was set up to select which building instance is to be placed each frame. The user can watch the buildings being generated in the scene in real time. A rule was set in place to stop the application from becoming stuck in an infinite loop, if too many frames go by without a new building being placed then it is concluded that there is not enough space for another building and the loop stops. The frame rate tends to drop significantly as the scene becomes more intricate, larger scenes would have to be left to render for several hours but this is a massive improvement over the time that it would take to create the city manually.

4.7 User Interface

There are several other possible variables that could be changed to have a considerable effect on the how the final version of a city looks, such as the minimum space between nodes, the minimum/maximum length of roads, the width of roads, the maximum number of roads that one node can have connected to it and the percentage chance of roads being drawn.skipped in the grid network. These variables, and others, have been pre-set to acceptable values and hidden from the user as it may have been too difficult to achieve a
desirable city model with so many options to set. Figure 27 shows some examples of the UI screens that the user will come across in the application.

![Screenshots of UI](image)

**Figure 27: Screenshots of UI**

### 4.8 Output

Figure 28 shows an area of the map from Grand Theft Auto 4 and a screenshot from this application using the grid method. There are clear similarities in these road networks.
Figure 29 shows a screenshot from a countryside area on the map of Grand Theft Auto 5 and an example of the random road network. Again, there are some similarities in the two networks. The roads in the GTA 5 screenshot are smoother but this could be adjusted by increasing the number of nodes used in the application.
FIGURE 30 compares a screenshot from Crackdown 2 and a screenshot of the Voronoi diagram method. This method can produce networks that closely resemble the random method but are usually more tightly packed with roads.
There are a few errors in the final application that solutions could not be found for, such as:

Occasionally, a road will be drawn and then be cut off from the network by an intersecting road and then the intersecting road will also be cut off which results in the output seen in Figure 31.

![Figure 31: Disconnected road](image)

When connecting the different types of networks together, the algorithm draws roads between each network in a loop until a minimum number is reached. A function is then ran which makes sure there is at least one connecting road between the networks so that a car would be able to start in any network and reach any of the other networks. E.g. there could be a road joining networks 1 and 2 and another joining 2 and 3 but none joining 1 and 3 – this is acceptable but sometimes only one road will be drawn which produces the output seen in Figure 32.

![Figure 32: Connecting networks problem](image)
The last error only seems to appear within the grid network. Buildings will sometimes be rotated incorrectly as seen in Figure 33.

4.9 Testing

Each method was tested using various city sizes to see how long it takes to fully generate a road network. The results of these tests can be viewed in the bar graph in Figure 34.
The grid method was the quickest by far but also has the least amount of influence from procedural generation and each node only has to check for two other nodes at a time. The time taken to complete a road network using the Voronoi method increases drastically as the size of the city area is increased. This is due to the number of Voronoi nodes that have to be checked and ignored due to them being outside of the area or in the same position as an existing node. The Voronoi method also checks for one more possible road per node than the random method when drawing the roads; this is what makes the network usually appear tighter with less open spaces. Figure 35 shows a direct comparison between the Voronoi and the random methods. The random network, on the left, contains larger open areas whereas the Voronoi method, on the right, has fewer long roads and has a more structured, less organic look to it.

Figure 35: Random method vs Voronoi method

5. Discussion

Regarding the research question:

*How can procedural generation techniques be used to create a road network, shape city zones and position buildings to help develop a 3D city model for use as part of a game world?*
It was discovered early on that Unity did not initially allow for the shaping of free form meshes that were planned to be used for the city zones, because of this, the road networks and building positioning became the main focus of the project. The screenshots comparing the applications outputs with road maps of sections in manually produced game cities adds to the evidence already out there that procedural generation techniques can be used to create man-made structures. Several ways of creating road networks were tested with positive results, particularly the random method, which was based on the technique used in Citygen. The random method is capable of producing a much more varied set of outputs than the other techniques and runs a lot faster than the Voronoi method. Only one, simple method of placing the buildings around the city was implemented in the final application and even though it does not use ‘lots’ as seen in the projects looked at earlier it still works quite effectively.

Unfortunately, even though it seemed to be the most effective method used in the past, L-systems were not included in the final application for this project. As explained earlier, the original L-systems algorithm needed to be heavily modified to be capable of producing a road network and this would have taken up too much time.

Overall, procedural generation techniques clearly have a place within 3D city generation and it is only a matter of time before we see the first AAA game set in a completely procedurally generated city.

6. Conclusion and Future Work

Procedural generation has played a part in creating many examples of naturally occurring phenomena such as galaxies, terrain, plant life and textures for several decades now. But it is only in the last decade or so that procedural techniques have been experimented with when creating urban environments and promising results can already be seen in previous city generation projects. The main bulk of work that took place in this project concerned the road networks and, although it is the slowest method used, the Voronoi method produced a
very successful output despite being rarely used as a method of creating road networks in the past. The other methods can also be looked at as successful when compared to high standards set by other projects such as CityEngine and Citygen.

The main problem with procedural generation techniques being applied to man-made structures in games is the possibility of repetition which risks boring the player. It will be difficult to discover a system which is capable of procedurally generating a truly realistic city that would be expected in a game without alterations being required by a developer, but this research has shown that steps are being made in the right direction. Further exploration of the methods used in this project, as well as the unused methods experimented seen in previous projects such as agent based simulation and L-systems, could possibly lead to the perfect system.
7. References

Bernstein, J. 2013. "Way Beyond Anything We've Done Before": Building The World Of "Grand Theft Auto V". [online] BuzzFeed. Available at:

CGJennings.ca, 2002. Lindenmayer Systems. [online] Available at:


Gardner, G. Y. 1984. Simulation of Natural Scenes Using Textured Quadric Surfaces. SIGGRAPH '84. [online] Available from:


Introversion Software. Not Dated. Subversion Development Diary. [online] Available from:


8. Bibliography


