A PROCEDURAL 2D ROAD NETWORK GENERATION APPROACH

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF INFORMATICS
OF
MIDDLE EAST TECHNICAL UNIVERSITY

NEVZAT GÖKSAN GÜNER

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
MULTIMEDIA INFORMATICS

JANUARY 2018

Approval of the thesis:

A PROCEDURAL 2D ROAD NETWORK GENERATION APPROACH

submitted by **NEVZAT GÖKSAN GÜNER** in partial fulfillment of the requirements for the degree of **Master of Science in Modeling and Simulation, Middle East Technical University** by,

Prof. Dr. Deniz Zeyrek Bozşahin
Dean, **Graduate School of Informatics** ______________

Assoc. Prof. Dr. Hüseyin Hacıhabiboğlu
Head of Department, **MODSIM, METU** ______________

Assoc. Prof. Dr. Hüseyin Hacıhabiboğlu
Supervisor, **MODSIM, METU** ______________

Dr. Çağatay Ündeğer
Co-supervisor, **General Manager, SimBT Ltd.Sti.** ______________

Examining Committee Members:

Prof. Dr. Bülent Özgüç
Computer Engineering, İhsan Doğramacı Bilkent University ______________

Assoc. Prof. Dr. Hüseyin Hacıhabiboğlu
MODSIM, METU ______________

Assoc. Prof. Dr. Ahmet Oğuz Akyüz
Computer Engineering, METU ______________

Assist. Prof. Dr. Elif Sürer
MODSIM, METU ______________

Assoc. Prof. Dr. Alptekin Temizel
MODSIM, METU ______________

Date: 12.01.2018

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: NEVZAT GÖKSAN GÜNER

Signature :

ABSTRACT

A PROCEDURAL 2D ROAD NETWORK GENERATION APPROACH

Güner, Nevzat Göksan
M.S., Department of Modeling and Simulation
Supervisor : Assoc. Prof. Dr. Hüseyin Hacıhabiboğlu
Co-Supervisor : Dr. Çağatay Ündeğer

January 2018, 56 pages

Modeling high scaled road networks with many intersections and distant streets takes a serious amount of modelers' time. The amount of work modelers need can be reduced by generating the basic model procedurally; therefore, modelers spend more time to work on the details. In this thesis, we focus on procedurally generating low detailed and lightweight 2D road network models with dynamic lanes and lines. Our system also has the same basic abilities with the traditional road network modeling tools capable of creating junctions and pavements. In order to make a better performing model, we use straight and arc road segments consisted of minimum amount of polygons. The main focus of this thesis will be creating a 2D procedural road network generation system with decreased polygon and texture costs.

Keywords: computer graphics, procedural 2d modeling, road networks

ÖZ

PROSEDÜREL 2-BOYUTLU YOL AĞI YARATIMI YAKLAŞIMI

Güner, Nevzat Göksan
Yüksek Lisans, Modelleme ve Simülasyon
Tez Yöneticisi : Doç. Dr. Hüseyin Hacıhabiboğlu
Ortak Tez Yöneticisi : Dr. Çağatay Ündeğer

Ocak 2018 , 56 sayfa

Geniş çaplardaki yol ağlarını modellemek, modellemeciler için ciddi zaman gerektiren bir iştir. Eğer modelin temel kısımlarını prosedürel olarak daha pratik bir şekilde yaratırsak, modellemecilere detaylar üzerinde çalışmaları için daha fazla zaman sağlayabiliriz. Biz bu tezde düşük detay ve hafif yükte iki boyutlu yol ağı modelleri oluşturulmasına yoğunlaşıyoruz. Aynı zamanda istendiği gibi yol şeritlerinin ve çizgilerinin değiştirilmesine olanak sağlayan yeni bir yöntem tanıtıyoruz. Sistemimiz geleneksel yol ağı modelleme uygulamalarında yer alan kavşak ve kaldırım gibi temel öğeleri de içerisinde barındırıyor. Daha iyi performansa sahip olan 2 boyutlu modeller üretebilmek adına düşük sayıda poligonlardan oluşan düz ve çembersel yol parçaları kullanıyoruz. Bu tezde poligon ve doku masraflarını kısarak 2 boyutlu prosedürel yol ağı üretebilecek bir sistem oluşturmayı amaçlıyoruz.

Anahtar Kelimeler: bilgisayar grafikleri, prosedürel 2 boyutlu modelleme, yol ağları

To my family and friends

ACKNOWLEDGMENTS

I would like to express my gratitude to my supervisor Assoc. Prof. Dr. Hüseyin Hacıhabiboğlu and my co advisor Dr. Çağatay Ündeğer for their guidance and support.

I would also like to thank my friends: Ekim Taylan for his trigonometrical teachings and Murathan Kurfalı for his academical insight;

Bora Yalçıner, Sema Kamışlı, Hakan Güler and Nail Akıncı for their support in various topics.

Lastly, I would like to thank my family for their support through my whole education life.

TABLE OF CONTENTS

ABSTRACT . . . iv

ÖZ . . . v

ACKNOWLEDGMENTS . . . vii

TABLE OF CONTENTS . . . viii

LIST OF TABLES . . . xi

LIST OF FIGURES . . . xii

LIST OF ALGORITHMS . . . xvi

CHAPTERS

1 INTRODUCTION . . . 1

1.1 Motivation . . . 2

1.2 Objective and Contribution . . . 4

1.3 Structure . . . 4

2 PREVIOUS WORK . . . 5

2.1 Polygonal Modeling . . . 5

2.1.1 Vertices . . . 5

2.1.2 Triangles . . . 5

2.1.3 Textures . . . 5

2.1.4 UV Mapping . . . 6

2.1.5 Normals 6
2.2 Mesh File Types and Data Structures 7
2.2.1 Triangle Soup Data Structure 8
2.2.2 Shared Vertex Data Structure 8
2.2.3 Winged-Edge Data Structure 9
2.2.4 Half-Edge Data Structure 10
2.3 Procedural Techniques 12
2.3.1 Fractals 13
2.3.2 Perlin Noise 14
2.3.3 Tiling 16
2.3.4 L-Systems 16
2.4 Road Network Generation Approaches in City Generator Systems 18
2.4.1 Grid Based Approach 18
2.4.2 L-Systems Based Approach 18
2.4.3 Agent Based Approach 19
2.4.4 Aerial Image Processing Approach 20
2.5 Unity Game Engine 20
3 PROPOSED METHOD 23
3.1 Overview 23
3.2 Road Topology 23
3.3 Road Lines 25
3.4 Junctions 27
3.5 Pavements 29

3.6 Nodes 31

3.7 Pseudo Code 32

4 RESULTS AND DISCUSSION 35

4.1 Comparison with CityEngine 2017 35

4.2 Discussion About Comparison with CityEngine 2017 42

4.3 Large Network Generation 46

5 CONCLUSIONS AND FUTURE WORK 51

REFERENCES 52

LIST OF TABLES

Table 4.1 Vertex and triangle counts for the road network models with few junctions with many straight and arc road segments. Both of the models are shown in Figure: 4.3. 37

Table 4.2 Vertex and triangle counts for the road network models with few arc road segments but many straight road segments and junctions. Both of the models are shown in Figure: 4.4 37

Table 4.3 Vertex and triangle counts for the road network models with no arc road segments but straight road segments and junctions. Both of the models are shown in Figure: 4.5 41

Table 4.4 Vertex and triangle counts for the sample road segment models from Figure: 4.6 and Figure: 4.7. 41

LIST OF FIGURES

Figure 1.1 Need for Speed uses detailed and highly customized road models. . 1

Figure 1.2 City Island Builder Tycoon, road network made of tiling road segments. 2

Figure 1.3 Google Maps' iconic illustration of a road network. 3

Figure 2.1 A cube mesh consists of vertices (v), edges (e), and triangles (t). . . 6

Figure 2.2 A cube with a checkered texture 7

Figure 2.3 Normal vector versus lightning vector from different angles. 7

Figure 2.4 A checkered cube with normals (indicated by blue lines), shaded by the directional light. 8

Figure 2.5 Winged edge data structure, the bold line indicates the selected edge. Left and right sides are with respect to the selected edge. 10

Figure 2.6 Half edge data structure, the bold line indicates selected edge. . . . 11

Figure 2.7 Fern is fractal structured plant in nature (taken from [1] under CC BY-SA 3.0). 13

Figure 2.8 An example to Mandelbrot Set (taken from [2] under CC BY-SA 3.0). 13

Figure 2.9 Perlin Noise outputs on different parameter values of frequency "f" and amplitude "a" (modified from [3] under GNU General Public License as published by the Free Software Foundation) 14

Figure 2.10 A terrain created by using Perlin Noise (taken from [4], declared for public domain use). 15

Figure 2.11 An environment created by a small tile set (taken from [5], declared for public domain use). 16

Figure 2.12 String representation of L-system, replacing characters according to the rules. 17

Figure 2.13 A tree created by an L-systems algorithm (taken from [6] and modified under CC BY-SA 3.0). 17

Figure 3.1 If both consequent directional normalized vector values are same between nodes, straight road segments are inserted. $P_2 - P_1 = v_1$ and $P_3 - P_2 = v_2$, if we normalize $v1$ and $v2$ both will have (1,0,0) value, hence between nodes P_1, P_2 and $P_2\, P_3$ straight road segments are placed. 24

Figure 3.2 The geometry problem for finding the arc angle α. 24

Figure 3.3 a) is a widening road, b) is a narrowing road. 26

Figure 3.4 A road with various lines and their polygons. 26

Figure 3.5 An example junction connecting 3 roads and its triangulation. 3 triangles for roads, and 3 triangles for connecting sequential roads. 28

Figure 3.6 Closed-loop roads can be achieved by using junctions. 29

Figure 3.7 Blue points denote vertices and green points denote nodes. Two parallels from previous and next straight road segments are casted by pavement width (w) distance from A and C points to find intersection point of pavements for the inner part of arc road segment. For the outer part, outer pavement vertices are extruded from the outer arc vertices. 30

Figure 3.8 From the center of the junction road segment line, a parallel distanced by the pavement width is casted. Also two parallels from road sides distanced by pavement width are casted. The intersection points of these lines construct the base points of a junction pavements. The same procedure is repeated in a clockwise order for each road connection part in the junction. 31

Figure 4.1 CityEngine pavement model has 2 parts of faces, one for pavement stones, another for sidewalk. But our model only uses sidewalks; therefore, we calculated how the results would be if CityEngine also had pavements similar to our pavements and used these results for fair comparison. 36

Figure 4.2 Our pavements only have the sidewalk part. 36

Figure 4.3 Two almost identical road networks with few junctions and many straight and arc road segments. Vertex and triangle counts are displayed in Table: 4.1. The proposed method's output is at the top, while CityEngine's at the bottom 38

Figure 4.4 Two almost identical road networks with few arc road segments but many straight road segments and junctions. Vertex and triangle counts are displayed in Table: 4.2. The proposed method's output is at the top, while CityEngine's at the bottom 39

Figure 4.5 Two almost identical road networks with no arc road segments but straight road segments and junctions. Vertex and triangle counts are displayed in Table: 4.3. The proposed method's output is at the top, while CityEngine's at the bottom 40

Figure 4.6 One almost identical straight road segment from CityEngine (on left) and the proposed method (on right). The polygon count difference comes from our line based approach and CityEngine's lane based approach. The triangles consist the road segments are shown in blue. 42

Figure 4.7 One almost identical arc road segment from CityEngine (on left) and the proposed method (on right). Again the polygon count difference comes from our line based approach and CityEngine's lane based approach. The triangles consist the road segments are shown in blue. 42

Figure 4.8 CityEngine uses a fixed 4 dashed line road lanes texture no matter the number of road lanes exist. 45

Figure 4.9 Procedurally generated 25x25 grid based road networks. Random distance parameter values increase along with the grid size 47

Figure 4.10 The generation time results for grid size 4 (2x2) to 1225 (35x35). . 48

Figure 4.11 The memory consumption results for grid size 4 (2x2) to 1225 (35x35). 48

Figure 4.12 The triangle count results for grid size 4 (2x2) to 1225 (35x35). . . 49

Figure 4.13 The road and junction count results for grid size 4 (2x2) to 1225 (35x35). 49

LIST OF ALGORITHMS

ALGORITHMS

Algorithm 1 Triangle Soup data structure. 9

Algorithm 2 Shared Vertex data structure. 9

Algorithm 3 Winged Edge data structure. 10

Algorithm 4 Half Edge data structure. 12

Algorithm 5 Pseudo code for the whole procedure 33

CHAPTER 1

INTRODUCTION

Today video games are using procedurally generated content[7][8] in many branches such as levels, landscapes, items, quests, etc. Road generation is a suitable category in this context as well. Other than video games; simulations, city engines and various other products use roads in their virtual environments. While some of these might need high level of customization, others might need a simple visual representation of the road network. The racing game “Need for Speed” (See Figure 1.1) is an example of highly customized roads. However, many games like “City Island: Builder Tycoon” (See Figure 1.2) just use few modular road segments, which can tile to each other, to construct their whole road networks. Road network do not have to be detailed and realistic, they can also be iconic; e.g., Google Maps (See Figure 1.3). Whether the level customization and detail is high or not, procedurally generating road network models as an assisting modeling tool would save remarkable amount of time.

Figure 1.1: Need for Speed uses detailed and highly customized road models.

Endless(runner) games, which the player continues moving further until the game is over, need to generate infinite content, because the player may not lose the game

Figure 1.2: City Island Builder Tycoon, road network made of tiling road segments.

for hours. Another example would be games with random level generation from the beginning. Some games want all players to have different experiences from the levels as a design choice. Therefore, such games might need randomly generated roads for their environments as well. Creating such content needs a huge amount of work, may not provide sufficient variant or be sustainable for the hardware. Generating road networks procedurally can both save time, increase variety, and burden hardware less.

This work aims to propose a lightweight procedural 2D road model generation approach to cater the needs mentioned above.

1.1 Motivation

The main motivation of this work is saving the hours of laborious work of modelers and giving them the possibility to create endless road environments even in run-time. Regardless of the modeler's talent, manually modeling even a simple road network can take a long time. Even if the raw output of the procedural model is not sufficiently satisfactory, modeler can work on the details after creating the main road network in a short time. We included pavements, junctions, road line customization, road lane transitions, and their settings in our tool.

Our primary focus in this thesis is explaining our low polygon road network generation approach while introducing a novel modeling approach of separated lines, yet we want to strengthen our argument by adding some common advanced features to state our approach is not limited and has the potential for further developments. A secondary motivation for this work is providing games and other applications running large scaled road networks even in lesser devices such as mobile phones. Large cityscapes and open

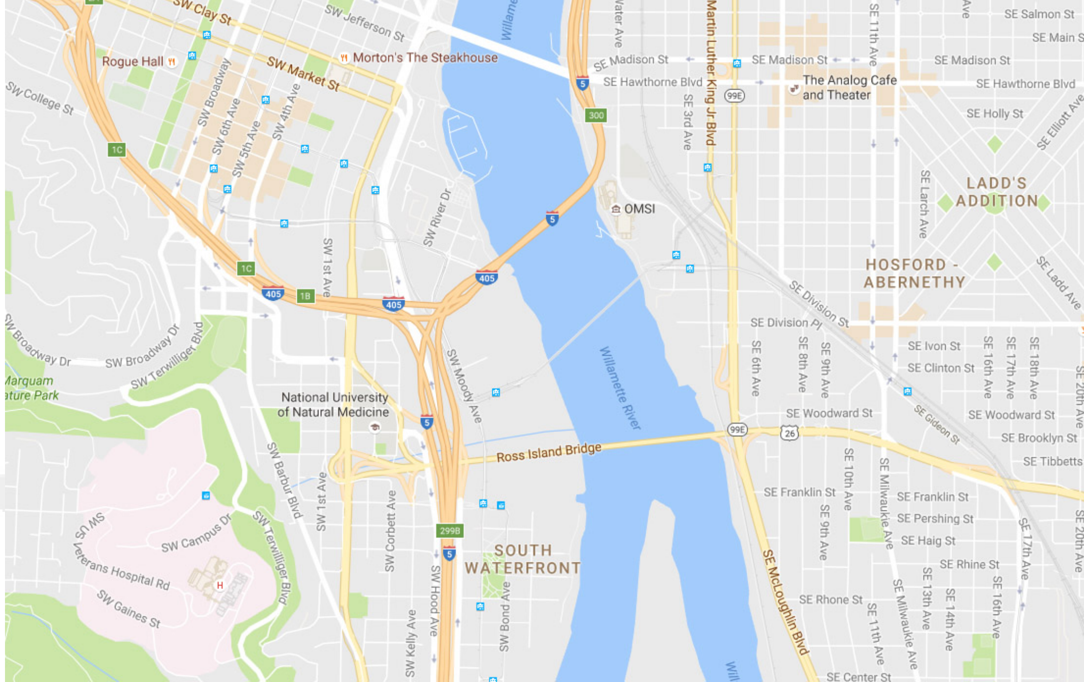


Figure 1.3: Google Maps' iconic illustration of a road network.

world environments are still problematic to run in every device. Although the hardware technology advances every day, advancing the software is the other crucial part. Our technique aims to create road models that can satisfy graphical and detail needs of some products, and able to work in run time without any difficulty. We target generating low cost models to work in devices with limited hardware. However, the proposed tool is not limited to low end, simple graphics. It can be customized to provide more detailed visuals or simplified for making a GPS (global positioning system) application (See Figure 1.3). Our tool offers some customization settings, whether the user wants detailed or simpler models is optional.

In traffic engineering, there are many researches for stating the importance of road markings and lines. According to [9], if edge lines and centerlines exist, all accidents reduce by 20%, moreover, 34% for single vehicles [10]. Edge lines reduce the accident rates, however, they have not proven to reduce the speed of the vehicles[11]. In contrary, in some cases edge lines cause drivers to increase their speed. This is possibly due to increased confidence [12] caused by edge lines. Another research concludes that road lines [13] reduce the work load of the drivers by directing them. Therefore, they are less busy for considering staying on the path. As a result, increased control causes higher driving speeds compared to roads without edge lines. Nonetheless, higher speed does not imply exceeding the speed limit [13]. There are also researches conducted for night driving safety. [14] states that road lines and markings help preventing night time accidents. Therefore, providing a modeling system based on road lines could be valuable for such researches in terms of simulation or application purposes.

1.2 Objective and Contribution

Main objective of this study is offering an alternative approach which has higher performance than the road model generation outputs made by step size based algorithms. Creation of curvy shapes such as bezier and spline [15] based roads need step sizes; nonetheless, straight roads does not need step sizes. We propose a more flexible solution for constructing road networks based on CityEngine (See Section:2.4.2) and arcs.

Our optimization has some limitation by the terms of geometrical shape detail, however, it can provide a lightweight solution for many products. Our tool creates the road mesh by dynamically shaped straight and arc (elbow) road segments with asphalt texture on them. Then road lines are inserted on the road mesh as separate faces from the road. It is common to use asphalt and road lines in the same texture. However, our approach trades textures with more polygons (See Section: 3.3), none of the road line textures are mixed with asphalt textures, which is more advantageous in certain cases. Additionally we implemented changing road lanes, which can also be adapted by other tools. Trading textures with extra polygons trade becomes more essential if the model includes different kinds of roads with various line combinations and changing number of lanes.

The proposed method and this thesis only cover modeling, which is the producing output part. How the modeling should be is the input part and it is another extensive topic. Basically, our method can generate road networks with random or decent inputs, but it is not capable of producing inputs, i.e how the roads inputs of a growing city would be. However, for illustrating that the proposed method is capable of generating big random road networks, a simple example will be shown. We use Unity3D game engine for the proposed method's implementation and visualization.

1.3 Structure

In this chapter we presented our motivation, objectives and contribution as an introduction.

In Chapter 2, we will give background information step-by-step starting with polygonal modeling, mesh and its data structures. We will continue with popular procedural techniques and finish with procedural road network generation usage in city generator systems. Lastly, we will give brief information about Unity Game engine and how it is utilized.

Our proposed method will be explained in detail in Chapter 3. We will provide an analysis and give information about costs for each key method we use. Later we will compare the proposed method with CityEngine in Chapter 4. We will use almost identical samples as our experiments and report them. We will also show how the results change as the network grows by using our simple grid based large network input generator.

The conclusion and potential future work is discussed in Chapter 5.

CHAPTER 2

PREVIOUS WORK

2.1 Polygonal Modeling

In this chapter, we will briefly introduce some fundamentals of polygonal modeling which we will be using. A polygonal mesh consists of vertices, edges and faces. These components are used for defining shapes of 3D polyhedral objects in computers graphics. For simpler rasterization purposes, faces are usually represented using triangles (triangles are chosen in this work) or other simple convex polygons[16]. Additionally, a 3D model also has UVs, normals, and texture properties, which will be introduced briefly in the following sections.

2.1.1 Vertices

Vertices are points in the 3D space and essential data structures in computer graphics. Their positions and quantity represent the mesh's detail. When three vertices are bound, they form a triangle.

2.1.2 Triangles

Two vertices comprise an edge. With three vertices connected as a loop, they make three edges, a triangle. The triangle has two sides, but this is a surface; therefore, only one side is shown. In order to state which side to be shown, we connect vertices either in clockwise or in counterclockwise direction. When all the triangles combine, they produce the mesh (See Figure 2.1).

2.1.3 Textures

Textures are 2D images used to cover by 3D models (See Figure 2.1). The most basic texture is used for coloring the surface of the model. UV Mapping is used to display textures on surfaces. Some of the other texture types are used for the following properties: specular color, bump, displacement, etc.

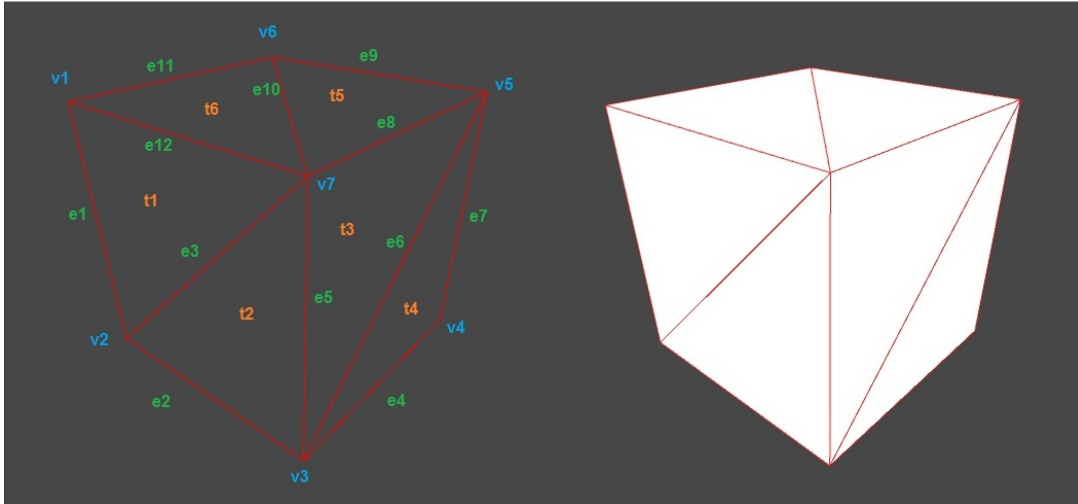


Figure 2.1: A cube mesh consists of vertices (v), edges (e), and triangles (t).

2.1.4 UV Mapping

U and V are axes of the 2D texture, needed for assigning the texture onto a 3D object. There are two types of mapping [7], non-parametric texture mapping and parametric texture mapping. Non-parametric texture mapping has fixed size texture and orientation, and these are unrelated to the polygons' size and orientation. Whole surface is tiled in the same way. With parametric texture mapping, however, texture size and orientation are tied to the polygons. This is done by separating texture space and screen space, then texturing the polygon as before but this time in the texture space. Finally rendering the textured polygon on screen space. UV mapping aims to place pixels of the texture image correctly on polygon segments. UV coordinate points' quantity is same with the vertices' quantity for correct mapping. At the end, while rendering, UV texture coordinates are used to colorize the 3D surface [17].

It is possible to use different UV sets for different textures[18]. Each texture is tiled according to their UV sets. Hence, this would result with having different effects for the objects such as setting frequency of repeating tiles.

2.1.5 Normals

Setting triangles are needed for constructing the simple shape of the 3D object. However, for a 3D object to be shaded correctly, extra information is needed. A normal vector is assigned to each vertex for correct interaction with the lightning. A normal vector is assigned on the vertex points to a direction. This direction is compared with the lighting received by the vertex position, this lighting is also a vector. After the comparison, if both vectors are coincident, then the surface is illuminated with full brightness of the lighting. As their alignments change, shading results differently between full brightness and darkness (See Figure 2.3). Even though there are enough vertices to shape a geometry, sometimes for a correct texturing and shading extra ver-

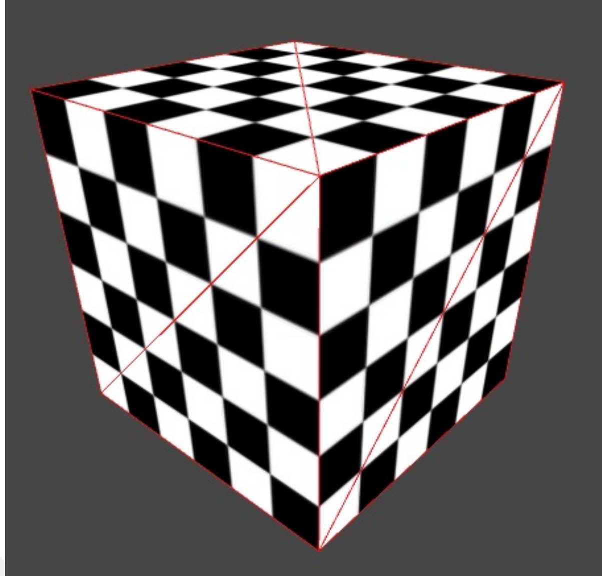


Figure 2.2: A cube with a checkered texture

tices are needed. For example a cube has 8 corners, so 8 vertices would be enough for its geometry. However, one corner vertex is shared by 3 faces. In this case 3 faces cannot be shaded by one normal. And since we cannot assign 3 normals for a vertex, we need 3 vertices with 3 normals for that corner (See Figure 2.4).

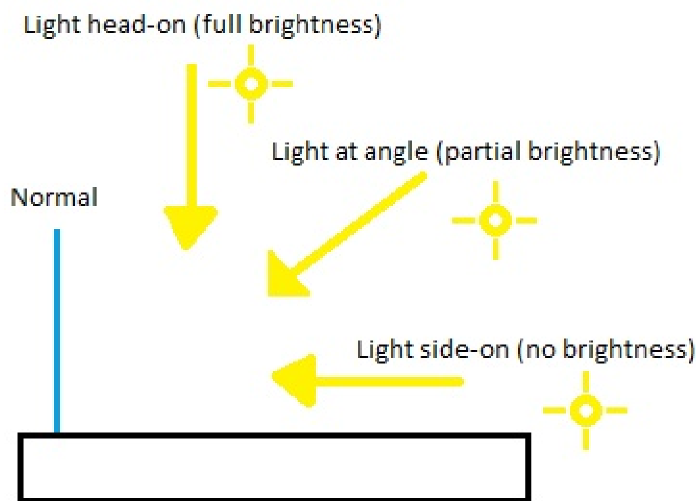


Figure 2.3: Normal vector versus lightning vector from different angles.

2.2 Mesh File Types and Data Structures

Non-procedurally created meshes are needed to store in a file. There are over 750 mesh formats according to [19]. Before constructing a new format, topological and

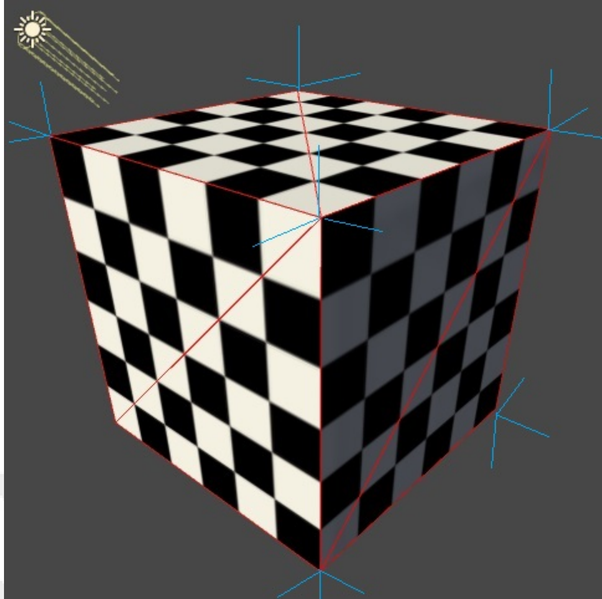


Figure 2.4: A checkered cube with normals (indicated by blue lines), shaded by the directional light.

algorithmic requirements should be considered [20]. We will not detail these but many factors such as meshes being manifold or non-manifold, closed or with boundaries, regular or irregular mesh affect the decision. We can process a mesh by editing it; changing the geometry or topology, or just render it. Whether our application needs to know adjacent faces or which vertex belongs which face or if we need to traverse on our mesh would answer our requirements. Mesh data structure types should depend on these factors. We will cover four of the most popular base structures [21] and Unity specific mesh data format.

2.2.1 Triangle Soup Data Structure

This is a naive and simple method to structure the mesh data. There are just vertices and triangles that consist of these vertices. This approach is not memory efficient, because it duplicates same vertices for each face. Other problems are traversing, searching and editing the mesh, because these operations are costly to handle for this data structure (See Algorithm 1).

2.2.2 Shared Vertex Data Structure

This approach is similar to triangle soup data structure, but it overcomes the memory problem. Shared vertices are not duplicated, instead this approach keeps them by pointers or indexing in an array. Then the triangles reference corresponding vertices. The connectivity data is still implicit in shared vertex data structure, hence, traversing the mesh is still costly. Though updating geometry is easier than triangle soup.(See Algorithm 2). This approach is more suitable for applications with solely rendering

```
// Vertex data struct
struct {
    double x,y,z;
} Vertex;
// Triangle data struct
struct {
    Vertex v1,v2,v3;
} Triangle;
// geometry and connectivity of the mesh are stored in one array
Triangle triangleArray[NUMBER_OF_TRIANGLES];
```

Algorithm 1: Triangle Soup data structure.

and simple topology edits.

```
// Vertex data struct
struct {
    double x,y,z;
} Vertex;
// Triangle data struct
struct {
    int v1,v2,v3;
} Triangle;
// geometry of the mesh
Vertex vertexArray[NUMBER_OF_VERTICES];
// connectivity of the mesh
Triangle triangleArray[NUMBER_OF_TRIANGLES];
```

Algorithm 2: Shared Vertex data structure.

2.2.3 Winged-Edge Data Structure

In triangle soup and shared vertex data structures, explicit connection information is not available. For this purpose, edge based solutions were found. Winged-edge data structure [22] bases two vertices as start and end. Then according to edge created by these vertices, the left and right faces are referenced. Also connection references are set by four other edges: left predecessor and successor, right predecessor and successor. Left predecessor and successor is used for traversing the left face, and right predecessor and successor is used for right face (See Algorithm 3). The mentioned edge between start and end vertices is the selected edge. The other four edges can be illustrated as the wings of this edge (See Figure 2.5). The order of the vertices in one triangle are clockwise; therefore, they can be traced and traversed. Each vertex and the face they correspond, need to store one reference for their adjacent edge.

Winged-edge data structure has an orientation problem respect to edges [23]. The edges are not consistent in global space, because the next edge might be oriented clockwise or counter clockwise relative to the face that is iterated. This results with an edge traversing towards the opposite direction when the left and right side faces are

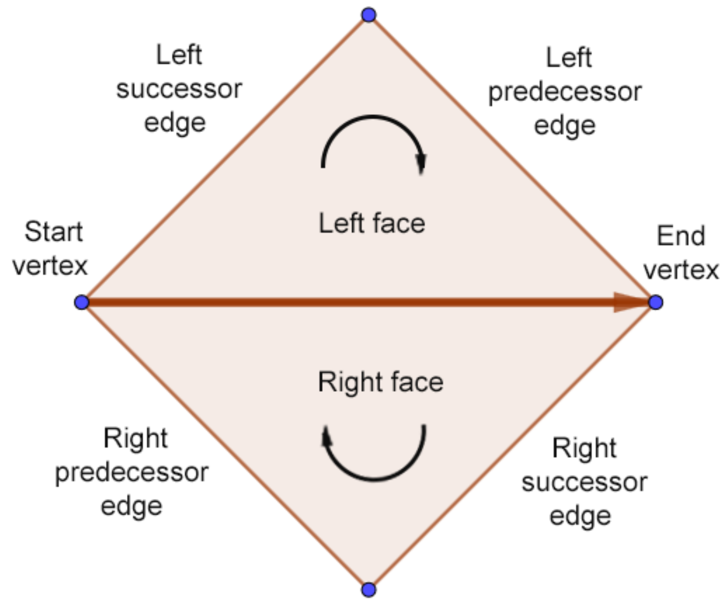


Figure 2.5: Winged edge data structure, the bold line indicates the selected edge. Left and right sides are with respect to the selected edge.

traversed; therefore, each time the orientation of an edge corresponding the traversed face must be calculated.

```
// Vertex data struct
struct {
    double x,y,z;
    WingedEdge* edge;
} Vertex;
// Triangle(face) data struct
struct {
    WingedEdge* edge;
} Triangle;
// WingedEdge data struct
struct {
    Vertex *vertexStart, *vertexEnd;
    Face *leftFace, *rightFace;
    WingedEdge *leftSuccessor, *leftPredecessor, *rightSuccessor,
     *rightPredecessor;
} WingedEdge;
```

Algorithm 3: Winged Edge data structure.

2.2.4 Half-Edge Data Structure

Half-edge data structure overcomes the orientation problem that exists in the winged-edge data structure. This method offers a solution for the inefficiency of the orientation

calculation of the edges that appears during the left and right traversals. A mesh stored by the half-edge data structure is capable of starting from an arbitrary vertex and traversing the whole mesh by following the next edges until returning the starting point.

The main difference with the winged-edge data structure is splitting edges into two half edges in opposite directions and orient both half-edges traditionally in counter clockwise order around the face [24]. Every half-edge stores a reference to its other pair half-edge, next half-edge, adjacent face and the end vertex of the whole edge it corresponds. All the half edges are aware of their left adjacent face and can find the right face by its pair half-edge's left face. Also every vertex references to its corresponding half-edge and each face references to its adjacent half-edge(See Algorithm 3, Figure 2.6).

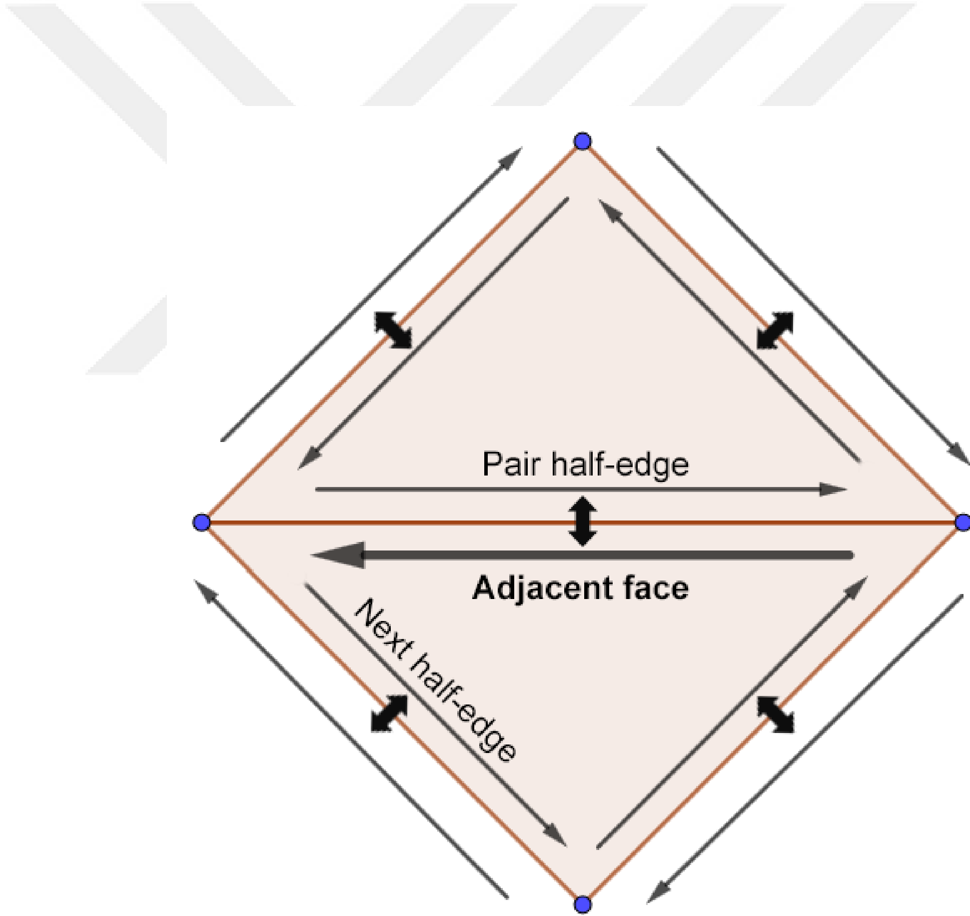


Figure 2.6: Half edge data structure, the bold line indicates selected edge.

Despite their advantages, half-edge and winged edge data structures are not used in game engines since their memory consumption is higher. Instead, they are more suitable for the editing mesh topology. Therefore, 3D modeling programs such as Autodesk's Maya uses similar approaches[25].

```
// Vertex data struct
 struct {
    double x,y,z;
    HalfEdge* edge;
} Vertex;
// Triangle(face) data struct
 struct {
    HalfEdge* edge;
} Triangle;
// HalfEdge data struct
 struct {
    Vertex *vertex;
    Face *face;
    HalfEdge *next, *pair;
} HalfEdge;
```

Algorithm 4: Half Edge data structure.

2.3 Procedural Techniques

Procedurally generated content in computer graphics have been proven to be beneficial for various reasons. Whether for creating textures, geometries or effects, procedural algorithms provide variety and detail complexity. Given a small set of parameters, a procedural system can generate a diverse set of complex outputs. This is known as data amplification [26]. Precise implementations can create tons of content without the need of possessing large data in advance. As a result, detail rich environments can be generated from small executables [27]. The outputs of procedurally generated content offer artists experimental freedom. Tweaking each parameter may result in infinite probabilities. Therefore, procedural generation is a powerful method in many areas it is used. According to [26], there are three key properties for a successful procedural generation:

1. **Abstraction:**
   The parameters, data and the behavior is abstracted into an algorithms or procedures. The user should need minimum amount of information about the geometric details and implementation.

2. **Parametric Control:**
   Parameters should answer to each specific behavior for generation. There could be various parameters for precise control of the output, which would let the user create freely. Some examples to parameters could be texture detail, particle emission rate, terrain height, water depth, buildings' number of floors, etc.

3. **Flexibility:**
   The parameters should not restrain the behaviors in terms of real world constraints. They could prompt unrealistic values and produce fantastic outputs. One can also use his/her imagination to find methods of irrelevant good looking outputs from a procedural generation algorithm developed for a different purpose.

Next we review four common methods [28] used for procedural generation in computer graphics.

2.3.1 Fractals

Most of the real world objects cannot be easily patterned by geometrical functions with high-quality output. Most of the real world objects are complex shapes, though these shapes can be processed by fractal mathematics. This method recursively duplicates self-similar copies and applies them to corresponding parts of its own. There is no limit for this recursion. Closer we look, more details are revealed.

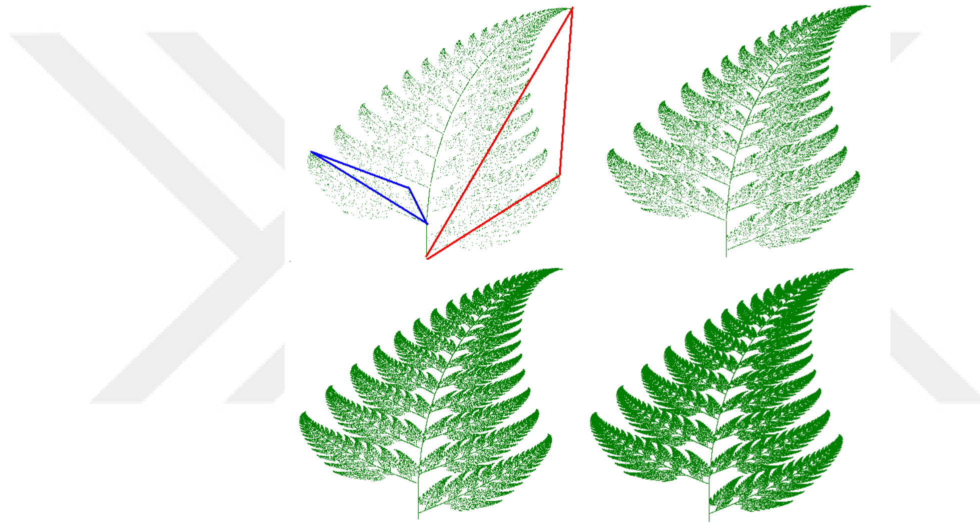


Figure 2.7: Fern is fractal structured plant in nature (taken from [1] under CC BY-SA 3.0).

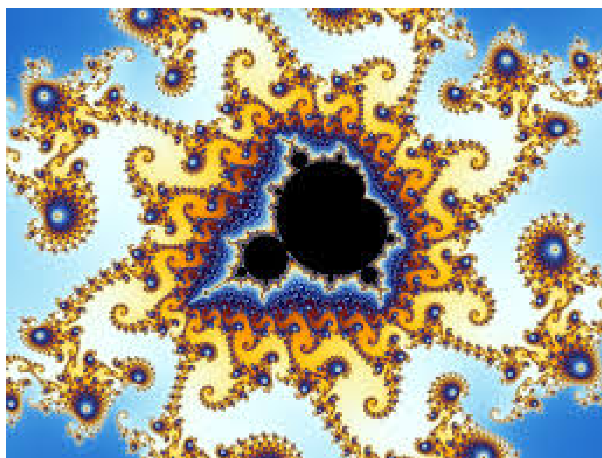


Figure 2.8: An example to Mandelbrot Set (taken from [2] under CC BY-SA 3.0).

Some of the real objects in nature are fractal structures (See Figure 2.7) Mandelbrot

[29] brought fractal approach for visualizing some of the natural and abstract shapes in computer graphics. Mandelbrot Set (See Figure 2.8) is one of the first examples produced by this technique. There are simple special recursion algorithms for fractal shapes [30]. Fractals are suitable for procedural mesh generation, because they have effective abstraction for the complex shapes that they are used for. Also they are powerful in terms of data amplification. Although very complex models can be generated by fractals, they are constrained by their self similar structure.

2.3.2 Perlin Noise

Ken Perlin introduced the method and it was first used in Tron the movie in 1982 [31]. Perlin Noise is used in various areas of computer graphics including procedural generation. Some of the examples that use Perlin Noise are generation of clouds, fire, terrain, dirt, camera effects and etc.

Perlin Noise combines layers of noise and produces the result into single coherent texture. Basically, Perlin Noise consists of three parts. First one is the noise generator. The initial noise data set is created by random values. Random values should be controllable and maintainable, so they become consistent pattern for the output.

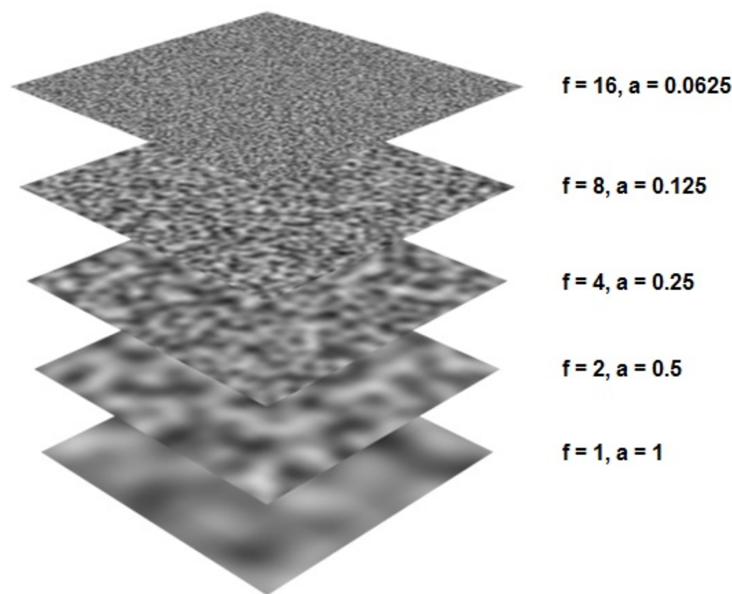


Figure 2.9: Perlin Noise outputs on different parameter values of frequency "f" and amplitude "a" (modified from [3] under GNU General Public License as published by the Free Software Foundation)

The second part is interpolation, which a function is created and fit into a curve that will match the data set generated at the beginning. By the interpolation function, initial data set will be processed and produce the output. Depending on the needs, various interpolation algorithms can be used in this part. Perlin Noise does not need a special interpolation algorithm for its interpolation function. Linear interpolation is

an example for the naive and basic algorithm, which could be used for fast and low quality outputs. Another example would be cubic interpolation, which would produce more complex outputs at increased computational cost [28].

There are two main parameters used by Perlin Noise, which are used to layer textures of noise during interpolation process; frequency and amplitude. Frequency refers to frequency of the data sampling and amplitude refers to scaling factor. Various texture layers can be generated depending the values frequency and amplitude take (See Figure 2.9). Each layer is called "octave", while amplitude and frequency ratio is referred by "persistence".

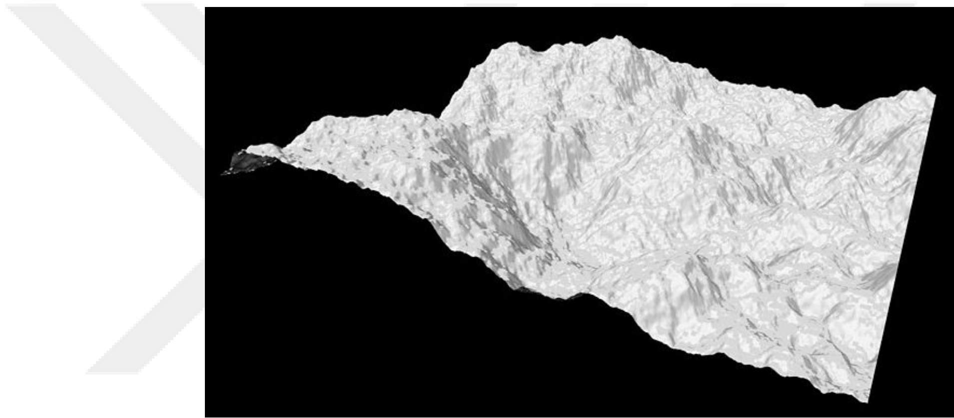


Figure 2.10: A terrain created by using Perlin Noise (taken from [4], declared for public domain use).

The last part is turbulence, which combines several noise texture layers that were generated previously for the final result. Turbulence reuses noise generation function within a simple loop that produces fractals. The function creates fault lines with gradient discontinuity at all scales which makes a fake turbulent flow. The final output produced by Perlin Noise is a natural looking procedurally generated noise texture which could be tweaked by other additional parameters on demand.

Perlin Noise suits well especially for natural environment elements. When Perlin Noise is used as a terrain noise generator, it can output natural landscapes. Combining or creating height map with the noise output produce a more organic environment [32]. Depending on tweaking parameters and persistence [31], it is possible to generate from a mountain region with slopes and valleys with a jagged terrain to smoothly detailed rich terrain with few hills and plain plateaus (See Figure: 2.10.

Hence, Perlin Noise amplifies data without the need of storing huge terrain geometry data-sets with creates a terrain with vast amount of detailing, which results with lesser storage sizes. And being suitable for various application areas make it a very popular procedural generation method.

2.3.3 Tiling

One of the oldest and most popular methods of procedural generation is tiling. The main idea is placing texture layers that will blend coherently with each other. With few tile sets it is possible to create vast amount of content. Some of the tiles can seamlessly loop with itself or other tiles, therefore, they create infinite fault-free repeatable visuals. Most of the games use this approach for texturing objects such as grass, pavements, metal, wood, etc. (See Figure: 2.11). This approach is mostly used in 2D games.

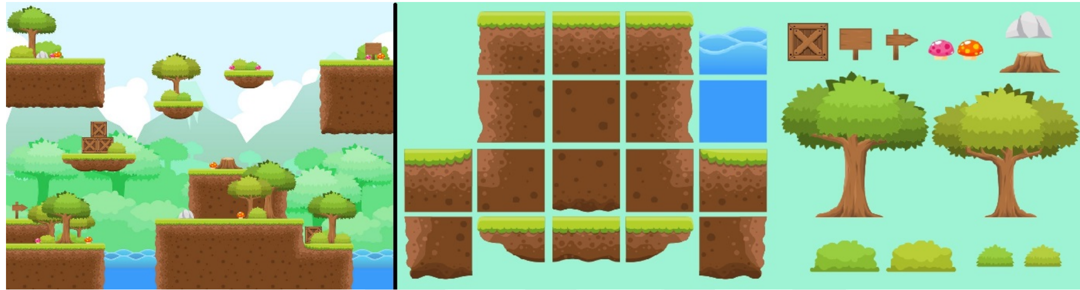


Figure 2.11: An environment created by a small tile set (taken from [5], declared for public domain use).

3D applications also use tiling. Many 3D applications benefit from tiling for texturing terrains. With parameters like height, grassy, rocky, etc. it is possible for applications to select and texture the areas as their characteristics. Neither for terrains nor 2D games, using single huge textures for painting the environment is not possible.

Advanced algorithms have been developed to benefit from tiling more. Stochastic information such as probability distribution maps to procedurally texture landscapes [33] is an example to this. Probability map stores the probability of which tile types can be placed on corresponding areas. Extra parameters can be set to specify some cases, i.e "can this tile join near that tile" or "does this area need a transitional tile before starting that area".

Tiling saves huge amount of artists' time and reduces application size and memory consumption drastically.

2.3.4 L-Systems

Lindenmayer developed L-systems as a mathematical theory for biological modeling, which is also a formal grammar. L-systems aimed to study bacteria replication and growth patterns of simple organisms [34]. However, L-systems carry the potential for the use in other fields. It has been advanced further to represent complex elements such as trees and branching objects. In 1990, Prusinkiewicz and Lindenmayer together published an article [35] about L-systems in computer graphics by illustrating development process of plants.

The main idea of L-systems is rewriting present parts by replacing them with new parts according to set of rules. At the beginning only an initial object exists, system rewrites

its parts which are defined by the rules. Initial object can be defined as a string for simplification. The replication process can be an infinite recursion, or continue until there are no more parts specified by the replication rules exist.

Figure: 2.12 is an example for a string rewriting in L-systems. **V** represents set of characters allowed, **w** is the initial state of the string, **P1** and **P2** represent the rules, whereas **n** states the recursion number. Outputs are shown at each step.

$$
\begin{aligned}
&V : \{a, b\} \\
&w : a \\
&P_1 : a \rightarrow bab \\
&P_2 : b \rightarrow a \\
\\
&n = 0 : a \\
&n = 1 : bab \\
&n = 2 : ababa \\
&n = 3 : bababababab
\end{aligned}
$$

Figure 2.12: String representation of L-system, replacing characters according to the rules.

In a similar way to the string example, plants and trees can be created (See Figure: 2.13). L-systems handle complex and organic models successfully. After a set of production steps, details can be adjusted parametrically. By setting rules combined with parameters such as angles, width, thickness, etc. vast number of outputs is possible [28].

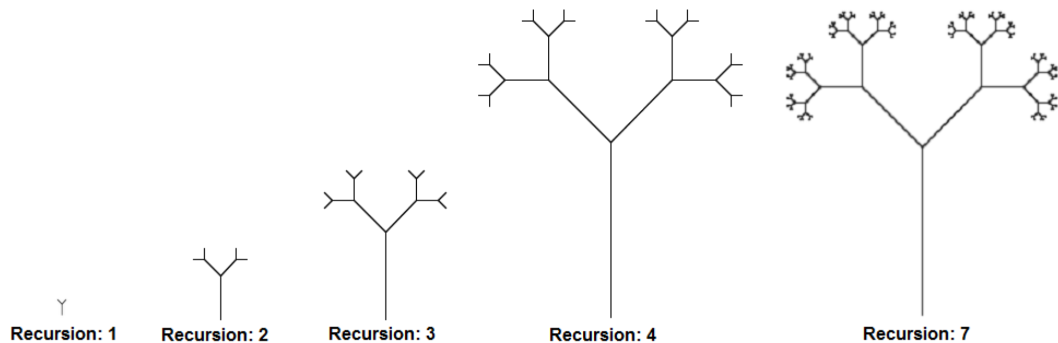


Figure 2.13: A tree created by an L-systems algorithm (taken from [6] and modified under CC BY-SA 3.0).

2.4 Road Network Generation Approaches in City Generator Systems

In this section, we will give information about three popular methods of city generation systems [36] and aerial image processing approach. Road network generation will be the main focus. We will avoid giving details about buildings and other elements of a city.

2.4.1 Grid Based Approach

In this method [37], road network is a grid made of rectangular shapes. It is a simple solution for virtual city generation. The whole city is made of blocks. The system aims real-time applications, and it can show hundreds of buildings at run-time. However, the grid based road network is homogeneous and looks artificial.

2.4.2 L-Systems Based Approach

As mentioned before in 2.3.4, L-systems are suitable for branching models. Road networks can also be formatted as a branching model. Parish and Müller used a modified L-Systems [38] method, titled "Self-sensitive L-systems" to generate cities, they named the implementation of this whole system "CityEngine". Their method takes small statistical and geographical data, then process this data to create a whole city from scratch. The user is prompted to create areas and input these areas' characteristics. CityEngine's main input list is as follows:

- Geographical Maps
  - Elevation maps
  - Land/water/vegetation maps
- Socio-statistical maps
  - Population density
  - Zone maps (residential, commercial or mixed zones)
  - Street patterns (control behavior of streets)
  - Height maps (maximal house height)

Parish and Müller created two types of roads for different purposes, referred as "highway" and "street". CityEngine connects high population density areas by highways. Inside each highly populated area, local road network is a web of streets which have access to nearby highways. Similarly in L-systems, highways are main branches whereas streets are side branches. Together these two classes are used to generate road maps.

Once the road map is created, smaller areas are marked. Inside each area geometrical calculation are made and as a results building allotments are generated. Building generation progress includes another L-system algorithm which is stochastic. All the

buildings created are tiled into superimposed and nested grid structures. Finally each grid cell facade is assigned with manual or procedural textures.

CityEngine uses two set of rules named "global goals" and "local constraints". Road segments are firstly planned by global goals, then the local areas are constructed by local constraints.

- Global goals:
  - Defining highways and streets from population data.
  - Streets should be generated due to superimposed geometric pattern.
  - Streets should be generated due to least elevation path.
- Local constraints:
  - Streets cannot be allowed to pass over certain areas, i.e over water.
  - Roads are redirected if they pass allowed area to fit into allowed areas.
  - If highways passes over disallowed areas, they are allowed to do so, i.e bridge-road over water.
  - If roads are near some junction or highway, and their end is below some distance, they are connected to the junction or highway.

CityEngine is a powerful tool for procedurally creating a decent looking virtual city in a short time. Extended versions of this work are done by [28] [39]. However, according to [40], L-Systems would be too complex when detailing area types and setting more rules to them, in terms of parameters.

2.4.3 Agent Based Approach

Proposed in the article [40], this method is an alternative to L-systems approach. The main idea of agent-based method is using developers for various areas of a city, i.e commercial, residential, industrial. etc. After that, developers support specialized agents for government buildings, squares, schools kind of detailed city component placements. This method is capable of simulating the generated city's growth after initialization as well.

The algorithm assigns each agent for their local area portion of work and they are constrained by the set of rules. Agents do not interact with each other directly, however they are aware of each other indirectly. This way, the system does not need global but only local agents. The areas are parted into rectangles and they are named "patches".

The road type in this work is *tertiary*, which means the lowest level road type in the hierarchy in urban terminology. They are mostly private roads servicing land within a certain distance, therefore roads in this method are narrow in diameter. Two types of agents are used for placing tertiary roads, "extenders" and "connectors". When a road segment is placed, within a given radius, it notifies the patches. Meanwhile, an extender agent is also sent to terrain in order to find a new area which is not served by existing road network. After a new are is found, the agent checks the distance

values from patches that it passed during the journey to the new area. While the agent is accomplishing its journey, it avoids paths with water and high elevation. Once the agent's journey finishes, the system checks if the new area is suitable for the needs, i.e if road density geometrically fits, or distance to existing intersections. If the proposition passes the test, new area is added to the city. The extender agent is sent back to roaming for more new area discoveries. However, extender agents are allowed to roam around constrained distances from existing land. Otherwise, road network would develop faster than buildings.

Connector agents are constrained to roam over the existing road network. While a connector agent moves on the road, it samples patches within the given radius. It tries to find a shorter point for reaching the selected patch from the road network by the given radius. If it cannot reach the selected patch or the distance is too long, then it proposes to connect the selected patch and its current patch by a new road segment. This new road segment proposal is tested same with the extender agent's constraints in order to be accepted.

An important derivation parameter is set for agents. Derivation defines if new road segments should be placed in a grid pattern or more organic pattern. It could also be considered as how much angle of roads allowed to rotate or continue straight. Another important set of parameters is about interconnectivity of a road network. This affects road connectors, if they can connect a patch within some distance from the start or not. As a result, road network becomes either denser with a crowded narrow loops or with dead ends and more distance between road branches.

2.4.4 Aerial Image Processing Approach

Another method for creating a road network is processing a high-quality aerial image [41] [42] [43]. These algorithms extract roads, buildings and other components like zebra lines. Then they create their models according to data they extracted. After they generate a real city, the output can be edited for different outcomes, however, they are not very suitable for creating a virtual road network from scratch.

2.5 Unity Game Engine

Unity is a powerful free game engine [44] including a detailed set of graphical features. We used Unity's 5.6.0f3 version and implemented our project in C# as Unity supports it. We followed Unity's mesh data structure and made comparisons in Section:4.1 according to it. Unity does not record adjacent edges, faces or etc. like half-edge or winged-edge data structures. Its purpose is only to render; therefore, the comparison is based on triangle and vertex data costs as in the shared vertex data structure since Unity's mesh structure is similar to it.

Unity's mesh data structure is based on shared vertex data structure. If a user is working on a procedural mesh generation, following properties might be needed to set if using Unity's Mesh class:

- bindPoses: The bind pose at each index refers to the bone with the same index
- blendShape: BlendShape count on the mesh.
- boneWeights: The bone weights of each vertex.
- bounds: The bounding volume of the mesh.
- colors: The Vertex colors of the Mesh.
- colors32: The Vertex colors of the Mesh.
- isReadable: Is Read/Write enabled?
- normals: The normals of the Mesh.
- subMeshCount: The number of sub-Meshes. Every Material has a separate triangle list.
- tangents: The tangents of the Mesh.
- uv: The base texture coordinates of the Mesh.
- uv2: The second texture coordinate set of the mesh, if present.
- uv3: The third texture coordinate set of the mesh, if present.
- uv4: The fourth texture coordinate set of the mesh, if present.
- vertexBufferCount: Number of vertex buffers present in the Mesh.
- vertices: Vertex positions.
- vertexCount: The number of vertices in the mesh.

Among these properties, we used vertices, triangles, normals, UV and subMeshCount. We mentioned about these properties on the previous subsections, except the sub-mesh. subMeshCount is used for setting the number of sub-meshes. It is possible to part a mesh into smaller sub-meshes for stating different characteristics on the main mesh. For example a wooden door with a metal knob wouldn't look realistic if both door and knob have same properties such as color and reflection strength. They need at least two materials to start looking realistic; one for the wooden door, and one for the metal knob. Each sub-mesh is represented by a group of triangles and their characteristics are specified by materials [45].

Materials utilizes shaders for how to represent pixels on the model surface appear [46]. Unity has a set of built-in shaders for most common purposes. A set of properties i.e textures, colors, illumination, etc. can be entered by the material in order to processed by the shader. We used Unity's most common shader "Standard Shader" for this work. Further information related to Standard Shader can be accessed from [47].

Unity has many road plug-ins, though most of them are not at sufficient quality. We tried EasyRoads3D [48] as one of the better ones, nonetheless, decided not to compare it to our proposed method due to considerable amount of differences. Our final decision for the comparison became comparing CityEngine's outputs(See Section: 2.4.2) as .fbx

file format[49] in Unity by importing them. The main reason for choosing CityEngine is it is based on the paper published by Müller and Parish [38]. Secondly, it is capable of producing similar outputs with our proposed method for a decent comparison.

CHAPTER 3

PROPOSED METHOD

3.1 Overview

In section 2.4 we gave a brief summary about some popular methods for procedural road network generation. These methods focus on more general aspects of city generations i.e logical distribution of networks, how major and minor roads should connect, creating special areas, building distribution etc. We, however, focus on graphical representation of road networks. Therefore, we base the proposed method on tiling (See Section:2.3.3) like many other procedural road network modeling tools [38] [48].

Our model aims to be a lightweight one. We can define lightweight in terms of memory consumption, reconstruction time, real-time render, and size. We do not keep any data sets for mesh creation. Texture usage is optimum, which is detailed in Section 3.3. All the meshes are created from scratch. Roads and junctions are created on a plane basis, only the pavements have height thickness. The user basically creates nodes and these nodes are connected with road segments. Various parameters can be tweaked for different results.

3.2 Road Topology

Road segments are the main elements of a road network. In our model, we use two main types of road segments, straight and arc. Straight road segments are created if two consequent nodes have the same angle with the previous node. In other words if normalized values of directional vectors of subsequent nodes have same values, straight road segments are created. In Figure: 3.1, this is illustrated. 1 straight road segment is consisted of 4 vertices and 2 triangles.

If subsequent nodes do not have the same normalized values of directional vectors with the previous nodes, then an arc road segment is placed. Starting from the previous road segment's end, new road segment is shaped as a circular arc segment until it is aligned with the next node. Later, the remaining distance is filled by a straight road. The arc segment is created by triangles. We define a parameter named *arcStepSize* for creating a more detailed arc shape, which increases number of triangles for a more circular look. For finding the arc angle, we solve a geometry problem which is illustrated at Figure: 3.2. P_0, P_1 and P_2 denote nodes. Between P_1 and P_2 first an arc road segment, then a

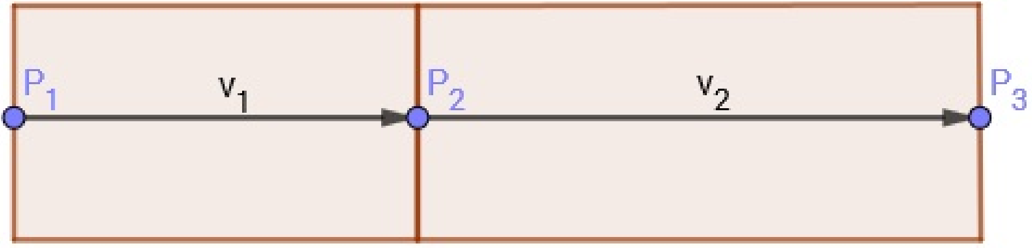


Figure 3.1: If both consequent directional normalized vector values are same between nodes, straight road segments are inserted. $P_2 - P_1 = v_1$ and $P_3 - P_2 = v_2$, if we normalize $v1$ and $v2$ both will have (1,0,0) value, hence between nodes P_1, P_2 and $P_2\, P_3$ straight road segments are placed.

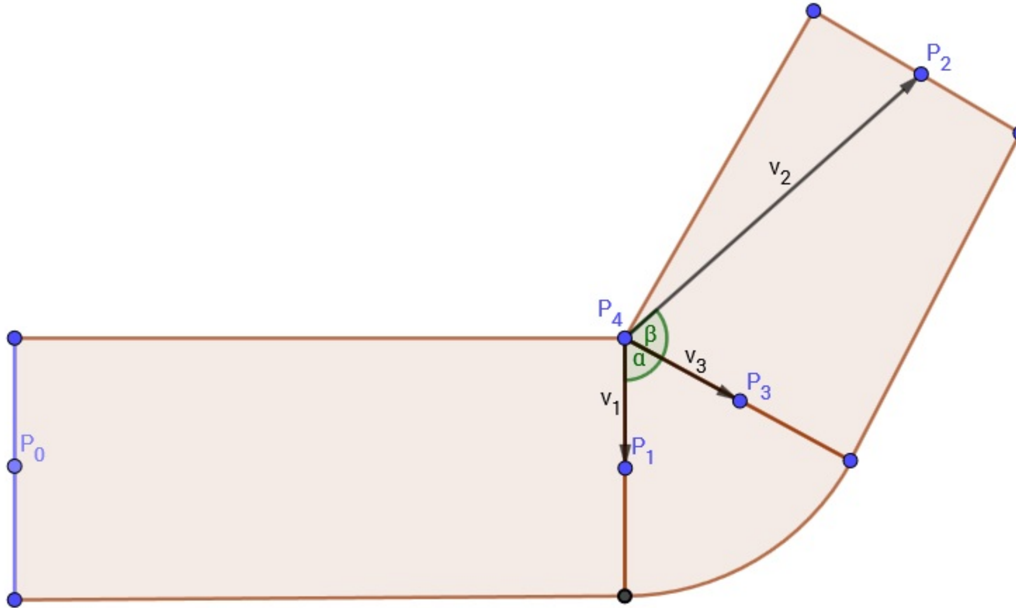


Figure 3.2: The geometry problem for finding the arc angle α.

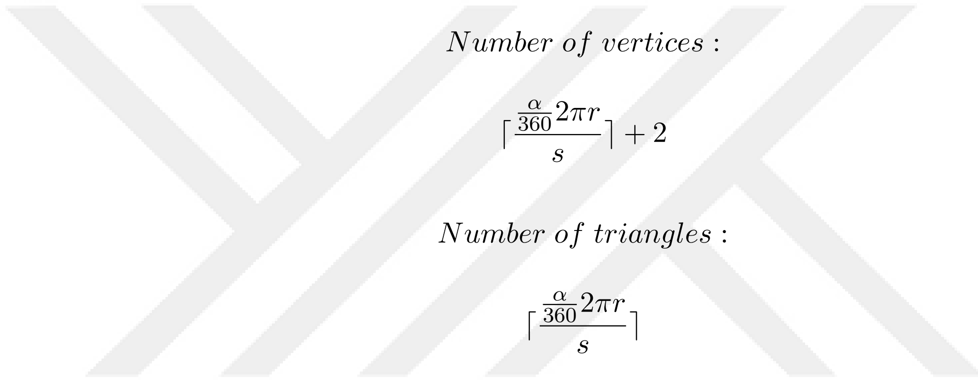
straight road segment is created, as mentioned before. Arc angle is represented by α.

$$\vec{v_2} = P_2 - P_4$$
$$|\vec{v_3}| = r$$
$$\beta = \arccos \frac{r}{|\vec{v_2}|}$$

$$R(\beta) = \begin{bmatrix} \cos\beta & -\sin\beta \\ \sin\beta & \cos\beta \end{bmatrix}$$

$$R(\beta)\vec{v_2} = \frac{|\vec{v_2}|}{r}\vec{v_3}$$
$$\Rightarrow$$
$$R(-\alpha) = \vec{v_3} \otimes \vec{v_1}$$

Once α angle is found, arc length is calculated by current radius (r), which is half of the road width. Arc length is found by $\frac{\alpha}{360}2\pi r$, then according to *arcStepSize*, how many vertices and triangles are to be placed is decided. Let s be the *arcStepSize*, the calculation for number of vertices and triangles in an arc road segment is as follows:



$$Number\ of\ vertices:$$
$$\lceil \frac{\frac{\alpha}{360}2\pi r}{s} \rceil + 2$$
$$Number\ of\ triangles:$$
$$\lceil \frac{\frac{\alpha}{360}2\pi r}{s} \rceil$$

$\lceil \frac{\frac{\alpha}{360}2\pi r}{s} \rceil$ gives the number of steps the arc segment is divided into. The constant 2 of the vertex count calculation is for the last vertex at the last arc step and the orbit vertex of the arc.

We could make a similar and smoother geometry by using quads instead of triangles, but that would increase number of vertices and triangle cost since quads need 3 vertices and 2 triangles whereas 1 triangle needs 3 vertices. Additionally, all the road segments are planar, they do not have y-axis vertices. Because the common usage is embedding asphalt on the ground, hence the asphalt does not need to be seen from the below since that would be the underground. Also the road sides are hidden by the pavements' coverage. Therefore, planar approach is the efficient solution.

Our model allows roads to be widened or narrowed by one lane at a time. During widening, the end part of road segment is increased by one lane width. In contrast, narrowing decreases the road width at the end of the road segment by one lane width (See Figure: 3.3). The user can choose whether to widen/narrow a lane from left or right. widening/narrowing lanes are actually edited straight roads. Instead of rectangles, they are trapezoids. Dashed road line is added when a road segment is widened or narrowed. The end vertices are shifted in order to have node at the center.

3.3 Road Lines

Road lines are most important aspect of our model. Users have the freedom of choosing each line type for each lane freely. Also side lines can be enabled or disabled as well.

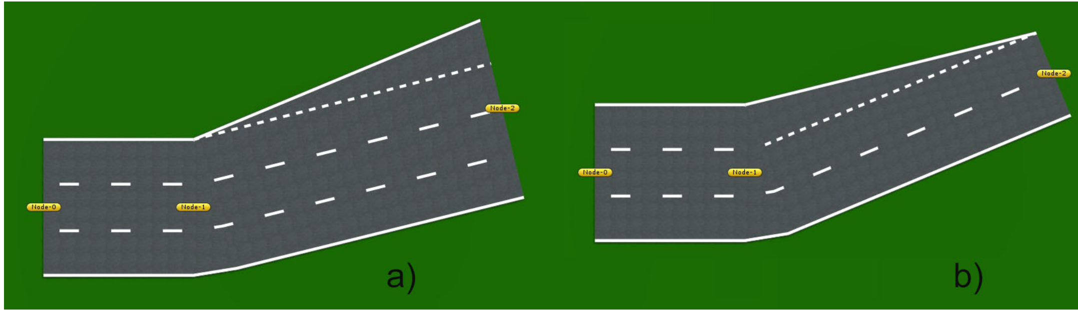


Figure 3.3: a) is a widening road, b) is a narrowing road.

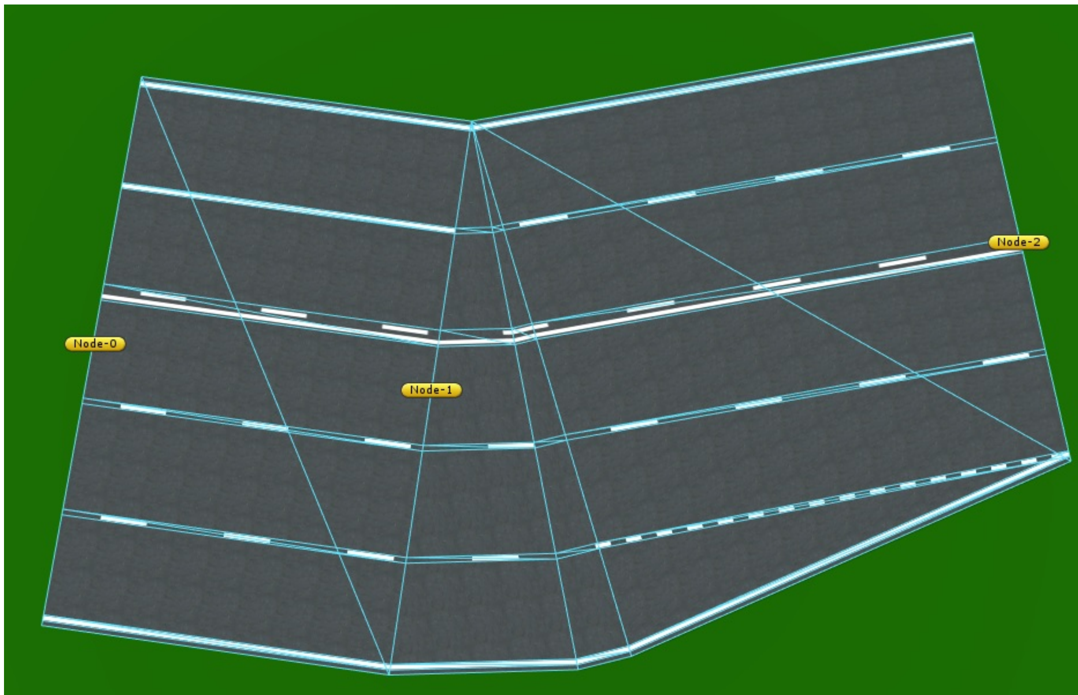


Figure 3.4: A road with various lines and their polygons.

Dashed road line which is used for widening and narrowing roads is actually in this category. The implementation method is very similar.

Traditionally, if road lines are needed, they are drawn upon the asphalt texture. There are cases for which the traditional approach is a better approach. For example if the whole road network is going to consist of two lanes with dashed road lines. However, in a realistic city, road lanes and lines often change into various combinations. For example in a four lane road first and second lane can allow overtaking while the third lane only allows overtaking from left and the fourth lane does not allow any overtaking (See Figure: 3.4). As this road continues, the lines can change. Each road-line combination is a big texture, if we use different texture for each combination, texture cost would be huge. Another downside is that, this approach is not suitable for narrowing or widening road lanes.

Our solution for this issue is defining each road line as a separate submesh. When each line is a separate submesh, they can be textured with simple small tiling textures. If our application uses 5 types of road lines, we only need 5 small tiling textures. Their combinations with respect to the number of lanes do not increase the file size or memory consumption in terms of texture costs. However, this approach requires more polygons, since every road line is a submesh instead of a texture. One road line costs 2 triangles and 4 vertices for straight road segments, whereas lines upon arc road segment cost:

$$Number\ of\ vertices:$$

$$2(\lceil \frac{\frac{\alpha}{360}2\pi r}{s} \rceil + 1)$$

$$Number\ of\ triangles:$$

$$2(\lceil \frac{\frac{\alpha}{360}2\pi r}{s} \rceil)$$

$\lceil \frac{\frac{\alpha}{360}2\pi r}{s} \rceil$ gives the number of steps the arc segment is divided into. The multiplier 2 is for doubling the triangle count to quads at each arc step. And the constant 1 of the vertex count calculation is for the last couple of vertices at the last arc step.

In terms of polygons, even a single line costs more polygons than the road segments beneath it. Though, our approach contains traditional approach as well. If a user set all the road lines "none" and uses an asphalt with road lines texture, extracting traditional approach from our model is possible. Therefore, depending on the need, the user should decide which approach would be more advantageous.

Created line sub-meshes are inserted on the road higher than the road segment by a very small value. This approach offers flexibility for all types of road lines while it is almost indistinguishable to be noticed. Another advantage of this approach is we can track and recalculate UV values for each line. Otherwise, with traditional approach, textures are scaled as they are located closer to outer area of arc segments. Also when arc segment is finished and continuing texture starting from next straight road segment does not start from the right texture position. This results in an unrealistic view.

3.4 Junctions

Roads consist straight and arc road segments. They can be connected by junctions. Junctions accept either first or last road nodes. The same junction can take first and last node of a road, resulting with a closed-loop road (See Figure: 3.6).

When a junction is created as prompted by node inputs, the junction is positioned at the average coordinates of the nodes. Using this approach, the following procedure results in a better visual quality in terms of triangulation. Junction align nodes in a clockwise order, then creates one triangle for each road and another triangle for connecting sequential roads for a complete look. Triangles for roads leave a gap for

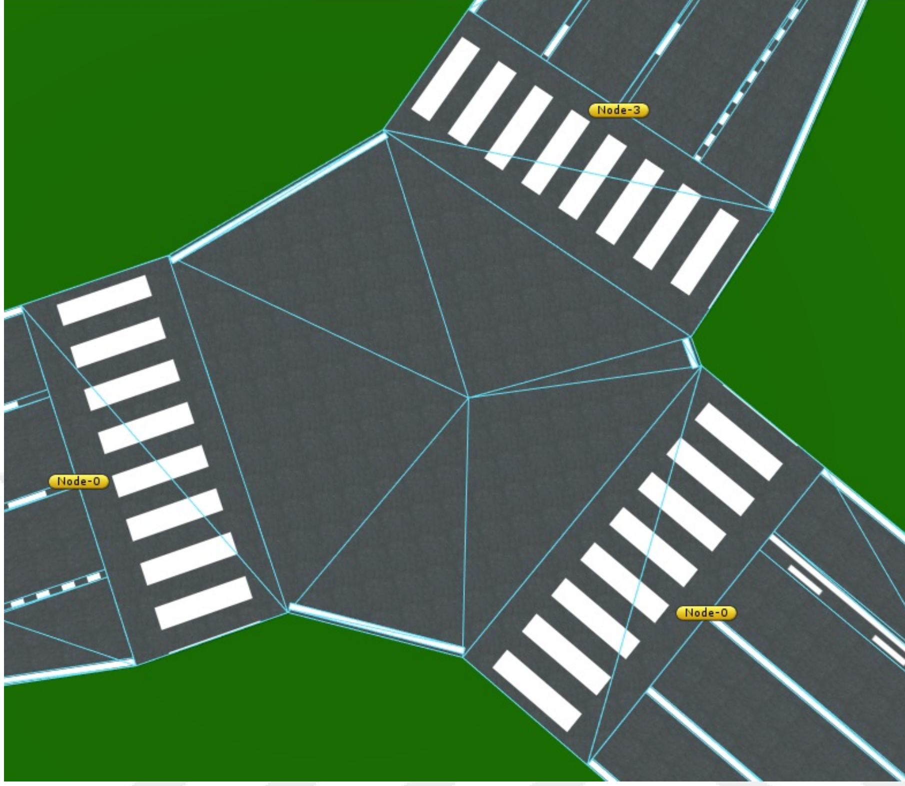


Figure 3.5: An example junction connecting 3 roads and its triangulation. 3 triangles for roads, and 3 triangles for connecting sequential roads.

crosswalks between junction and road. Triangles for connecting sequential roads have optional lines on them, for smoother visuals (See Figure: 3.5).

Let n be number of nodes junction takes as input, vertex and triangle costs for junctions are as follows:

$$Number\ of\ vertices:$$

$$14n + 1$$

$$Number\ of\ triangles:$$

$$8n$$

The multiplier 14 for the vertex count is summation of $4+4+4+2$, which respectively vertex costs for each crosswalk quad, asphalt beneath crosswalk quad, side line quad and junction's asphalt triangles' two corners. The constant 1 represents the central shared vertex for the asphalt triangles. The multiplier 8 for the triangle count is summation of $2+2+2+2$, which respectively triangle costs for each crosswalk quad, asphalt beneath crosswalk quad, side line quad and junction's asphalt triangles.

Similar to road segments, junctions are also plane basis while their lines and crosswalks are upon asphalt as separate faces.

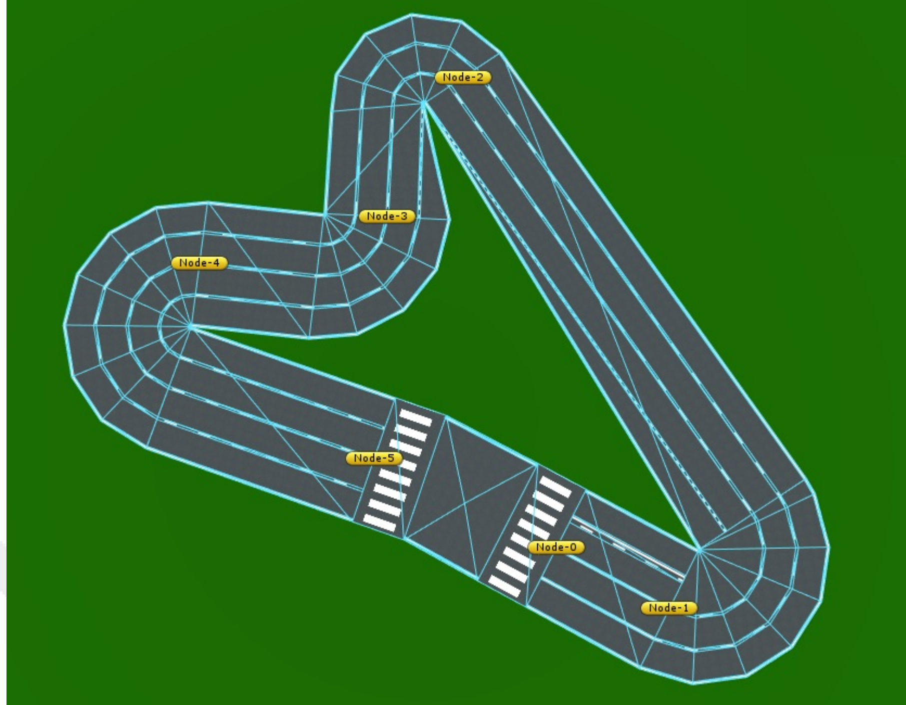


Figure 3.6: Closed-loop roads can be achieved by using junctions.

3.5 Pavements

A road network could include a large set of supporting assets. Pavements are one of the most important ones. Our model supports pavements optionally. Roads can be manageable without height, though pavements need height for realism.

We first create height face looking to road segments, secondly top face looking towards the sky, then another height face looking opposite to road. Lastly faces for enclosing the start and end parts of the pavements are inserted.

Pavements are set parallel to straight roads segments. They are distanced from road by a pavement width parameter. For arc road segment's inner-side pavements, arc road segment checks the previous and next straight road segments and cast lines parallel to them. The intersection point from these parallels becomes the meeting point for pavements. For the outer-side pavements, extruded version of the arc road segment shape is inserted.

When inserting pavements for the junctions, three parts of pavements per road connection is used. First we cast a line, which is parallel to the corresponding road connection part of the junction, distanced from the center of the corresponding road connection part of the junction with the width of the pavement. Then first and second roads cast their parallel lines distanced by their pavement width. Both first and second road pavement lines intersect with the junction's pavement road connection part line and creates points for pavement generation (See Figure: 3.8). Other points are set to crosswalk start and end points. By connecting these points, which are actually vertices, we

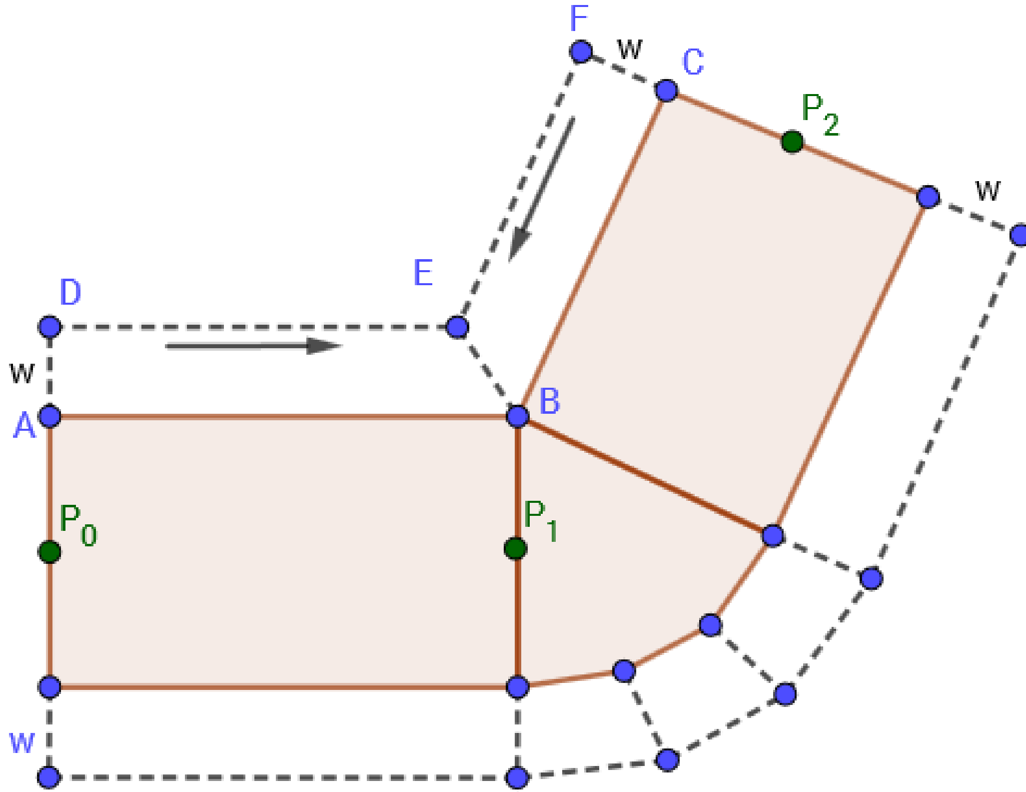


Figure 3.7: Blue points denote vertices and green points denote nodes. Two parallels from previous and next straight road segments are casted by pavement width (w) distance from A and C points to find intersection point of pavements for the inner part of arc road segment. For the outer part, outer pavement vertices are extruded from the outer arc vertices.

create our base pavement shape. We reapply the steps from the road segment pavement approach and junction pavement is created. We do not enclose the pavement for junctions, because they are most-likely the continuation of road pavements; therefore, it is not necessary.

The vertex and triangle cost for both sides of a straight segment is 24 vertices and 12 triangles. The cost for arc road segments and junction pavements is as follows:

$$Number\ of\ vertices\ for\ arc\ road\ pavement:$$

$$6\left(\left\lceil\frac{\frac{\alpha}{360}2\pi r}{s}\right\rceil+1\right)$$

$$Number\ of\ triangles\ for\ arc\ road\ pavement:$$

$$6\left(\left\lceil\frac{\frac{\alpha}{360}2\pi r}{s}\right\rceil\right)$$

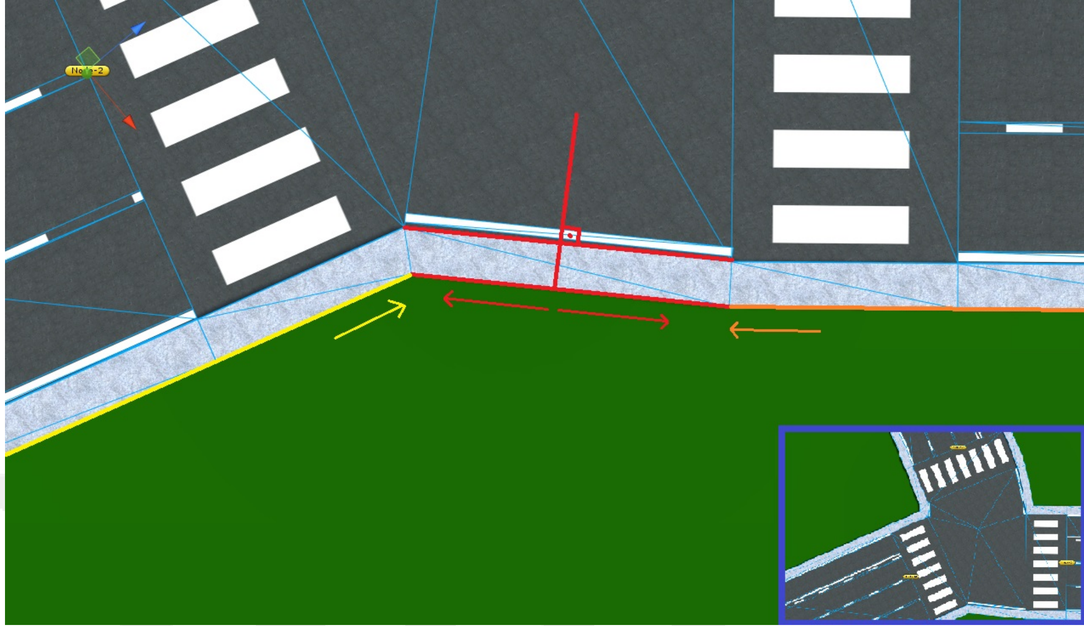


Figure 3.8: From the center of the junction road segment line, a parallel distanced by the pavement width is casted. Also two parallels from road sides distanced by pavement width are casted. The intersection points of these lines construct the base points of a junction pavements. The same procedure is repeated in a clockwise order for each road connection part in the junction.

The calculation is same with the arc road line calculation. The only exception is instead of multiplying with 2, here we multiply by 6. Because pavements have 3 faces whereas lines have 1.

$$Number\ of\ vertices\ for\ junction\ pavement:$$

$$36(\lceil \frac{\frac{\alpha}{360}2\pi r}{s} \rceil)$$

$$Number\ of\ triangles\ for\ junction\ pavement:$$

$$18(\lceil \frac{\frac{\alpha}{360}2\pi r}{s} \rceil)$$

3.6 Nodes

Lane width, pavement height and width, arc step-size kind of details are general settings for each road. But nodes are used for road segment changes and setting the road path. Nodes are always centered at the beginning or end of the road segments. This is done by modifying the road alignment. For example if we set a road segment with two nodes on the same axis, and the road has two lanes at the beginning but finishes with

three lanes (the new lane is added from left or right side of the road segment), then the road segment does not become parallel with the nodes' axis. Because the road segment's beginning part is centralized by the beginning node' two lanes width and the finish part is centralized by the finish node's three lanes width from left or right. Therefore, the road segment is rotated slightly for having both nodes at the centers of its beginning and finish parts. We chose this approach because it is easier for visually controlling road paths.

The user sets road segment's beginning lane count from the first node of the entire road. As the road continues, the road segments consist the road are always between two sequential nodes. So the previous road segment's second node is the current road segment's first node. Except the first node of the entire road, all the road segment features are set from the corresponding secondary nodes of the road segments. The user can choose the road segment's behavior from the corresponding secondary nodes such as narrowing/widening right or left. Also road line types are set from the corresponding secondary nodes.

3.7 Pseudo Code

Algorithm 5 describes the whole procedure for our proposed method:

while *all* roads are not checked **do**
  get all node positions in order
  while *all* nodes in current road are not checked **do**
    if *current* and next node have same direction with each other **then**
      initialize straight road segment
    else
      initialize arc road segment
      calculate arc road angle
      initialize straight road segment according to the arc road angle
    end if
    next node ← current node
  end while
  number of vertices ← calculate number of vertices for road
  number of triangles ← calculate number of triangles for road
  while $i <$ number of road lines **do**
    number of vertices ← number of vertices + needed vertices for line i
    number of triangles ← number of triangles + needed triangles for line i
  end while
  if *pavements* enabled **then**
    number of vertices ← number of vertices + needed vertices for pavements
    number of triangles ← number of triangles + needed triangles for pavements
  end if
  while *all* road segments in current road not checked **do**
    if *road* segment is straight **then**
      set straight road segment vertices and UV
      if *road* segment narrowing right **then**
        edit current road segment vertices and UV

decrease total road length for next segment from right
else {*road* segment narrowing left}
edit current road segment vertices and UV
decrease total road length for next segment from left
else {*road* segment widening right}
edit current road segment vertices and UV
increase total road length for next segment from right
else {*road* segment widening left}
edit current road segment vertices and UV
increase total road length for next segment from left
end if
else
set arc road segment vertices and UV
end if
next node becomes current node
end while
create road triangles and set normals
create pavement vertices, UV, normals and triangles
while $i <$ number of road lines **do**
create line vertices, UV, normals and triangles for line i
end while
set road, line and pavement materials
update mesh data
while *all* junctions are not checked **do**
get node inputs from roads for current junction
set center point of junction
sort nodes according to center point in clockwise order
number of vertices of junction j $\leftarrow$ calculate number of vertices for center, crosswalk, lines, and pavements
number of triangles of junction j $\leftarrow$ calculate number of triangles for center, crosswalk, lines, and pavements
create vertices, UV, normals and triangles for center area
create vertices, UV, normals and triangles for crosswalks
create vertices, UV, normals and triangles for lines
create vertices, UV, normals and triangles for pavements
set center, crosswalk, line and pavement materials
update mesh data for current junction
end while
end while

Algorithm 5: Pseudo code for the whole procedure

CHAPTER 4

RESULTS AND DISCUSSION

4.1 Comparison with CityEngine 2017

CityEngine's (See Section:2.4.2) commercially available, up-to-date 2017 version [50] is currently live. We chose to experiment and compare the proposed method with City Engine 2017 since it is a very popular and modern tool. Our aim is to provide proposals for advancing some of City Engine's features. We also take example of some of its features in our model. The main aim is to create a graphical road model which would perform well and have a balanced graphical quality at the same time.

CityEngine also uses nodes for controlling roads as our tool. After creating a road network in CityEngine 2017, we prompt CityEngine to export model as a .fbx file [49]. The file is transferred to Unity, since our implementation is also in Unity and it does not bring any limitations for comparison. We compare CityEngine built-in model with our procedural model in terms of texture usage, vertex and triangle count, which are basically the main components for performance as mentioned previously.

All the comparisons and data gathered in this paper are done by the configuration provided below:

- **Operating System**: Microsoft Windows 8.1
- **CPU**: Intel Core i7-3630QM
- **GPU**: NVidia GeForce GTX 670M
- **RAM**: 16GB DDR3

We created three almost identical road networks by CityEngine and our proposed method. CityEngine allows vast amount of customization options. We were able to tweak its parameters to provide similar level of detail as our model. However, CityEngine has a difference with our proposed model. The difference is the pavement style, which consists of two parts: pavement stones and sidewalks (See Figure: 4.1). Our model allows only for sidewalks as a complete pavement (See Figure: 4.2). Pavement stones add extra graphical quality and significant cost for a fair comparison, though CityEngine does not allow the user to disable them. Therefore, for a fair comparison we calculated how many vertices and triangles would CityEngine have if they

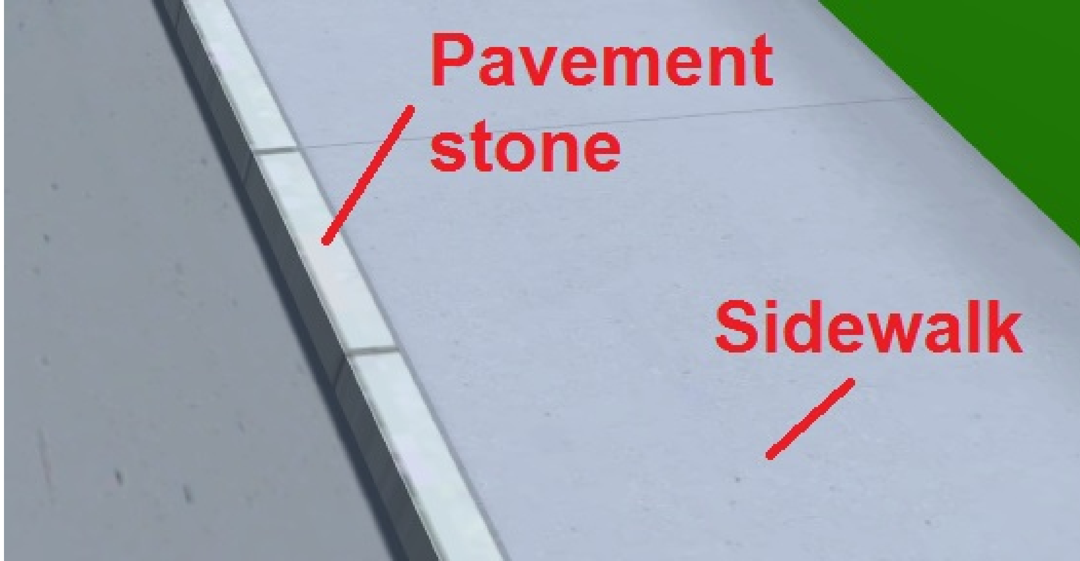


Figure 4.1: CityEngine pavement model has 2 parts of faces, one for pavement stones, another for sidewalk. But our model only uses sidewalks; therefore, we calculated how the results would be if CityEngine also had pavements similar to our pavements and used these results for fair comparison.

Figure 4.2: Our pavements only have the sidewalk part.

implemented pavement part as the proposed method. We tracked down number of vertices and triangles CityEngine uses for pavement stones and sidewalks each. We subtracted pavement stone vertex and triangle count from total pavement vertex and triangle count. CityEngine spends 1 face for its sidewalks only for top side. The proposed method uses 3 faces as explained in Section 3.5. Hence, it would be exactly 2 times more than CityEngine. Unity shows us the exact numbers of vertex and triangle counts for all models. The results for vertices and triangle count for this sample

road network models are shown in Table 4.1, Table 4.2 and Table 4.3. The corrected differences are displayed in the tables, under the section "CityEngine - Optimized".

Table 4.1: Vertex and triangle counts for the road network models with few junctions with many straight and arc road segments. Both of the models are shown in Figure: 4.3.

		CityEngine	CityEngine-Optimized	Proposed Method
Roads	Vertices	1078	1078	255
	Triangles	564	564	173
Pavements	Vertices	4832	2496	1830
	Triangles	2660	1272	1170
Lines	Vertices	572	572	412
	Triangles	316	316	324
Junctions	Vertices	207	207	171
	Triangles	109	109	96
Total	Vertices	6689	4353	2668
	Triangles	3649	2261	1763

Table 4.2: Vertex and triangle counts for the road network models with few arc road segments but many straight road segments and junctions. Both of the models are shown in Figure: 4.4

		CityEngine	CityEngine-Optimized	Proposed Method
Roads	Vertices	788	788	245
	Triangles	564	564	157
Pavements	Vertices	7936	3546	3414
	Triangles	4432	1788	1914
Lines	Vertices	276	276	432
	Triangles	184	184	318
Junctions	Vertices	791	791	772
	Triangles	430	430	432
Total	Vertices	9791	5401	4863
	Triangles	5610	2966	2821

We did not try to create a realistic road network model in terms of city architecture standards. However, we tried to compare road networks with different characteristics by including various common features of the proposed method and CityEngine. The sample road network has widening and shrinking road segments, different road lines, three junctions with different number of roads intersecting and different number of road lanes for roads.

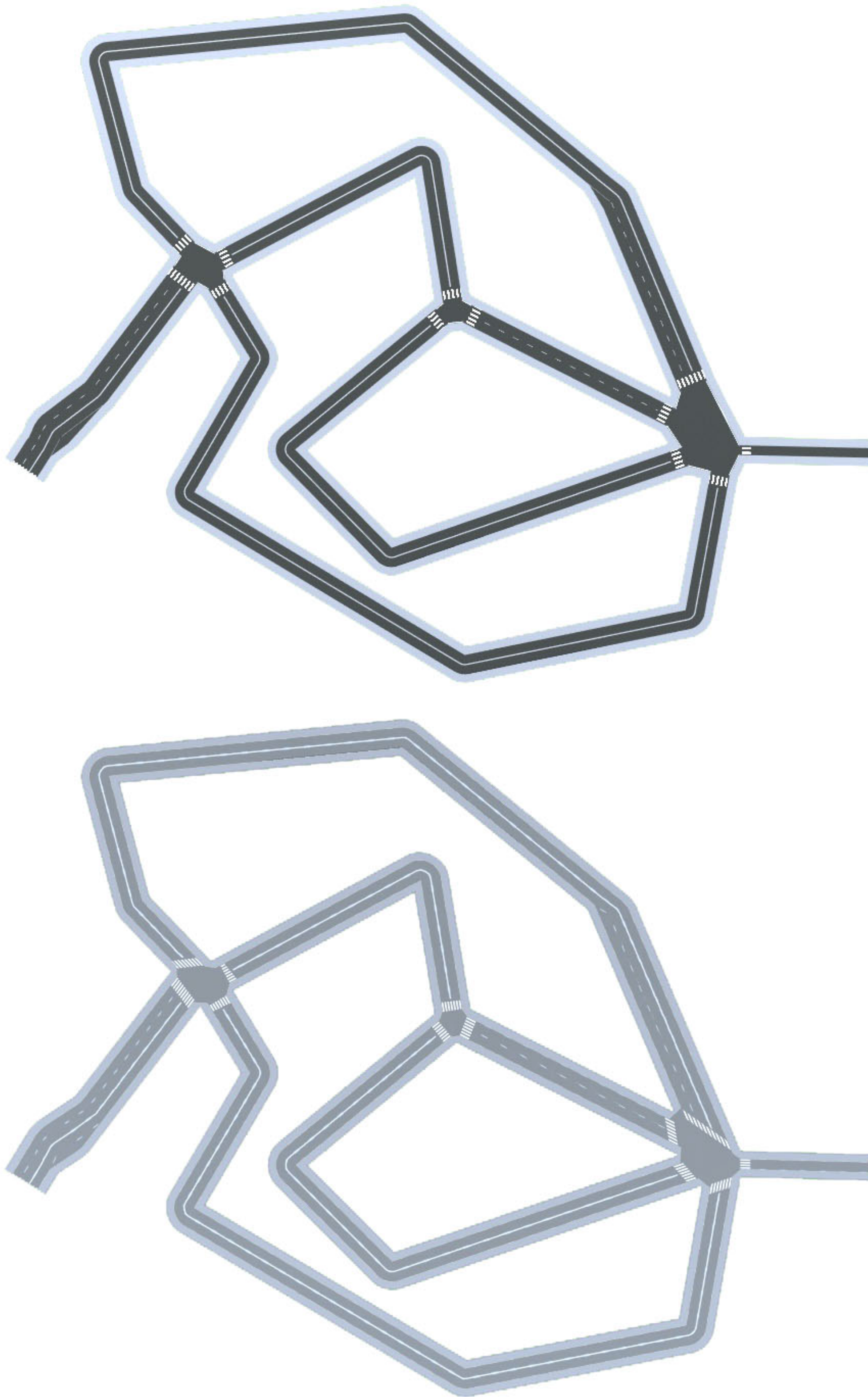


Figure 4.3: Two almost identical road networks with few junctions and many straight and arc road segments. Vertex and triangle counts are displayed in Table: 4.1. The proposed method's output is at the top, while CityEngine's at the bottom

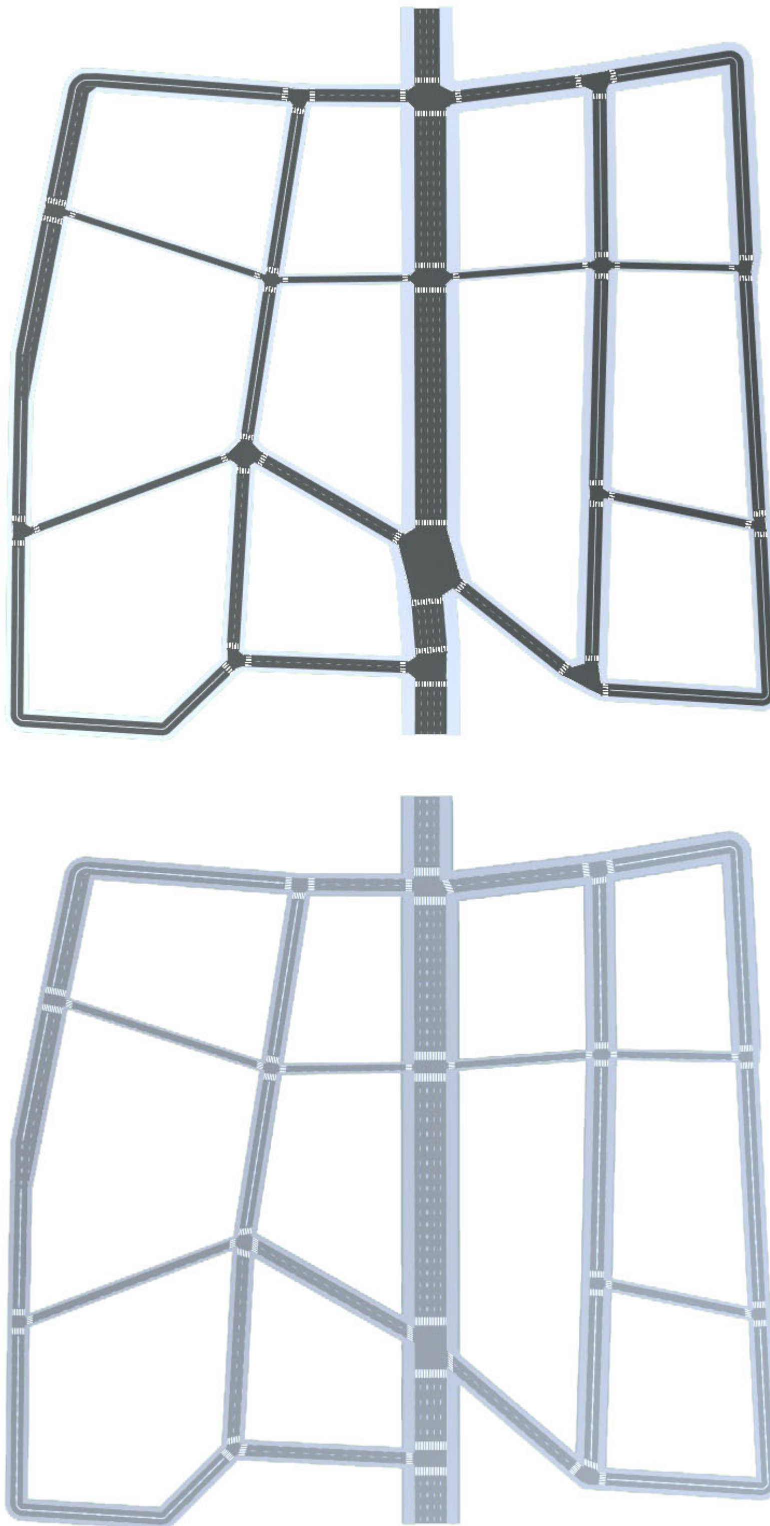


Figure 4.4: Two almost identical road networks with few arc road segments but many straight road segments and junctions. Vertex and triangle counts are displayed in Table: 4.2. The proposed method's output is at the top, while CityEngine's at the bottom

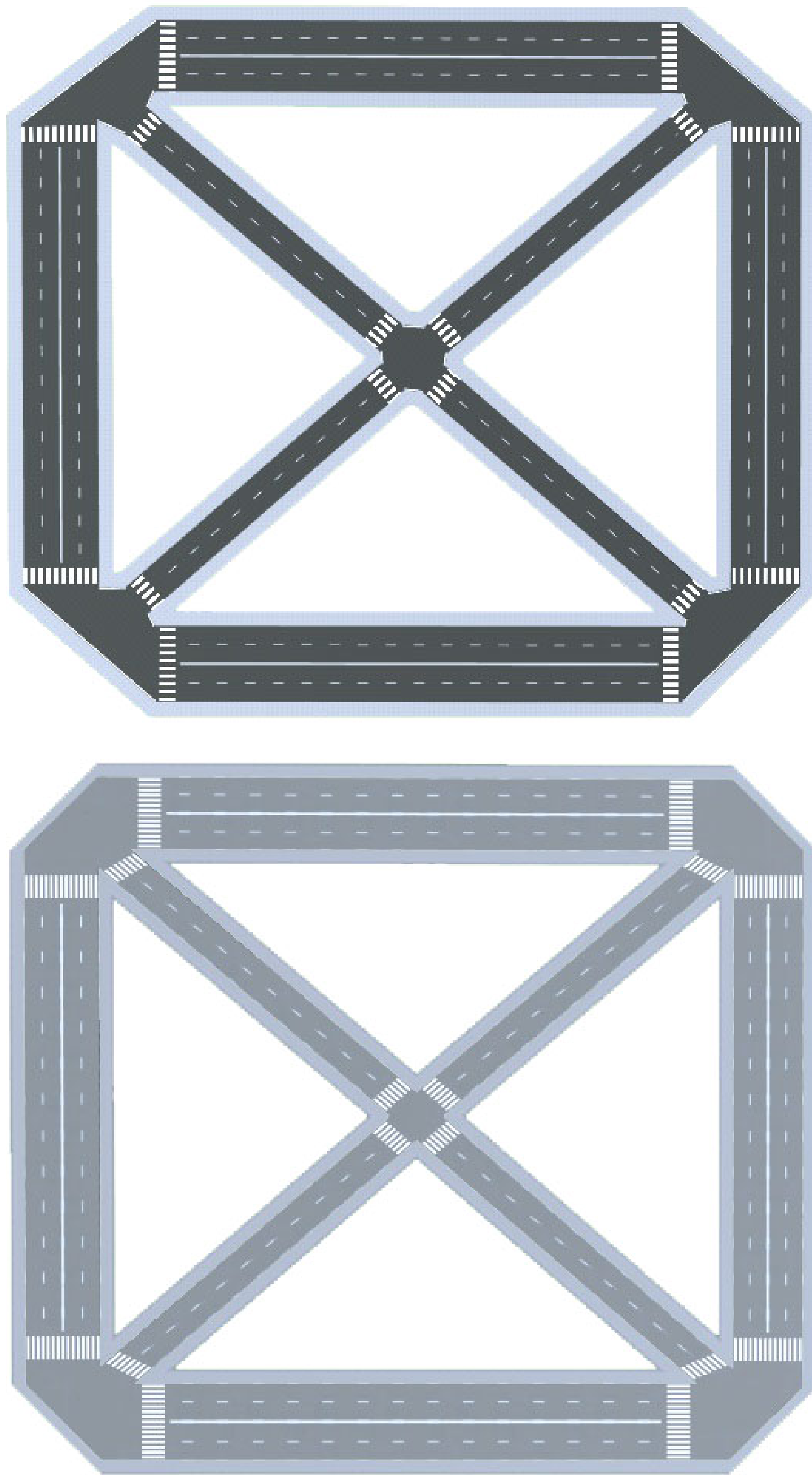


Figure 4.5: Two almost identical road networks with no arc road segments but straight road segments and junctions. Vertex and triangle counts are displayed in Table: 4.3. The proposed method's output is at the top, while CityEngine's at the bottom

Table 4.3: Vertex and triangle counts for the road network models with no arc road segments but straight road segments and junctions. Both of the models are shown in Figure: 4.5

		CityEngine	CityEngine-Optimized	Proposed Method
Roads	Vertices	120	120	32
	Triangles	96	96	16
Pavements	Vertices	1496	648	768
	Triangles	816	336	384
Lines	Vertices	24	24	64
	Triangles	16	16	32
Junctions	Vertices	220	220	229
	Triangles	114	114	128
Total	Vertices	1860	1012	1093
	Triangles	1042	562	560

Table 4.4: Vertex and triangle counts for the sample road segment models from Figure: 4.6 and Figure: 4.7.

		CityEngine	Proposed Method
Straight Road Segment	Vertices	24	32
	Triangles	18	16
Arc Road Segment	Vertices	692	331
	Triangles	357	315

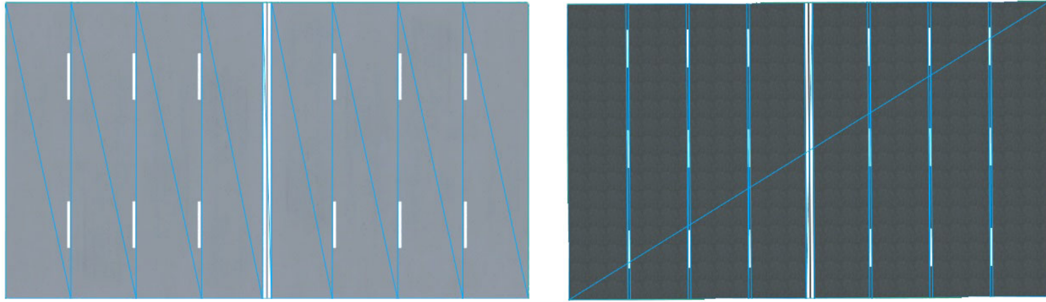


Figure 4.6: One almost identical straight road segment from CityEngine (on left) and the proposed method (on right). The polygon count difference comes from our line based approach and CityEngine's lane based approach. The triangles consist the road segments are shown in blue.

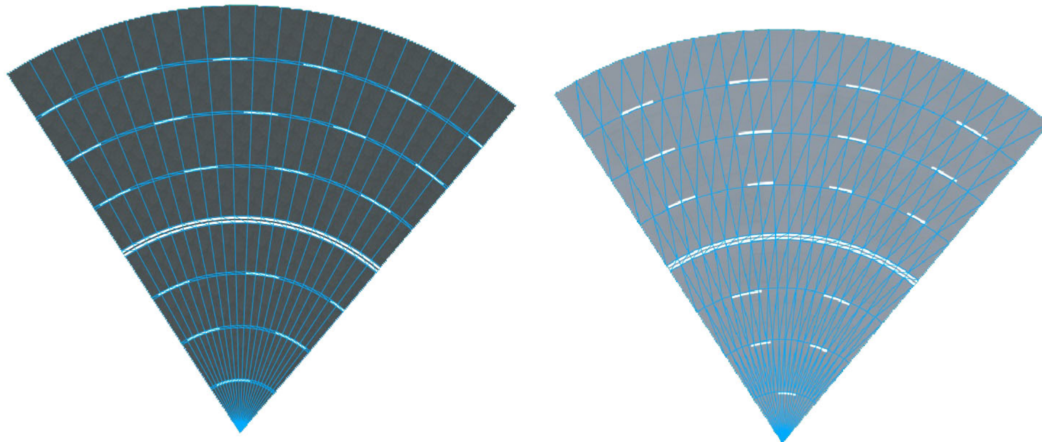


Figure 4.7: One almost identical arc road segment from CityEngine (on left) and the proposed method (on right). Again the polygon count difference comes from our line based approach and CityEngine's lane based approach. The triangles consist the road segments are shown in blue.

4.2 Discussion About Comparison with CityEngine 2017

Based on the results according to the tables provided, the results do not follow a consistent ratio in total. In general, junction results are close to each other in both methods. For pavements, in Table 4.1 the proposed method has a significantly better result, in Table 4.3 CityEngine results slightly better whereas in Table 4.2, on the contrary, the proposed method results slightly better. Unlike the roads, CityEngine and the proposed method use different approaches for creating their pavements and junctions. Although the similarities, their visual outputs are different from some aspects. Therefore, a direct comparison may not lead to a correct conclusion between these.

However, the roads and lines are more similarly constructed and the visual outputs are almost identical. In the tables, roads and lines are shown in different sections for emphasizing the two method's differences, though they should be considered as one. Hence the when we considered summed results of the roads and the lines they will give the more accurate result for road segments. The results for single straight road segment and single arc road segment shown in Table 4.4 are the summation of lines and roads. The CityEngine's straight road segment has less vertices because adjacent lanes can share vertices, though it costs 2 triangles more. The 2 extra triangles would not exist as well if the middle line was a dashed line. The summation results for road segments in the other tables which have different and more complicated road segments become:

- **Road network from Table 4.1**:
  - CityEngine: 1650 vertices, 880 triangles
  - The Proposed Method: 667 vertices, 497 triangles
- **Road network from Table 4.2**:
  - CityEngine: 1064 vertices, 748 triangles
  - The Proposed Method: 677 vertices, 475 triangles
- **Road network from Table 4.3**:
  - CityEngine: 144 vertices, 112 triangles
  - The Proposed Method: 96 vertices, 48 triangles

All the results for the road segments by the proposed method use significantly less vertices and triangles. However, the simple road segments' results in Table 4.4 and the other tables' result seem inconsistent since they result very different. The reasons for this and the other aspects for the results in the tables are itemized and compared in detail:

- **Lane Division**: If a street has more than 1 lane, CityEngine divides its whole road segment by the number of road lanes the whole road segment has. Therefore, the number of polygons is multiplied by the number of lanes. CityEngine also offers a middle line feature which divides the road into two and this line costs the same as the adjacent road lane. It is similar to our road line approach. Though, this cost is extra for CityEngine's lane approach, because the road lines are textured onto road lanes, but the middle line becomes a separate set of polygons instead of being textured onto a lane. When we compare the results for the two almost identical arc road segments, Table: 4.4 shows that our line based approach uses less polygons than CityEngine's approach. The most important reason for this difference is that CityEngine does not use triangles if the minimum arc angle detail is set to 0. Instead, it creates quads and places two vertices at the same coordinate position. Therefore, even though the arc segments visually look like made of triangles, in the background they are actually quads with two vertices at the same coordinate position. However, in Table 4.3 there are only

straight road segments and junctions yet the proposed method has less vertices and triangles. Compared to Table 4.4 the results should be closer. The reason for that is CityEngine includes polygons at the start and end areas of straight road segments connected to a junction. These polygons are added in case the user enables some extra elements such as stop line for a crosswalk. But they are not erased when the user sets no lines.

- **Road Expansion Detail**: Even with the least detailed road network generation options, CityEngine uses too many polygons on road sides for widening and narrowing segments. If lane division vertical detailing, widening/narrowing division is horizontal detailing, in other words it is similar to dividing a rectangle into pieces vertically and horizontally like rows and columns in a table. Hence, CityEngine divides road lanes horizontally into more polygons. When the number of vertical and horizontal divisions are multiplied they result with the total number of polygons. Since we do not divide roads into lanes, and our roads have constant 4 vertices and 2 triangles for widening/shrinking roads, the polygon cost difference between the proposed method and CityEngine becomes significantly more for this issue.

- **Road Lines**: The main novelty in the proposed method is road lines. Because we separated them from the road segments, this approach brings optimization, flexibility and easier maintenance. However the results show our line vertex and triangle count is close to CityEngine's. The reason is we use separated lines from roads. Therefore, road and line vertex and triangle count should be examined together for the evaluation. As it can be seen, the proposed method's roads' and lines' total vertex and triangle counts are significantly fewer in the results. However, these results are not only due to our road line approach. A significant polygon count is due to the road expansion detail and lane division. Our line method would perform worse under specific circumstances in terms of vertex and triangle count. If single straight road segments with 8 lanes divided by 6 dashed and 1 middle lines are created by both methods, CityEngine would use 24 vertices and 18 triangles whereas the proposed method would use 32 vertices and 16 triangles as in Table 4.4. However the proposed method would perform better with more complex road line scenarios with a road network consists of various road segments comprising of different number of road lanes as in the sample road networks. A more consistent inference could be achieved if, theoretically, we assume CityEngine always divide its road segments by the number of road lanes and its road expansion detail is same with the proposed method. If CityEngine follows this approach, it can use textures for each corresponding lane; therefore, it can include different types of road lines. As a result: for a 4 lane road segment consisted of 4 independent lanes versus the proposed method consisted of 1 whole road and 3 line segment would have the same polygon cost for a straight road segment. Because all these segments are quads and at the end the result is 4 quads for the both. In general it would decrease the cost and increase the texture quality. Additionally CityEngine could still use their road expansion details. This method is similar to the proposed method, but it is more suitable for CityEngine's lane based methods. However, this method or CityEngine's method cannot provide the freedom the proposed method offers. Because having an asphalt at the bottom and placing lines separately on it eliminates texturing problems for many specific cases as mentioned before.

- **Textures**: CityEngine assumes all the road lanes would take dashed lines. It only allows double straight lines as a middle line and this parts the whole road at least into two for straight road segments, and equal to number of lanes for widening/shrinking road segments. However, in reality multiple lane lines might have different combinations and different settings, i.e overcoming from right or left road lane lines. Due to this approach, CityEngine uses a 4 dashed lines over an asphalt texture 4.8, the aim is: if the road have more than 1 lane it will continue to use this texture until there are 4 lanes on one side. If there are not 4 lanes, a part of the texture becomes unused. Hence, the resolution of the texture for covered area decreases, which is not an efficient method. The proposed method uses a single asphalt texture, so it can repeat seamlessly without a problem according to UV mapping scale we set. We are able to do it, because we create road lines separated from the asphalt itself. Our road lines also become size efficient, because we can snap the texture exactly as the polygonal area. Lastly, using solely asphalt texture enables us to use the same texture for roads and junctions at the same time, whereas CityEngine has to use 2 textures: asphalt with lines for roads and sole asphalt for junctions.

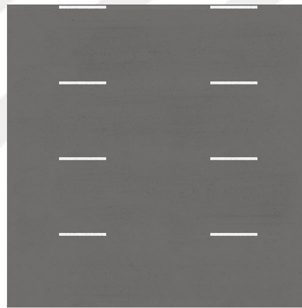


Figure 4.8: CityEngine uses a fixed 4 dashed line road lanes texture no matter the number of road lanes exist.

- **Pavements**: Pavements are the most costly parts for the both approaches. Therefore optimizing them is vital. CityEngine pavements are costly due to its pavement detail settings. There are many invisible faces are created. Even with our optimization applied on CityEngine for pavements, it is still costly. Though the reason is CityEngine places the pavements according to its road detail. We mentioned about road details of expansion areas; therefore, as the road includes more vertical details, the pavements also become more costly.

- **Junctions**: Both methods create simple junctions. However, CityEngine's road connection areas are uneven whereas the proposed method uses rectangular ends. Therefore, CityEngine's connection areas, which are crosswalks in this case, become rhomboid instead of rectangular. Placing rectangular texture onto a rhomboid area creates a faulty visual.

4.3 Large Network Generation

For showing the proposed method is capable of generating large networks procedurally, we created a simple input generator. This input generator first creates abstract nodes by the given distances on x and y axes as a firm grid. Secondly, the actual nodes are created by a random distance range. The abstract nodes represent junctions, and the actual nodes are both connection points and start or end positions of roads, which connect junctions to each other. Therefore, there could be at least two actual nodes of a junction if it is a corner junction, three, if the junction is situated on an edge, and four, if the junction is positioned neither on a corner nor edge. After connecting these junctions, the connector roads have two nodes. Then by a random distance range, random number of in-between nodes are added in the connector roads. As a result, due to the small direction differences, instead of having just straight road segments, some curved road segments are also created. Lastly, randomly ranged number of lanes are set for the network along with their road lines.

As the random distance parameters change, the grid network becomes more dispersed. In Figure: 4.9, we presented four samples. The distance between junctions are 60 units, whereas roads have 2 to 4 lanes with dashed lines and each road has 1 to 3 road segments. For a, b, c and d, respectively, the random distance radius for shifting junction distances are 0, 5, 10 and 15 units. The random distance radius for shifting in-between node distances are 0, 1, 2 and 3 units. The triangle counts are 94866, 100712, 102151 and 104772. As the grid network disperse more, the arc road segments have more triangles. Therefore, the triangle cost increases.

We measured generation time, memory consumption during the generation; triangle, road and junction counts for the increasing grid sizes and showed as graphs. Grid size is calculated by multiplying number of columns (X) and rows (Z). For more consistent results, random distance parameters are used with the same values. Also all the roads are consist of 3 lanes with 2 dashed lines, 3 straight and 2 arc segments. The generation time is both for the grid input creation and the output combined. Time, memory consumption; triangle, road and junction counts increase linearly as the grid size becomes larger. Therefore, the complexity of the whole generation process is linearly bound to input cost. The results for grid size 4 (2x2) to 1225 (35x35) are shown in Figures:4.10,4.11,4.12,4.13. After the grid size 35x35, the present configuration almost reaches its limits.

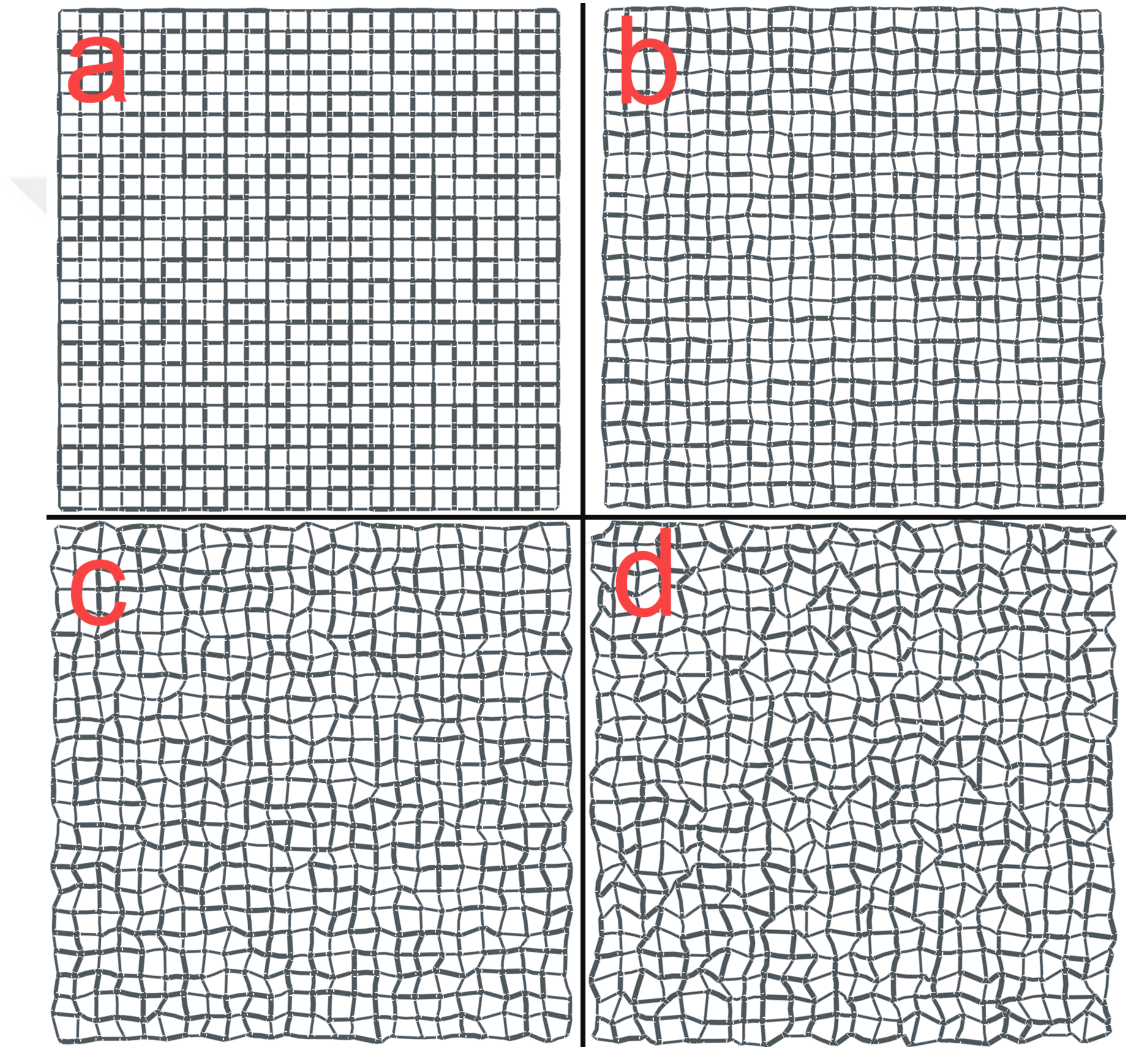


Figure 4.9: Procedurally generated 25x25 grid based road networks. Random distance parameter values increase along with the grid size .

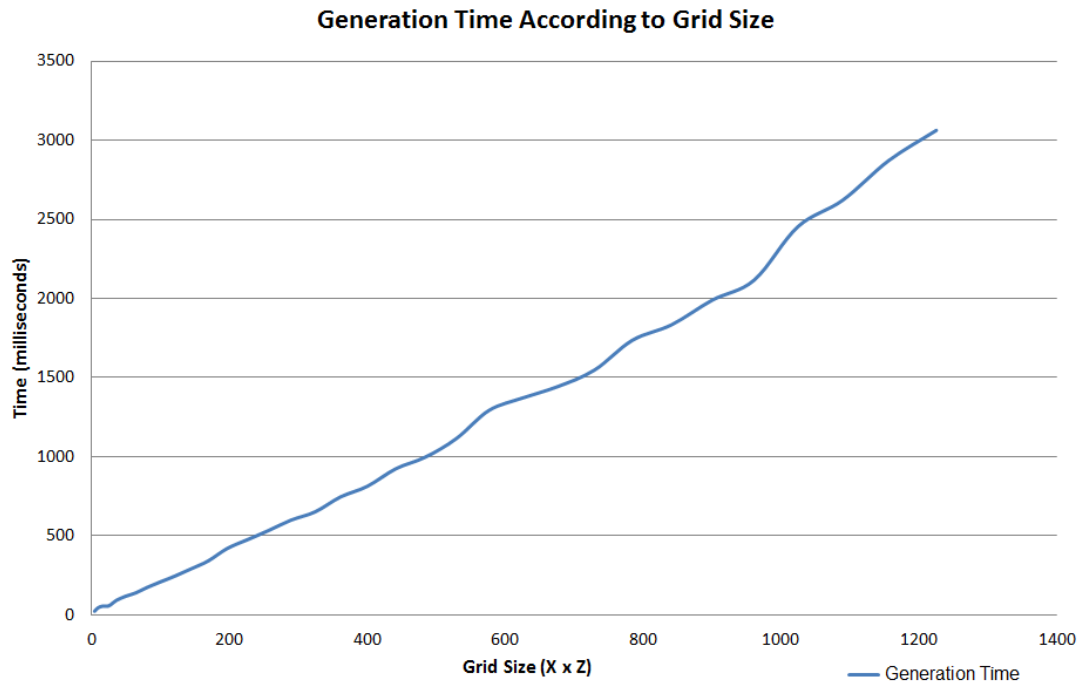


Figure 4.10: The generation time results for grid size 4 (2x2) to 1225 (35x35).

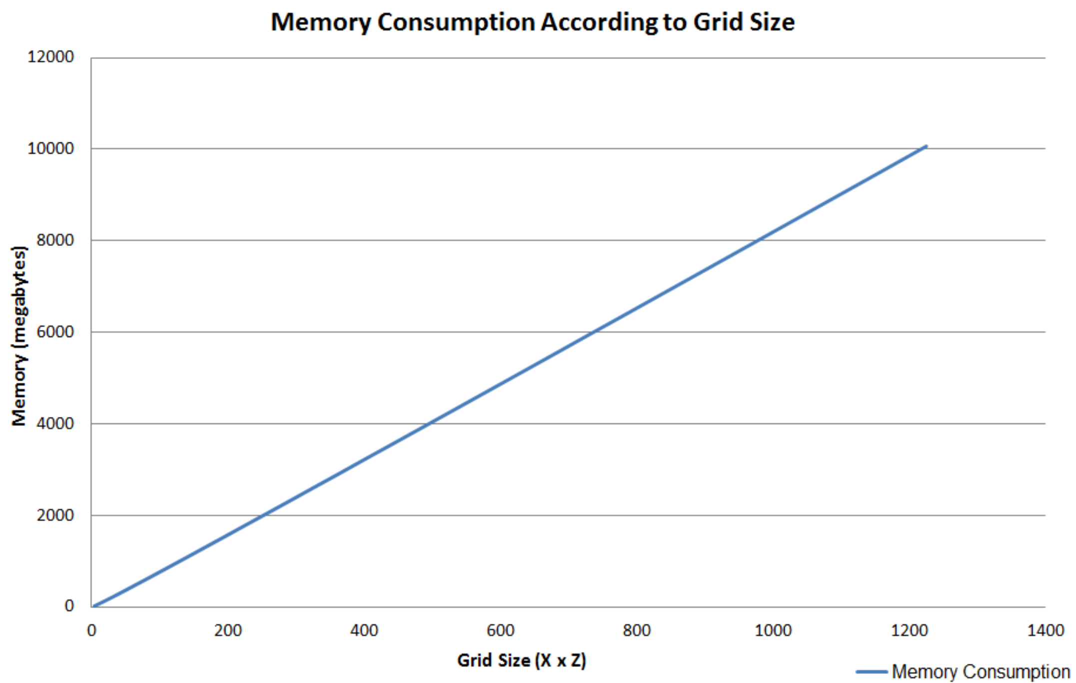


Figure 4.11: The memory consumption results for grid size 4 (2x2) to 1225 (35x35).

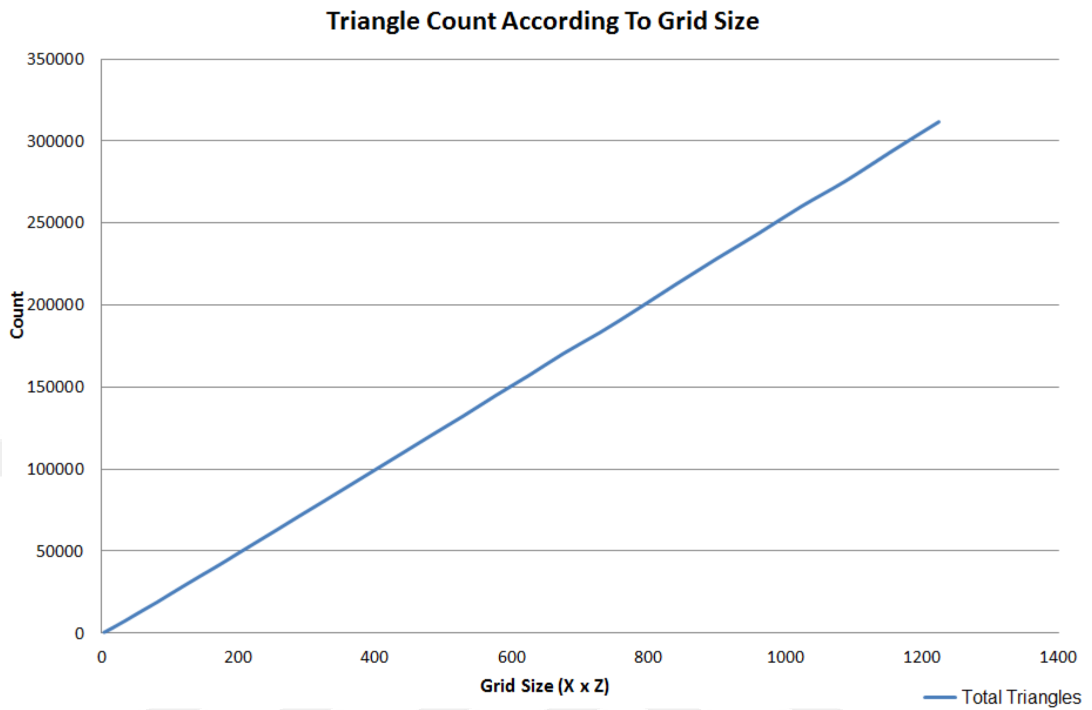


Figure 4.12: The triangle count results for grid size 4 (2x2) to 1225 (35x35).

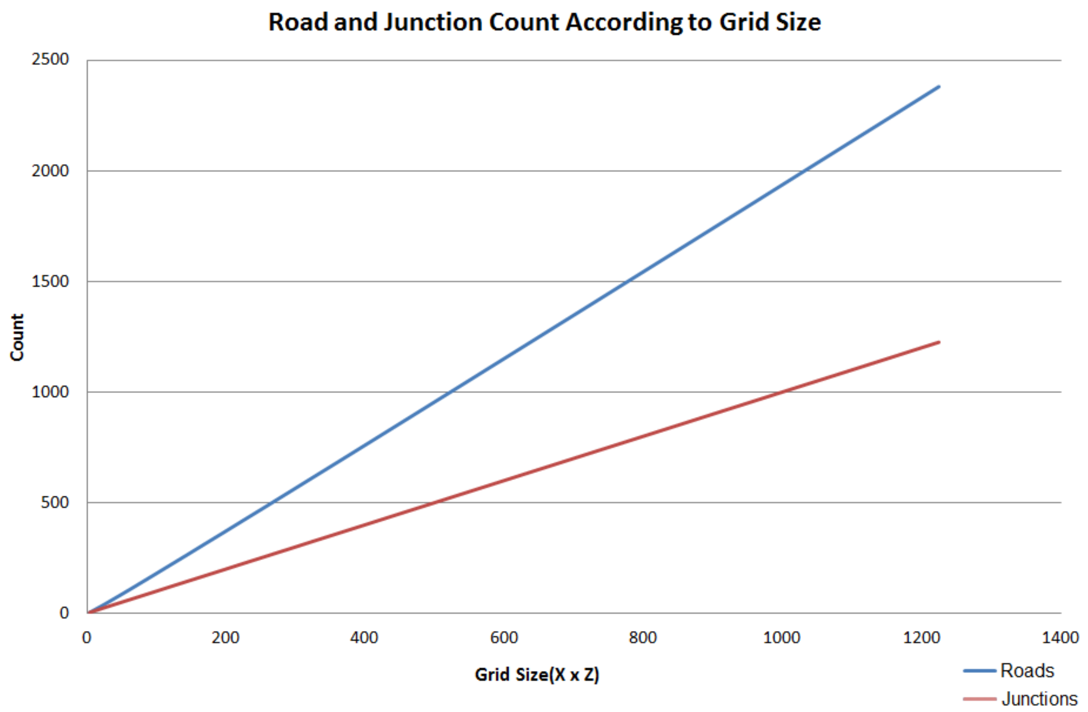


Figure 4.13: The road and junction count results for grid size 4 (2x2) to 1225 (35x35).

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

CityEngine is a complex tool with more advanced usage purposes. The method proposed in this thesis aims to generate road networks with lower polygon costs with higher visual quality. We compared our results with CityEngine with the almost identical road networks. The results show that the proposed method performs better in terms of texture and especially polygon costs in general. However, this is due to our specific conditions and does not imply CityEngine is less powerful. By concluding all the observations and comparisons, our conclusion and the possible future work is listed:

- Road networks can considerably vary and different variations would results in different costs. In some cases, i.e heavy use of sole straight road segments with no road line changes, CityEngine could work more effectively. However, in many cases for low graphical detail road networks, the proposed method is more effective as the comparison results show. Therefore, the proposed method seems more advantageous for limited devices, or high scale projects.

- Our separated road line method improves flexibility and visual quality while decreasing the cost associated with texture and not increasing polygon cost. The modeling with separated lines is similar to drawing to an asphalt canvas instead of struggling with recessing between shape bounds for various road line components i.e overtaking lines, highway shoulders, etc. Similar to road lines, the same method allows for other line based components such as arrows or crosswalks.

- The proposed method is capable of processing large procedural road network inputs and output them as a model.

- We aimed for low detail geometry, but in our implementation we left some detailing parameters to the user for better quality visuals. There is no restriction to add more optional features for detailing. For example we can add spline based curvy roads instead of limiting user to arc roads. Therefore, our tool would become more useful for projects ranging low detail to higher details. CityEngine is more capable of addressing projects with different quality scales than the proposed method's present incarnation.

- We can create a file format for storing data and make our implementation capable of interpret it. Therefore we can offer an interface for reading inputs such as node positions or which nodes consist a junction, then produce the corresponding

output. We can also use other file formats from other GPS applications such as OpenStreetMaps [51]. They include manually entered data gathered from many people and requires no license for usage.

- Our implementation supports placement of the road elements such as traffic lights. This could be detailed into more elements such as barriers, arrows, light poles, etc. Some of these components can tile like pavements (road barriers), whereas some are occasionally inserted (light poles). CityEngine's rule set is a method for coping with such cases. A similar method to CityEngine's rule sets can be added to our implementation.

- Height and terrain management is the weakest side of our proposed method. We are able to add slopes, however, our quad straight road segment approach would not let us create realistic slopes. Our straight road segments are linear, a realistic slope would continue by a smooth curve. Unless the user adds numerous nodes as an artificial curve, it is impossible to achieve that by the proposed method. We can also implement terrain embedding but Without a realistic slope curve, this feature would not be very useful.

- As future work, We could extend the use of our road network capable of generating cities. However, an extensive research on city plans may be necessary as well as the height maps.

- An AI extension can be added in the future. Our system is suitable for inserting simple driving routes. As in the case of city generation, for more advanced solutions, further research is required.

REFERENCES

[1] Wikipedia, "Barnsley Fern Fractals 4 States." [Online]. Available: https://commons.wikimedia.org/wiki/File:Barnsley{_}Fern{_}fractals{_}-{_}4{_}states.PNG

[2] ——, "Mandelbrot Zoom." [Online]. Available: https://commons.wikimedia.org/wiki/File:Mandel{_}zoom{_}07{_}satellite.jpg

[3] Scratchapixel 2.0, "Value Noise and Procedural Patterns." [Online]. Available: https://www.scratchapixel.com/lessons/procedural-generation-virtual-worlds/procedural-patterns-noise-part-1/simple-pattern-examples

[4] Wikipedia, "Fractal Terrain." [Online]. Available: https://commons.wikimedia.org/wiki/File:Fractal{_}terrain.jpg

[5] GameArt2D, "Free Platformer Game Tileset." [Online]. Available: https://www.gameart2d.com/free-platformer-game-tileset.html

[6] Wikipedia, "Graftal." [Online]. Available: https://commons.wikimedia.org/wiki/File:Graftal7.png

[7] N. Shaker, J. Togelius, and M. J. Nelson, *Procedural Content Generation in Games.* Springer, 2016.

[8] M. Hendrikx, S. Meijer, J. Van Der Velden, and A. Iosup, "Procedural Content Generation for Games: A Survey," *ACM Trans. Multimedia Comput. Commun. Appl.*, vol. 9, no. 1, 2013. [Online]. Available: http://doi.acm.org/10.1145/2422956.2422957

[9] T. R. Miller, *Benefit-cost analysis of lane marking*, 1992, no. 1334.

[10] P. Moses, "Edge lines and single vehicle accidents," *Western Road, April*, pp. 8–9, 1986.

[11] Y. Zadok and H. M. Tal, "Engaging Students in Class through Mobile Technologies–Implications for the Learning Process and Student Satisfaction," *Research Highlights in Technology and Teacher Education 2015*, p. 105, 2015.

[12] I. R. Johnston, *The effects of roadway delineation on curve negotiation by both sober and drinking drivers*, 1983, no. ARR128 Monograph.

[13] T. A. Ranney and V. J. Gawron, "The effects of pavement edgelines on performance in a driving simulator under sober and alcohol-dosed conditions," *Human factors*, vol. 28, no. 5, pp. 511–525, 1986.

[14] T. Horberry, J. Anderson, and M. A. Regan, "The possible safety benefits of enhanced road markings: a driving simulator evaluation," *Transportation Research Part F: Traffic Psychology and Behaviour*, vol. 9, no. 1, pp. 77–87, 2006.

[15] R. H. Bartels, J. C. Beatty, and B. A. Barsky, *An introduction to splines for use in computer graphics and geometric modeling.* Morgan Kaufmann, 1987.

[16] J. D. Foley, A. Van Dam, and Others, *Fundamentals of interactive computer graphics.* Addison-Wesley Reading, MA, 1982, vol. 2.

[17] E. Catmull, "A subdivision algorithm for computer display of curved surfaces," UTAH UNIV SALT LAKE CITY SCHOOL OF COMPUTING, Tech. Rep., 1974.

[18] M. Vesterinen and Others, "3D Game Environment in Unreal Engine 4," p. 21, 2014.

[19] 3D Object Converter, "Formats." [Online]. Available: http://3doc.i3dconverter.com/formats.html

[20] M. Botsch, M. Pauly, L. Kobbelt, P. Alliez, B. Lévy, S. Bischoff, and C. Röossl, "Geometric modeling based on polygonal meshes," 2007.

[21] M. Yirci, "A comparative study on polygonal mesh simplification algorithms," Ph.D. dissertation, Middle East Technical University, 2008. [Online]. Available: http://etd.lib.metu.edu.tr/upload/12610074/index.pdf

[22] B. G. Baumgart, "A polyhedron representation for computer vision," in *Proceedings of the May 19-22, 1975, national computer conference and exposition.* ACM, 1975, pp. 589–596.

[23] S. Bischoff and L. Kobbelt, "Teaching meshes, subdivision and multiresolution techniques," *Computer-Aided Design*, vol. 36, no. 14, pp. 1484–1491, 2004.

[24] K. Weiler, "Edge-based data structures for solid modeling in curved-surface environments," *IEEE Computer graphics and applications*, vol. 5, no. 1, pp. 21–40, 1985.

[25] Autodesk, "Maya," 2017. [Online]. Available: https://www.autodesk.eu/products/maya/overview

[26] D. S. Ebert, *Texturing & modeling: a procedural approach.* Morgan Kaufmann, 2003.

[27] The International Scene Organization, "File Archive." [Online]. Available: https://scene.org/

[28] G. Kelly, "An Interactive System for Procedural City Generation," Ph.D. dissertation, Institute of Technology Blanchardstown, 2008.

[29] B. Mandelbrot, "The Fractal Geometry of Nature," 1982.

[30] M. F. Barnsley, *Fractals everywhere.* Academic press, 2014.

[31] K. Perlin, "An image synthesizer," *ACM Siggraph Computer Graphics*, vol. 19, no. 3, pp. 287–296, 1985.

[32] R. C. E. Aluning and J. A. C. Hermocilla, "Terra: A 3D Terrain Generator and Visualizer."

[33] S. Lefebvre and F. Neyret, "Pattern based procedural textures," in *Proceedings of the 2003 symposium on Interactive 3D graphics.* ACM, 2003, pp. 203–212.

[34] A. Lindenmayer, "Mathematical models for cellular interactions in development I. Filaments with one-sided inputs," *Journal of theoretical biology*, vol. 18, no. 3, pp. 280–299, 1968.

[35] P. Prusinkiewicz and A. Lindenmayer, "Modeling of cellular layers," in *The Algorithmic Beauty of Plants.* Springer, 1990, pp. 145–174.

[36] G. Kelly and H. McCabe, "A survey of procedural techniques for city generation," *ITB Journal*, pp. 87–130, 2006. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.91.8713{&}rep=rep1{&}type=pdf

[37] S. Greuter, J. Parker, N. Stewart, and G. Leach, "Real-time procedural generation of pseudo infinite'cities," in *Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia.* ACM, 2003, pp. 87—-ff.

[38] Y. I. H. Parish and P. Müller, "Procedural modeling of cities," *Proceedings of the 28th annual conference on Computer graphics and interactive techniques - SIGGRAPH '01*, no. August, pp. 301–308, 2001. [Online]. Available: http://portal.acm.org/citation.cfm?doid=383259.383292

[39] G. Chen, G. Esch, P. Wonka, P. Müller, and E. Zhang, "Interactive procedural street modeling," *ACM SIGGRAPH 2008 papers on - SIGGRAPH '08*, vol. 27, no. 3, p. 1, 2008. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1399504.1360702

[40] M. F. Thomas Lechner, Ben Watson, Pin Ren, Uri Wilensky, Seth Tisue, "Procedural Modeling of Land Use in Cities," p. 8, 2004.

[41] A. Gruen and X. Wang, "CyberCity Modeler, a tool for interactive 3-D city model generation," *Photogrammetric Week*, pp. 1–11, 1999. [Online]. Available: http://www.ifp.uni-stuttgart.de/publications/phowo99/gruen99.pdf

[42] P. Doucette, P. Agouris, and A. Stefanidis, "Automated road extraction from high resolution multispectral imagery," *Photogrammetric Engineering & Remote Sensing2*, vol. 70, no. 12, pp. 1405–1416, 2004.

[43] C. Zhang, E. Baltsavias, and A. Gruen, "Knowledge-based image analysis for 3D road reconstruction," vol. 1, no. 4, pp. 3–14, 2001.

[44] Unity Technologies, "Unity Game Engine," 2017. [Online]. Available: https://unity3d.com/

[45] ——, "Creating and Using Materials," 2017. [Online]. Available: https://docs.unity3d.com/Manual/Materials.html

[46] H. P. Lensch, "Realistic Materials in Computer Graphics," 2005. [Online]. Available: https://cg.informatik.uni-freiburg.de/course{_}notes/graphics2{_}07{_}materials.pdf

[47] Unity Technologies, "Standard Shader," 2017. [Online]. Available: https://docs.unity3d.com/Manual/shader-StandardShader.html

[48] Andasoft, "EasyRoads3D," 2017. [Online]. Available: https://assetstore.unity.com/packages/tools/easyroads3d-free-987

[49] Autodesk, "FBX File Format Overview." [Online]. Available: https://www.autodesk.com/products/fbx/overview

[50] Esri Enterprise, "CityEngine." [Online]. Available: http://www.esri.com/software/cityengine

[51] OpenStreetMaps, "Open Street Maps." [Online]. Available: https://www.openstreetmap.org

[52] M. K. Agoston, *Computer Graphics and Geometric Modeling: Implementation and Algorithms*, ser. Computer Graphics and Geometric Modeling. Springer London, 2005. [Online]. Available: https://books.google.com.tr/books?id=TAYw3LEs5rgC

[53] J. D. Foley, A. Van Dam, S. K. Feiner, J. F. Hughes, and R. L. Phillips, *Introduction to computer graphics*. Addison-Wesley Reading, 1994, vol. 55.

[54] P. S. Heckbert, "Survey of texture mapping," *IEEE computer graphics and applications*, vol. 6, no. 11, pp. 3–10, 1986.

[55] T. Ijiri, "Sketch L-System: Global Control of Tree Modeling Using Free-form Stroke." [Online]. Available: takashiijiri.com/ProjSketchLsystem/

[56] T. Ijiri, S. Owada, and T. Igarashi, "The sketch l-system: Global control of tree modeling using free-form strokes," in *Smart Graphics*, vol. 4073. Springer, 2006, pp. 138–146.

[57] G. Kelly and H. McCabe, "Citygen: An interactive system for procedural city generation," *Fifth International Conference on Game Design and Technology*, 2007. [Online]. Available: http://www.citygen.net/files/citygen{_}gdtw07.pdf

[58] Unity Technologies, "Anatomy of a Mesh." [Online]. Available: https://docs.unity3d.com/Manual/AnatomyofaMesh.html

[59] Wikipedia, "Koch Snowflake." [Online]. Available: https://commons.wikimedia.org/wiki/File:KochFlake.svg