Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

David Bjerregaard Madsen

Procedural City Generation in Unity Engine

Master's thesis in Industrial Cybernetics
Supervisor: Sverre Hendseth
June 2020

NTNU
Kunnskap for en bedre verden

David Bjerregaard Madsen

Procedural City Generation in Unity Engine

Master's thesis in Industrial Cybernetics
Supervisor: Sverre Hendseth
June 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU

Kunnskap for en bedre verden

Assignment Description

A back-end for drawing a procedurally generated city environment shall be developed with applications in game development in mind. The software is to be implemented using Unity engine as the main development platform.

The software should be modular and have a variable input in the form of manipulating a user interface or configuratin file, which determines the properties and features of the generated geometry.

The student shall as a background study examine existing methods on how urban geometry can be modeled, and make choices/recommendations on the methods to implement. New methods may also be investigated. Note that even though ”graphics quality” is great, what will make the world interesting is the level and variation of detail rather than the graphics quality itself.

A software demo is to be presented, showcasing the capabilities of the system.

Student:

David Bjerregaard Madsen

Supervisor:

Sverre Hendseth

Abstract

Procedural generation is an ever increasing area of research with a wide number of applications. In this thesis we present and discuss existing methods of procedurally generating urban environments, most importantly by Parish and Müller (2001) and Chen et al. (2008).

A module based generation model is described and discussed. Using Unity engine as the development platform, several of the existing methods as well as some new techniques are implemented. Road networks are generated using hyperstreamline tracing over a Perlin noise field and connected using a Bézier curve connection algorithm. A new technique for generating building primitives based on underlying road geometry is devised and implemented. Meshing techniques for both road generation and building primitives is discussed and implemented. The results are presented and compared with real life examples, along with the strengths and limitations of the developed system. A study is also done to measure performance and scalability.

The results suggest that hyperstreamline tracing is a viable method of generating road paths. A high degree of variation was achieved with adjusting a few input parameters. The lower detail level was found to be inadequate. The paper concludes that while the overall macroscopic level of detail was satisfactory, the implemented system is too crude and requires additional development to be a viable alternative to manual game world creation at this stage.

Sammendrag

Prosedyrisk generering er et stadig økende forskningsområde med en rekke bruksområder. Denne avhandlingen undersøker eksisterende og nye metoder innen prosedyrisk generering av urbane områder, med hovedvekt på verkene av Parish and Müller (2001) og Chen et al. (2008).

En modulbasert modell blir beskrevet og diskutert. Flere eksisterende og nye metoder implementert med Unity-motoren som utviklingsplattform. Veinett blir generert ved å tegne strømlinjer i et underliggende Perlin-støyfelt, og koblet sammen ved med Bézier-kurver. En ny teknikk for å generere bygningsprimitiver basert på underliggende veigeometri blir utviklet og implementert. Meshing-teknikker for både vei- og bygningsgenerering blir diskutert og implementert. Resultatene blir presentert og sammenlignet med virkelige eksempler, i tillegg til styrkene og begrensningene til det utviklede systemet. En studie blir gjennomført for å undersøke systemets ytelse og skalerbarhet.

Resultatene foreslår at å tegne strømlinjer er en brukbar metode for å generere veinett. Høy grad av variasjon ble oppnådd ved å justere få inngangsparametre. Detaljnivået var ikke tilstrekkende. Det konkluderes med at makroskopisk detaljnivå var tilstrekkelig, men at det implementerte systemet fortsatt trenger ytterligere utvikling for å være et brukbart alternativ til å manuelt generere spillverdener ved dette stadium.

Table of Contents

List of Figures

Chapter 1

Introduction

Procedural generation is a technique often used to describe automatically generated data. This is typically achieved through the application of algorithms based on organized randomness, where a set of boundary rules are set and the contents of said boundaries are generated using different methods and functions. Procedural generation has applications in many areas, from entertainment such as visual effects, animation and video games to industrial applications like architecture or engineering.

With an ever increasing urban population worldwide the modeling and generation of city environments is a highly relevant application for procedural generation. However, it is far from a new area of research. Generating a whole city is naturally a complex task, and therefore a number of subproblems requiring different solutions need to be tackled. In this regard, there exist a multitude of projects focused around procedurally generating various elements of an urban environment.

1.1 Motivation & Problem Description

The last decades has seen a large increase in game development, with more and more games released each year. Many of these games take place in urban environments, and with more and more processing power available to both developers and end users, the size and complexity of game maps naturally increase. However, most of these game maps are typically manually created. This may put an upper limit in both size and content variation, as designing game worlds can be a time consuming process.

To increase the time efficiency and content variation in game maps, elements of procedural generation may be introduced. The use of procedural generation in game development has long been a popular topic. One of the advantages of procedural generation is that we can achieve high degrees of variation with little additional time investment. This naturally makes partly or completely procedurally generated game worlds highly desirable.

Although enticing, it is far from a trivial area of research. There exists no general model for procedurally generating content as a whole. Ad-hoc methods are often used to take care of special cases, as it can be difficult to completely generalize an increasingly complex system. Secondly, the limitations of current procedural generation techniques may not allow for the level of detail needed for a completely auto-generated world, and as such procedural generation is often used in combination with manual adjustment to produce the desired result.

When applying procedural generation to generate urban environment, these challenges become evident. We require different models to generate different elements of a city; from the road network with its interconnection of roads and intersections, bridges or tunnels that may be dependent on the surrounding terrain, or the building shapes, placements and sizes. On top of this, whether or not these principles are useful to game development poses additional requirements to the model itself. We may require a model that can be rapidly altered and tested to allow for fast prototyping, and performance restrictions may be taken account for. Combining all this into a single system comprises a difficult challenge. How, if at all, can this be done?

1.2 Project Goal

The goal of this project is to survey existing procedural generation techniques related to urban environments and investigate their viability for game development. We aim to develop a system for procedurally generating cities and urban geometry in Unity engine using the discovered methods. The possibility of implementing new techniques is also to be considered where applicable. The hypothesis is that existing methods used to generate urban geometry can be applied in Unity engine to allow for rapid prototyping of game worlds. Thus we aim to create a general model that may serve as a foundation for producing procedural citycapes, mainly with applications in game development in mind.

Research Questions

- Can existing methods of procedurally generating cities be efficiently implemented in Unity engine?
- Is the designed system viable for applications in video game development?

Chapter 2

Background

2.1 Procedural Generation and its Applications

The areas where procedural generation can be applied is almost limitless. Anywhere wher large amounts of content is needed, procedural generation can usually be applied in some form.

2.1.1 Applications in Video Games

Video game development and design may be one of the areas where procedural generation is most commonly used. Developers can utilize powerful techniques for quickly generating a large variety of content. Within game design, procedural generation is also used for a variety of different purposes; from generating levels or whole game worlds to generating quest-lines, objectives or encounters during gameplay.

Procedural Level Design

One of the earliest games with procedurally generated levels is *Rogue* from 1980. The text-based dungeon exploration game takes the player through a maze of monster filled dungeons, with the main objective being finding the Amulet of Yendor at the end. In addition to the dungeons, monster encounters and collectable items are also procedurally generated, making each playthrough different from the last. A multitude of similar role-playing games have been developed since. These games are collectively referred to as *Roguelikes* after the original game, and all build on the same principles of procedurally generated dungeon levels in varying ways.

Another game that implements extensive use of procedural level generation is the widely popular role-playing game series *Diablo* from Blizzard Entertainment. The games all feature a third-person isometric style map where the player navigates the character

around. Although the games generally feature fixed storylines, the dungeons and wilderness areas between key cities are procedurally generated.

Open-World Procedural Games

Many modern video games are *open-world*, meaning the player can move freely around in the game world with little or no pre-defined objectives to follow. These games emphasize heavily on the exploration aspect of gameplay, and are therefore ideal candidates for large, procedurally generated worlds. Many open-world games are often also *sandbox* games, where the player is able to create, modify and destroy the game environment, much like a child in a playground sandbox.

Figure 2.1: Screenshot from *Minecraft*

One such game is *Minecraft* (2011). The game features a sandbox world of biomes (forests, deserts, oceans etc.) and underground cave systems, all built up by discrete blocks (voxels) of varying materials. The application of procedural world generation in this case gives the player almost endless opportunity to travel and explore, which would not have been practically possible in a manually made game world. The terrain is generated using Perlin noise[1].

Similarly, in *No Man's Sky* (2016) the game revolves around visiting and exploring different planets, gathering resources and engaging in combat with alien organisms living on the planets. Each planet is procedurally generated, and has an unique ecosystem of flora and fauna.

2.1.2 Applications in Visual Effects

Another area where procedural generation is often used is in visual effects. Many movies today have elaborate visual effects and require lots of 3d rendered objects. In Peter Jack-

[1] According to https://minecraft.gamepedia.com/Customized#Advanced_settings, accessed 2020-03-29. For more details about Perlin noise, see section 2.1.4

son's *Lord of the Rings* triolgy, large armies of soldiers were procedurally generated using the software *Massive*. The software was initially developed specifically for the films, and has since become the leading software for generating large, natural-looking crowds and autonomous character animation (Massive Software, 2020). Notable big picture titles that utilize the software include *Inception* (2010), *I, Robot* (2004), *Avatar* (2009). In addition to its motion picture applications, it can also be used to simulate crowds for engineering and architectural purposes.

2.1.3 Limitations and Challenges

A problem often faced with procedurally generated content is repetitiveness. Although the world in *Minecraft* is practically endless, the lesser features of the terrain is repeated with little variation. This is a common factor in procedurally generated content. Many times procedurally generating everything is simply not viable, as the exercise of designing the generator functions can become as time consuming as it would be to just generate the content manually, defeating the purpose of using procedural generation.

2.1.4 Commonly Used Methods

As procedural generation has many applications in a wide amount of fields, a number of different methods are used. In terrain generation, a noise function such as Perlin noise is often used to create terrain (Parberry, 2015), and L-systems may be used to describe complex growing patterns in plants (Lindenmayer, 1968).

L-systems

Lindenmayer systems, or L-systems, are a form of rewriting systems that use a set of rules to succesively modify an object through iteration (Lindenmayer, 1968; Prusinkiewicz and Lindenmayer, 1990). L-systems were first devised by Lindenmayer to model the growth processes of plant development, and are useful in drawing fractal patterns and shapes. The L-system starts with an axiom ω, typically a string, which serves as the root of the generation. A set of rules are applied at each iteration, succesively modifying the previous iteration. Deterministic in nature, L-systems are capable of producing complex geometrical patterns from a compact dataset.

Coherent Noise Functions

When generating terrain, a coherent noise[2] function is often used. The noise function is often rescaled and stacked with varying amplitude and frequency to create different levels of features in the terrain.

One such noise function is Perlin noise, first described by Perlin (1985). The function was mainly developed in two dimensions for generating textures, but can also be

[2]Coherent noise is a smooth (continuous), pseudorandom type of noise, whereas typical Gaussian or white noise is random and discontinuous in nature.

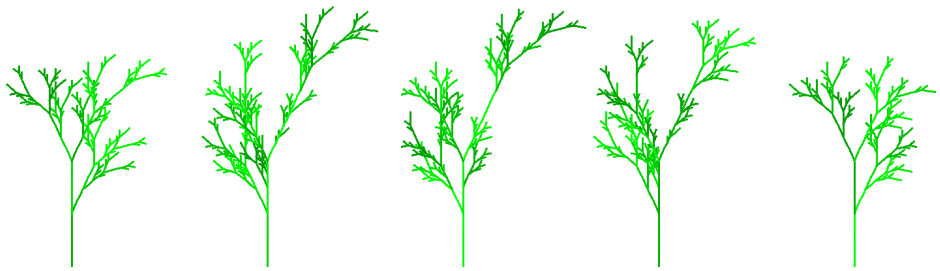
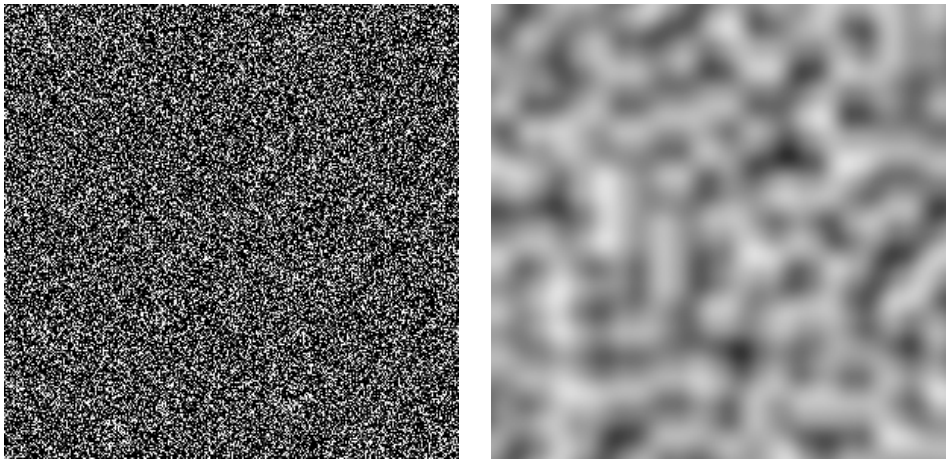


Figure 2.2: L-system modeling the growth of a plant (Bhadury, 2017)

implemented in any number of dimensions. The algorith works in short by calculating pseudorandom gradient vectors at regular intervals, and interpolating between these and the sample point to get a noise value. In the one-dimensional implementation of the algorithm, the gradient vectors are replaced by scalars in the range $[-1, 1]$ instead.



(a) Gaussian noise **(b)** Perlin noise

Figure 2.3: Gaussian noise in comparison to Perlin noise (Thomas, 2011)

In addition to generating terrain, coherent noise functions can be used for generating a variety of elements. Notable application examples include volumetric clouds (Kutz, 2012) and generating simulated breast tissue (Dustler et al., 2015).

Markov Chains

Markov chains is a stochastic mathematical model that describes a chain of possible events. Each event in a Markov chain satisfies the *Markov property*, meaning the event is independent of all previous events. The chain may be represented as a directed graph, in which

each node has edges with the associated probabilities of a move along the respective edges. Similar to L-systems, such models can be used to describe and model many natural processes, and have applications in physics, chemistry and biology.

Markov chains can also be used to procedurally generate text. By training the model on existing text, a Markov model may be used to generate text that is visually similar to the training data. The subreddit *Subreddit Simulator* consists of exclusively robot users that use Markov chains in this manner to generate content.[3]

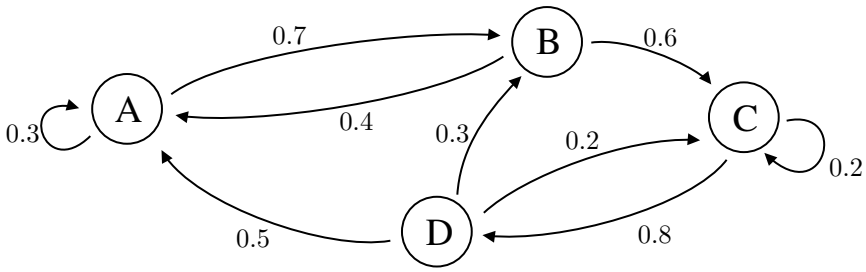


Figure 2.4: Example of a graph illustrating a Markov chain process. The edge costs represent the probability of moving between nodes in the direction of the arrow.

Voronoi Diagrams

Voronoi diagrams or Voronoi cells is a method of partitioning an area of points into cells based on the proximity to the nearest point. The borders between the cells represent the points of equal distance between two points. The technique produces cells of irregular polygons when applied to randomly scattered points on a plane, but will also form regular tesselation of hexagons when applied to a 2D lattice.

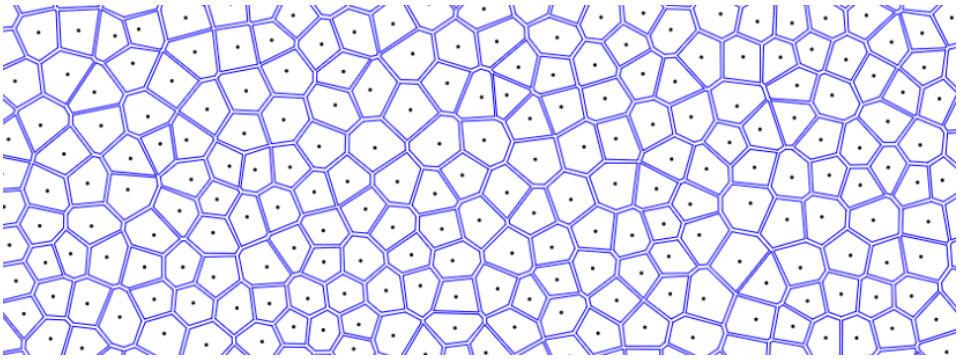


Figure 2.5: Voronoi cells generated from randomly scattered points (Hosier, 2016)

[3]According to `https://www.reddit.com/r/SubredditSimulator/comments/3g9ioz/what_is_rsubredditsimulator`

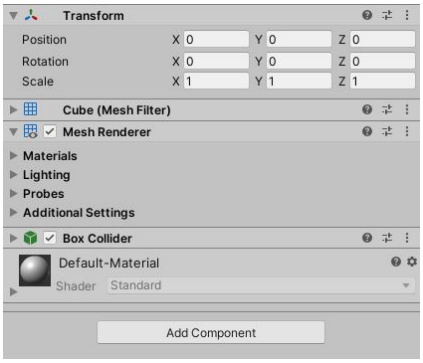
2.2 Unity Engine

Unity Engine is a game development engine first released in June 2005. The engine, originally released as a MacOS-exclusive, has since its introduction been extended to support over 25 platforms. It features a scripting API in C# as well as a drag and drop user interface which allows users to make both 2D and 3D games. The scripting API features a wide variety of extension methods useful when making games on top of the native libraries in C#. This flexibility and ease of use has led it to become a popular game development engines, with a large number of released games using Unity as the main development platform.

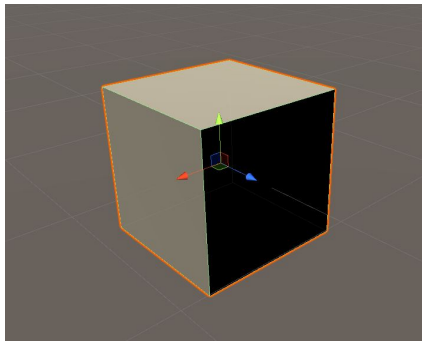
However, the latter years Unity has seen an extension into industries outside of game development.[4] From visual effects and animation rendering to engineering and architectural applications, Unity's 3D rendering engine is a powerful tool that has applications in a number of industries. In engineering, Unity can be used in product lifecycle and 3D rendering, with support for CAD model importing and rapid prototyping.

2.2.1 Game Objects, Components & Scripting

Unity is built around *game objects*, which is the base class for all entities in a scene. Game objects serve as a container for attached *components*. The components are extension classes that add functionality to otherwise inert game objects. By default, an empty game object has a *transform* component attached, which is used to manipulate spatial position, rotation and scale of the object. Unity has an array of built in components that can be attached, from mesh rendering to physics interactions and collisions, or sound manipulation. Unity's scripting API also allows for trivial implementation of custom components, which allows for near limitless customization.



(a) Components overview panel in Unity



(b) 3D view of the resulting cube

Figure 2.6: Example of components attached to a game object. Other than the default *transform*, *mesh filter* and *mesh renderer* components are attached to define the mesh and draw the cube to screen. A *box collider* is also attached to allow for physics interactions. Screenshots from Unity engine.

[4] For a more detailed list of application areas, see `https://unity.com/solutions`

Game objects also support inheritance, with the transform of a child object being dependent on its parents transform. Game objects can also be created and destroyed at runtime, and components can be attached in code outside of the drag-and-drop interface.

2.2.2 Mesh Rendering

Related to game development and procedural generation are also meshing. In general a mesh is a subdivision of a continuous geometrical area into discrete cells. When drawing 3D objects to a screen, each object is represented as a mesh of vertex coordinates connected in sets of three to create triangular faces. The vertex coordinates and triangles are then mapped onto 2D pixel coordinates on screen during rendering to produce the image of the 3D object. Meshing is used mainly to simplify continous geometry to save computation time. Combined with texturing and lighting techniques, a relatively low-poly mesh can appear indistingushable from a continunous surface.[5] By varying the complexity of the mesh, one can trade off detail for faster computation time. This technique, often called *level of detail* or simply *LOD*, is utilized extensively in game development. In practice, this is implemented by replacing the mesh with a lower poly mesh when the object is farther from the camera.

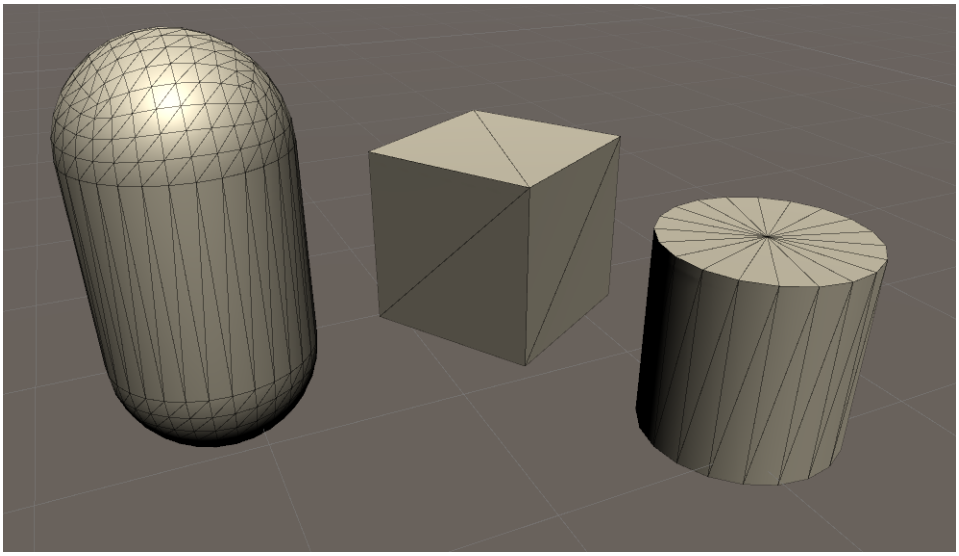


Figure 2.7: Various shaded objects with their wireframe meshes visible. Note the difference in polygon count between the curved and flat surfaces. Screenshot from Unity engine.

Since meshing and rendering 3D objects can be complex tasks, an advantage with using Unity is that it contains built-in components for this. As mentioned briefly in the section above, unity has two components responsible for mesh rendering. The *mesh filter* acts as a container for the mesh geometry; the triangles and vertices arrays as well as the

[5] Polygon count refers to the number of triangles in the mesh. A higher poly count is naturally associated with increased computational load, so keeping the polygon count at a minimum is usually desirable

UV coordinates for texturing. The *mesh renderer* is responsible for drawing the mesh to the screen. The mesh renderer also contains methods for lighting, texturing and shading. As each triangle of vertices naturally has two faces, but only one can be drawn to the screen at any given time, it is common practice for rendering algorithms to only render one face per triangle in any given mesh. Which face to render is given based on the *winding order* of the face. Unity engine uses a clockwise winding order to render mesh faces, meaning vertices forming each triangle should be listed in a clockwise order.

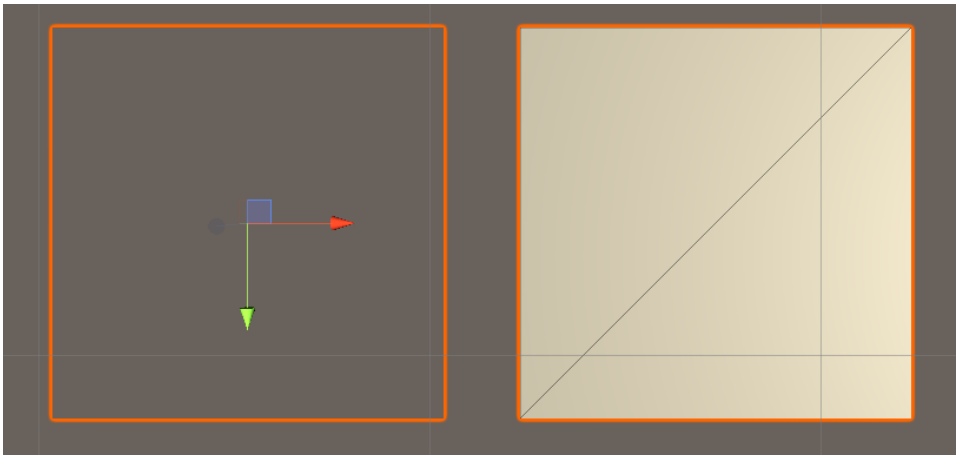


Figure 2.8: Two identical square meshes where the leftmost is flipped 180^{o} along the x-axis (red). The triangles of the left mesh are thus not visible to the camera. Screenshot from Unity engine.

The mesh renderer also handles texturing and lighting. To texture the mesh, a second set of coordinates usually called UV coordinates[6], are used. The flat texture is mapped onto the vertex coordinates of the mesh, by associating each vertex with an UV texture coordinate. The UV coordinates are typically in the range $[0, 1]$ in both the u and v axes relative to the texture. To calculate lighting, the renderer uses the normal vectors of each vertex, as the reflected light intensity is dependent on the angle of attack on a surface (figure 2.9).

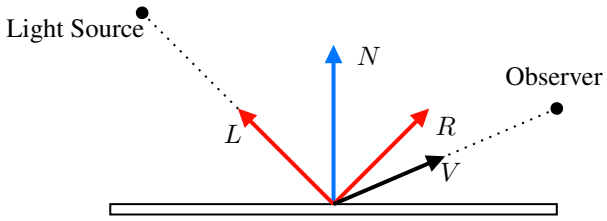


Figure 2.9: Reflection of light off a surface

[6]"UV" is used instead of "XY" to avoid confusion, as "XYZ" axis labels are typically reserved for vertex coordinates

2.3 Related Work

Procedural generation of cities is an open research area with many interesting approaches and solutions. The complex city environment gives rise to a lot of interesting and challenging procedural generation problems. In this section we will give an overview of the subproblems faced when modeling and generating cities and related content, and present a selection of existing approaches to these challenges.

2.3.1 Street Network Modeling

Central to the problem of modeling urban areas is modeling street networks. In the system *CityEngine* developed by Parish and Müller (2001), an extended L-system is used for generating both the road networks and buildings in a virtual city. The system takes into account population density maps as well as geographical data to to connect densely populated areas with networks of highways. The highway network is subsequently subdivided into smaller streets and lastly populated with procedurally generated buildings to form a complete city. A unique aspect of this L-system is the fact that it is *self-sensitive*, meaning the network generated can intersect with itself and form closed loops. This is done by generating intersections if the street end is close or overlaps another street as seen in figure 2.10.

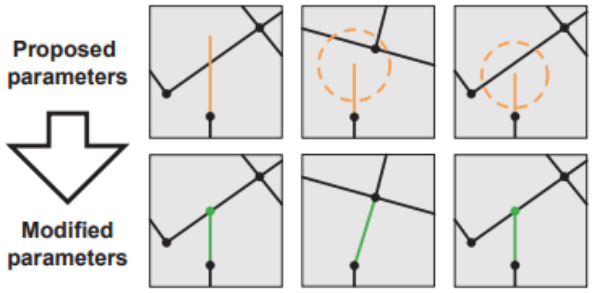


Figure 2.10: Principle of the self sensitive L-system in CityEngine (Parish and Müller, 2001)

Chen et al. (2008) approaches the problem of modeling street networks by utilizing user defined tensor fields as a basis for generating street networks. Similar to *CityEngine*, this approach takes in geographical data as well as user input to generate a tensor field, which the road network is generated from. In general, a tensor field T is a continuous function that associates every point $\mathbf{p} = (\mathbf{x}, \mathbf{y}) \in \mathbb{R}^2$ with a tensor $T(\mathbf{p})$ (Chen et al., 2008; Delmarcelle and Hesselink, 1994). An example of a tensor used for road network generation is a symmetric 2×2 matrix on the form:

$$R \begin{bmatrix} \cos 2\theta & \sin 2\theta \\ \sin 2\theta & -\cos 2\theta \end{bmatrix}, \quad R \geq 0, \quad \theta \in [0, 2\pi) \tag{2.1}$$

An important concept when it comes to tensor fields are *hyperstreamlines*, first presented by Delmarcelle and Hesselink (1993). Hyperstreamlines are curves tangent to an eigenvector field, and have a wide range of applications. By defininining $\theta = \tan\frac{\partial x}{\partial y}$, the tensor in (2.1) has major and minor eigenvectors parallel to and perpendicular to the gradient of the field respectively. Thus, the roads of a road network can be traced along the hyperstreamlines to get a road network that follows the geographical data of the surrounding terrain Chen et al. (2008). They also demonstrate how other tensors than (2.1) can be used to form different patterns of road networks, for example a "circular" tensor described by Zhang et al. (2007) to form radial type road patterns, and techniques on how to blend different tensor fields for interesting results.

To trace the hyperstreamlines along the eigenvector field, Chen et al. (2008) use a numerical integration technique based on the Runge-Kutta methods described by Cash and Karp (1990). This family of integration methods is often used to calculate numerical solutions to otherwise continuous differential equations. One of the simplest and most well known of these methods methods is the *Euler method*, which can be expressed on the form (Egeland and Gravdal, 2003):

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h\mathbf{f}(\mathbf{y}_n, t_n)$$

where h is the step size, $\mathbf{y}_n$ is the current state, and $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}_n, t_n)$ is some continuous time-varying function. The Euler method approximates the integral $\mathbf{y} = \int \mathbf{f}(\mathbf{y}_n, t_n)dt$ with increasing accuracy as h becomes smaller, at the cost of increased computation time. However this method has its drawbacks, namely that large fluctuations in $\dot{\mathbf{y}}$ may require very small step sizes h in order to give an accurate approximation. A solution to this is calculating more approximations of $\mathbf{f}(\mathbf{y}_n, t_n)$ recursively, and using a linear combinations of these approximations to calculate the next step $\mathbf{y}_{n+1}$. More accurate approximations increases the stability region of the numeric integration method (the range of h which gives accurate solutions), and thus a larger step size can be utilized (Egeland and Gravdal, 2003). This concept is the core of Runge-Kutta methods, and σ-stage Runge-Kutta methods can be written on the general form:

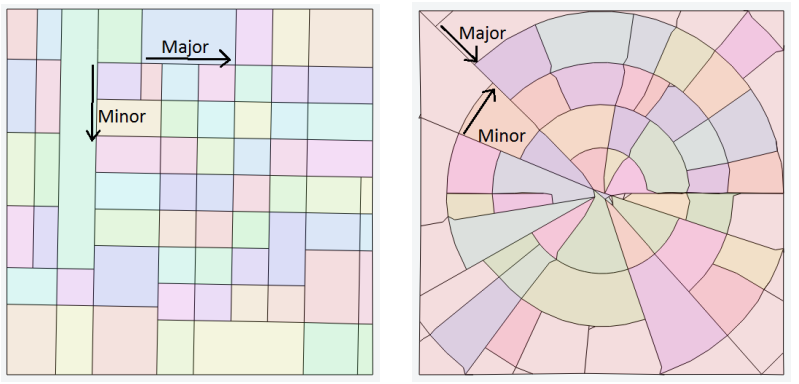
$$\begin{aligned}
\mathbf{k}_1 &= \mathbf{f}(\mathbf{y}_n, t_n) \\
\mathbf{k}_2 &= \mathbf{f}(\mathbf{y}_n + ha_{21}\mathbf{k}_1, t_n + c_2 h) \\
\mathbf{k}_3 &= \mathbf{f}(\mathbf{y}_n + h(a_{31}\mathbf{k}_1 + a_{32}\mathbf{k}_2), t_n + c_3 h) \\
&\vdots \\
\mathbf{k}_\sigma &= \mathbf{f}(\mathbf{y}_n + h(a_\sigma \mathbf{k}_1 + \cdots + a_{\sigma,\sigma-1}\mathbf{k}_{\sigma-1}), t_n + c_\sigma h) \\
\mathbf{y}_{n+1} &= h(b_1\mathbf{k}_1 + \cdots + b_\sigma \mathbf{k}_\sigma)
\end{aligned}$$

where $\mathbf{k}_1, \ldots, \mathbf{k}_\sigma$ are the stage computations. The various parameters in the stage computations of explicit Runge-Kutta methods are usually presented in a standard format as a

Butcher-array:

$$\begin{array}{c|ccccc} 0 & & & & & \\ c_2 & a_{21} & & & & \\ c_3 & a_{31} & a_{32} & & & \\ \vdots & \vdots & \vdots & \ddots & & \\ c_\sigma & a_{\sigma 1} & a_{\sigma 2} & \dots & a_{\sigma,\sigma-1} & \\ \hline & b_1 & b_2 & \dots & b_{\sigma-1} & b_\sigma \end{array}$$

One of the most common Runge-Kutta methods is the four stage explicit Runge-Kutta method RK4. It is often referred to as *the* Runge-Kutta method as it is widely used. A practical implementation of this method can be seen in the .NET implementation by Evans (2015). The project uses the road generation methods of Chen et al. (2008), and uses RK4 to trace the hyperstreamlines through a tensor field. Since the tensor field is not a time varying function, the time varying element of $\mathbf{f}(\mathbf{y}_n, t_n)$ can be disregarded along with the interpolation parameters $c_1, \dots, c_\sigma$. Furthermore, a point on the plane $\mathbf{p}$ is used as the state vector $\mathbf{y}$, thus the eigenvectors of $T(\mathbf{p})$ can be approximated at each stage calculation. This is done iteratively until the end of the map is reached, resulting in a traced hyperstreamline along the tensor field. The process is then repeated at regular intervals in the plane for both the major and minor eigenvectors for a complete road network. Examples of this can be seen in figure 2.11.



(a) Grid tensor field **(b)** Radial tensor field

Figure 2.11: Examples of hyperstreamline grids traced using RK4 by Evans (2015)

2.3.2 Procedurally Generated Buildings

Another vital part of the cityscape is naturally the buildings in the city. As described by Parish and Müller (2001), the buildings generated by *CityEngine* are modeled using a parametric, stochastic L-system. After the street network has been generated, it is subdivided recursively into lots where buildings are placed. The buildings are then created based on

the lot size and a height value determined from the population density, as well as a user set limit on building height. Using an iterative process including extrusion and transformations, the buildings are carved from the bounding box (lot area times the height). An example of this process can be seen in figure 2.12.

Figure 2.12: L-system building generation process (Parish and Müller, 2001)

Rooftop templates are also utilized to give more detailed roofs than the L-system is capable of generating. The output of the L-system is then fed to a parser that translates the output string into geometry which can be drawn to the screen. According to the authors, this method can produce a large variety in the outcome of buildings, but as the shape of the building is determined by its ground plan, the functionality can not be represented using this method.

The buildings are also procedurally textured. Different textures are layered in grids to make up the buildings facades. This principle is illustrated in figure 2.13. By overlaying a background texture, i.e. bricks, with a grid of window textures, they achieve a high degree of variety in the textures of the facades of the buildings.

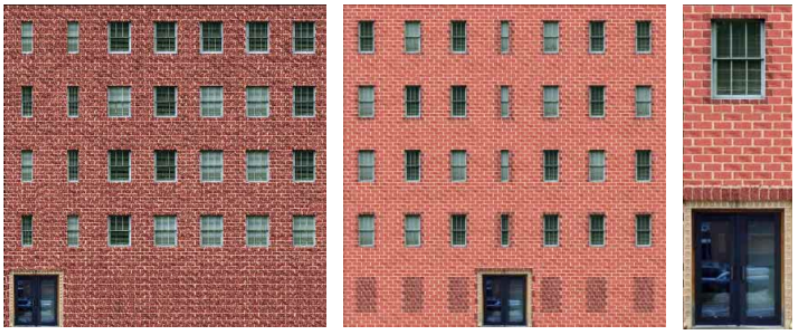


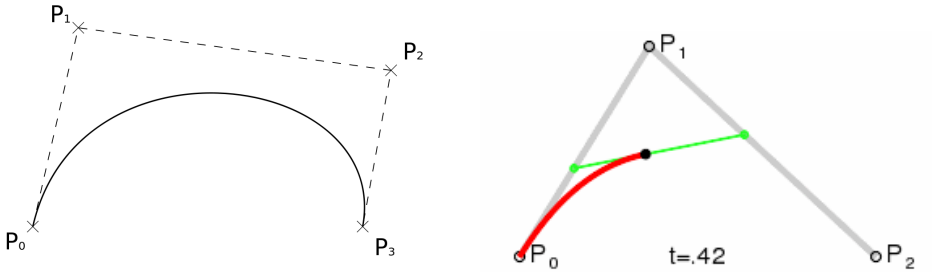
Figure 2.13: Procedurally generated facade textures (Parish and Müller, 2001)

2.3.3 Spline-Based Procedural Geometry

Another technique that is frequently used in procedural generation is spline-based geometry. A spline is a piecewise defined line segment that is generally obtained by interpolation

techniques. Splines have applications in many areas where parametric curves are useful. One of the earliest published papers was by Birkhoff and de Boor (1964) at General Motors, where they used splines to model automobile bodies in the early 1960s. A notable current-day application can be found in the Adobe software suite, where splines have become a standardized tool in everything from creating vectorized paths to controlling animations.

One of the most used spline types, named after French engineer Pierre Bézier is the *Bézier curve*. Bézier curves utilize linear interpolation between one or more *control points* to create a curved path. As seen in figure 2.14a, the curvature of the line between P_0 and P_3 is determined by the control points P_1 and P_2. Note that the curved line at each end is tangent to the line between the respective end point and its control point.



(a) Cubic Bézier curve **(b)** Quadratic Bézier curve

Figure 2.14: Examples of Bézier curves of different degrees

The simplest Bézier curve is one with a single control point shared by both end points. The curve is obtained by first interpolating some distance $t \in [0, 1]$ between line segments $P_0 - P_1$ and $P_1 - P_2$. A point on the line between these first interpolation points is then calculated using the same t-value. This third interpolation point "traces" the curve, as t is varied between 0 and 1, as illustrated in figure 2.14b. The principle is the same for the cubic Bézier curve, except it involves one more layer of interpolation between line segments. The degree of the curve is derived from the fact that the successive layers of linear interpolation that occurs as control points are added can be expressed as Bernstein polynomials of degree n. In general, a Bézier curve of n degrees has $(n-1)$ control points.

A notable application of Bézier curves in procedural generation is demonstrated in a presentation by Holmér (2015) at Unity's Unite conference in Boston 2015. By extending the concept of 2D splines into three spatial dimensions, the author uses Bézier curves for path creation in game development. The curves trace paths which are used to guide the placement of a mesh, resulting in road geometry that can be modified with ease by moving control points around.

Figure 2.15: Spline-based procedurally generated race track using cubic Bézier curves, screenshot from Holmér (2015)

Chapter 3

Analysis and Design

In this chapter we present a high-level overview in the structure of the City Generator, and discuss key aspects of each module's functions and overall integration in the software hierarchy.

3.1 Specification

A city generator system is to be designed. Existing methods as well as new proposed methods are to be implemented in a hierarchy according to the problem description. We require the ability to (1) generate a network of streets, and (2) populate the street network with building geometry. The generated geometry shall be decided by some input variables, which influence the output geometry.

Additionally we set the following non-quantizable high level requirements for the design of the city generator system:

- **Modularity:** A high degree of modularity is desirable when generating a city. This can be in the form of separating the generation elements into interchangeable modules or other ways to subdivide the hierarchy.
- **Flexibility:** We want the system to be flexible in the manner that we can generate high variety of content.
- **Scalability:** Methods need to be sufficiently scalable.

With these requirements in mind we propose a module based, iterative generation pipeline. When designing the generation pipeline, we take inspiration mainly from the works of Parish and Müller (2001) and Chen et al. (2008).

3.2 Generation Modules

Much like the works of Parish and Müller (2001), the proposed city generator is structured like a large-scale L-system with several successive modules each responsible for a subset of the generation. However there are some differences to take note of. One of the key differences is that the road network generation is based on hyperstreamline tracing like the works of Chen et al. (2008), and not a self-sensitive L-system.

3.2.1 Structure & Hierarchy

The road network serves as the skeleton of the city and is generated first. Roads are generated iteratively, with several checks for overlap or collision. To connect road segments, a Bézier curve connection algorithm is to be investigated. Upon completion of the road network, the paths are passed on to the building generator. Using the road paths as a framework, buildings are placed on either side of the roads. Checks for building overlap and placement relative to intersections are also performed. The generation order and submodules are illustrated in figure 3.1.

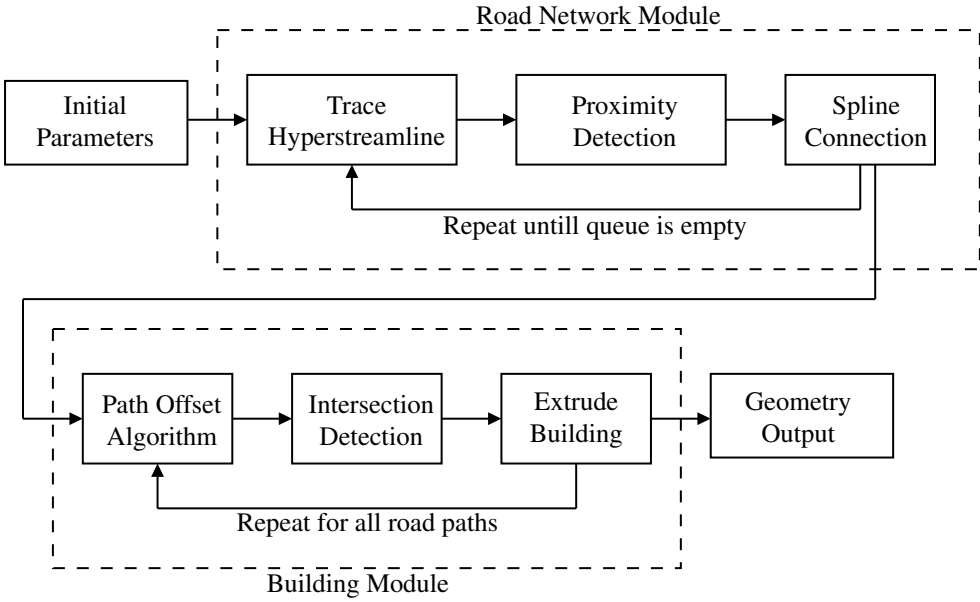


Figure 3.1: Generation modules block diagram

3.2.2 Road Network Generation

To generate the road network, the tensor field tracing techniques described by Chen et al. (2008) were investigated. There are several advantages of hyperstreamlines to trace roads, however the main advantage is the ability to vary the underlying tensor field to get different types of road networks. Another key advantage is that the field can take into account

heightmap data, and place roads accordingly, as described by Chen et al. (2008).

To trace the road paths, an algorithm similar to the methods described by Evans (2015) is proposed. A single seed point is provided as the input, as well as the parameters for the underlying tensor field and the road length. Using a tracing scheme similar to the methods described in section 2.3.1, the hyperstreamlines are traced iteratively as polyline paths through the map one at a time. During this tracing process, new seed points are picked and added to a priority queue. As the first road is completed, the next seed point is popped from the queue, and branching roads are generated in the same manner, except with the opposite (orthogonal) tensor field being used (figure 3.2). This alternating hyperstreamline tracing is repeated until the queue is empty.

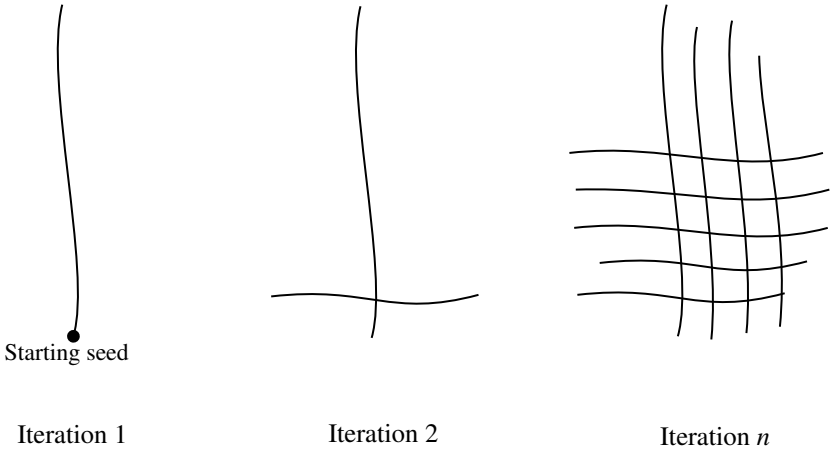


Figure 3.2: Alternate hyperstreamline tracing principle. Each seed point generates roads in either direction with the opposite (orthogonal) tensor field relative to its parent road.

Naturally by just naively generating roads on the basis of a tensor field, several problems may occur. Since a road of length n with m spacing between branching roads would generate approximately $n \bmod m$ additional seed points, this recursive behavior gives a potentially endless number of branching roads. Several proposed solutions to constrain the road generation are introduced. Firstly, a bounds check may be used to stop the generation once the roads reach a certain bounding box in the terrain. As the road generation algorithm iteratively adds new points to the polylines, terminating the generation process can trivially be implemented upon reaching the bounds of the map. Additionally, by introducing a manual limit on seed queue length or number of road generation iterations, we can also decide the number of roads even though there are still potential seed points available in the queue.

Another problem that may arise is that the nature of certain tensor fields create cyclical paths inwards towards a singular point. This may create tight loops that closely overlap each other, which result in less than desirable road patterns. To prevent this, a proximity check may be implemented to ”look” forward in a cone and detect the presence of nearby

roads, in which the generation of that particular road is terminated or connected to the adjacent road. This technique is also used in implementations by both Evans (2015) and Parish and Müller (2001).

In order to connect road endpoints seamlessly to adjacent roads where appropriate, a spline based connection algorithm is proposed. We can therefore also utilize the cone proximity check to check the angle between the road endpoint and the colliding road in order to determine if a spline connection is appropriate. Using splines in this manner is also advantageous because we do not have to take into account the underlying tensor fields (major or minor) of the two particular roads that are to be connected.

3.2.3 Building Generation

As seen in Parish and Müller (2001), the authors generate buildings by extruding a base area and modifying the extruded shape, turning it into a building-like object. Since photorealism is not the main focus of this thesis, we will stick to representing buildings as prism-like primitives of varying sizes. Due to the modular approach, the building generation can easily be extended in this manner to support more realistic building generation.

To generate our buildings, we need to take into account the road shape. At this stage in the generation, the road geometry is already defined as seen in figure 3.1. We can utilize this to generate our buildings, by selecting a subsection of a given road path and offsetting it on either side of the road as a base for the building. By iteratively offsetting each point orthogonally out from the road some distance d, we retain the curvature of the road. This new subpath is further offset once more to form the basis for the bottom face of the building. This principle can be seen in figure 3.3.

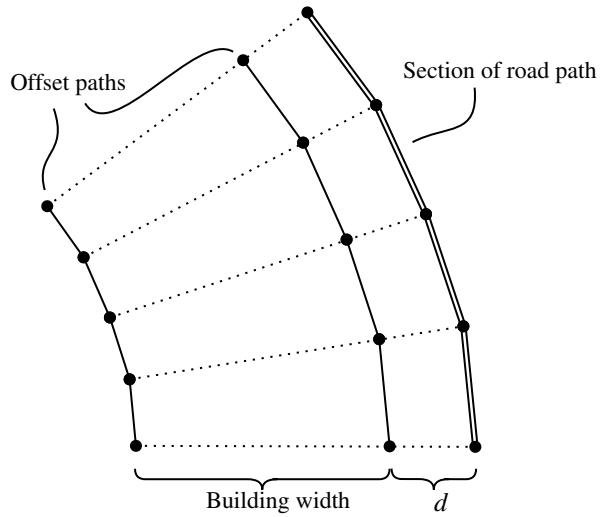


Figure 3.3: Path offset principle forming basis for buildings (top view)

By extruding this face upwards, we can generate a prism-like shape that follows the curvature of the roads, and whose proximal and distal faces are always perpendicular to the road. This is ideal for a dense city as it allows for seamless placement of buildings along paths of varying curvature. In addition by varying the length of the original road path, the second path offset distance and the extrusion height, we can get high degrees of variety in the generation outcome of the buildings despite the simple prism-like shape.

As the road network naturally consists of several intersections along the path, this also has to be taken into account when placing the buildings. One solution is to check each point for neighboring road points before offsetting. If the given road point has > 2 neighbors, it is naturally an intersection of some kind and should be skipped. To ensure that buildings do not overlap the intersection, the building length is cut short before intersection the intersection point, even if the building would otherwise stretch past the intersection point. The next building is subsequently started immediately after the intersection.

Since the road network may be quite dense we might get overlap between adjacent houses. To combat this, a collision detection system needs to be in place. A proposed solution to this is using the built-in physics engine in Unity. As the buildings are populated as 3D-objects in the scene, we can use ray cast methods to see if other buildings are within proximity. If true, the particular building is scrapped and generation is advanced.

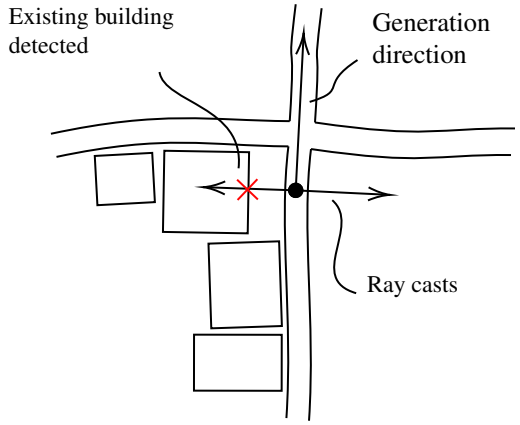


Figure 3.4: Simplified illustration of ray casting collision detection when generating buildings

3.3 Meshing

In order to draw out city geometry, meshing techniques and algorithms are also needed. As the general structure of the generation is iterative and successive, meshing algorithms can be implemented in each step and customized to the different features that need to be meshed.

3.3.1 Road Meshing

When generating the road meshes, a few simplifications can be taken into account. We can think of the roads as a single face on the macroscopic level. Hence, we can use a single flat mesh to draw out each road, and need not take into account the three-dimensional structure of the road profile. However, in order to make the meshing more adaptible and allow for three-dimensional road profiles later on, we will build the road mesh generator with this in mind. This meshing principle is inspired by the procedural geometry techniques by Holmér (2015) and is illustrated in figure 3.5.

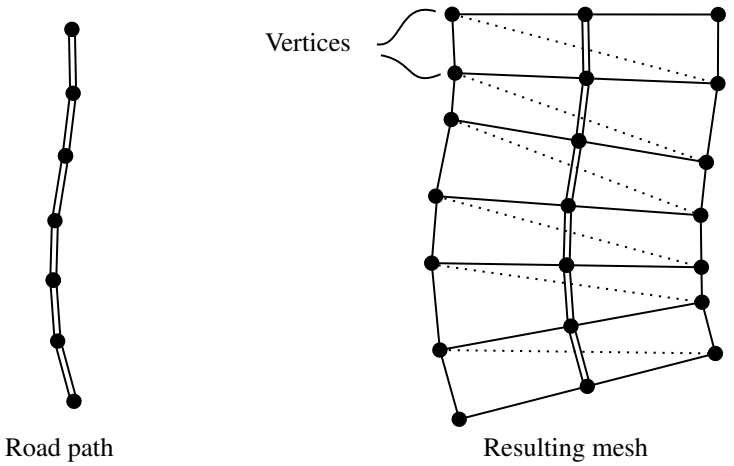


Figure 3.5: Principle behind generating flat meshes representing the road face (top view)

3.3.2 Building Meshing

As mentioned in 3.2.3, the buildings are extruded from the base face. As the buildings naturally need to be represented as three-dimensional meshes, the meshing algorithm for each building is slightly more complex. However, we can implement a modified version of the road mesh algorithm, as each face of the building can be meshed exactly like the road meshes are generated, and then combined to a final convex[1] mesh.

[1] A convex mesh is needed to utilize Unity's collision detection methods.

Chapter 4

Implementation

This chapter gives a thorough overview over the implementation details in Unity engine and key aspects of the methodology used during development. The implementation is given on a high level basis, and assumes an understanding of the C# language and its native libraries.[1]

4.1 System and Software

4.1.1 Platform & Language of Choice

This software project was developed in Unity engine, with C# as the main programming language as specified in section 3.1.

4.1.2 System Specifications

Component	Specification
OS	Windows 10 Pro
CPU	Intel Core i5-4670k @ 3.4GHz
GPU	Nvidia GeForce RTX 2060
RAM	16GB DDR3 @ 1867MHz

Table 4.1: System specifications

[1] For more details on the Unity specific extension libraries, please refer to the Unity API documentation at `https://docs.unity3d.com/ScriptReference/`

4.2 Development Methodology & Structure

As discussed in section 3, the generation hierarchy and associated modules are quite complex and contain many steps. To ensure a robust implementation, the modules are developed in an iterative manner with a goal of each iteration adding features compatible with the existing framework. This way we can not only ensure that basis features are working properly before attempting to add more complexity, but also reduce debugging time on an unnecessarily complex system.

4.2.1 Data Structures

Several data structures were needed to organize the data which define the generated geometry. In addition to basic arrays or matrices, a few custom data structures were also implemented.

Game Objects in Unity

One of the advantages with using Unity engine is the modular game object mechanic explained in section 2.2.1. In this project, the generation modules pictured in figure 3.1 are implemented as a hierarchy of game objects that generate game objects. Each road and building are represented as individual game objects with their parent being the respective generator modules.

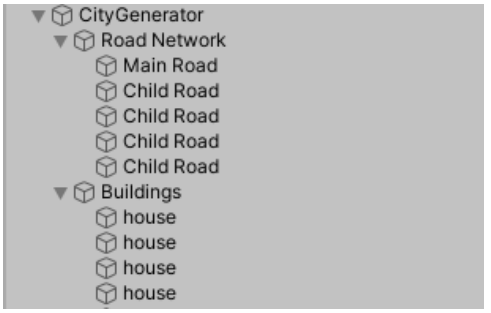


Figure 4.1: Game object structure hierarchy example. Screenshot from Unity engine.

Oriented Points in Space

Since all of our generated items are highly dependent on spatial coordinates, a data structure that stores this information was needed. Building on the works of Holmér (2015), the *Oriented Point* struct was implemented and expanded. This data type is used for all polyline points that define the road paths, as well as the paths that define building geometry.

The advantage of using oriented points over for instance the Vector3 data type is that we can include a *rotation* at a given point in addition to a position. This makes us able

to store information about direction, such as the tangential direction of the road at a given point. Additionally the *magnitude* property was added as a float value. This data field can be utilized in a number of ways, but the main intention was to store a sampled tensor field magnitude at the given point, hence the name. The field magnitude can later be used by the road generator to determine if a local minimum or maximum has been reached. Local maximas can be problematic when tracing the roads. This is discussed in detail in 4.3.2.

To keep track of the neighboring points in the road polylines, the *neighbors* property was also added to the oriented point struct. This way, we can store the nearby points in a list, and also use this lists length to determine if the point in question is an intersection or just a part of a road path.

Lists

The dynamic `List` data structure native to C# was also utilized extensively, namely to store the oriented point paths. The main advantage over static arrays is that implementing flexible algorithms with dynamic lengths is trivial as we need not consider initial array lengths when calculating for instance road paths. Furthermore, the slower speed of lists vs. arrays was deemed justified by the fact that the geometry is only generated once at runtime, and that there are other more significant bottlenecks when it comes to performance.

4.3 Road Network Generator

4.3.1 On Tensor Fields, Scalar Fields and Noise

As described in Chen et al. (2008), the tensor in (2.1) produces eigenvector fields that adhere to the gradient, with the major eigenvector field following the gradient and the minor field being orthogonal to the gradient. However, this is only the case if at a given point $\mathbf{p} = (x, y)$ with direction $[u_x, u_y]^\top$, we define $\theta = \arctan(\frac{u_y}{u_x})$ and $R = \sqrt{u_x^2 + u_y^2}$, as is described in the original paper.

In other words, the tensor field lines are dependent on an underlying *scalar field* and its *gradient*. This scalar field can for instance represent a height map, and thus the resulting eigenvector fields will adhere to the topological features of the height map. In creating the road network, an underlying scalar field was therefore needed to subsequently produce the orthogonal pair of eigenvector fields. To serve as a comprehensive test bed for the tracing algorithms, the Perlin noise function was chosen. Perlin noise is suitable for this purpose as it is continuous, scalable and can allow for interesting "organic" road patterns.

4.3.2 Tracing Tensor Fields

To trace the road networks, two separate functions were implemented; SAMPLE() to sample the tensor field, TRACE() to trace the polyline, which takes SAMPLE() as an argument. This way of using the possibility in C# to pass functions as arguments was done not only to keep the code tidy and readable, but also to adhere to the modular philosophy. This way we can design one function for tracing, which accepts a sampling function, which again can accept one of several functions that describe the actual field.

Quaternions

Before we go into the details of the tracing and sampling functions, we will digress shortly to take a look at rotations. As mentioned in 2.2.1, the rotation of game objects in Unity are defined by the `transform` class. Even though every game object is associated with a transform, the opposite is not necessarily the case. Every transform has a `.rotation` property, that defines its rotation in world space using the `Quaternion` type. This is also the case in the Oriented Point struct, where the rotation is stored as a quaternion. Quaternions in Unity derive from the mathematical concept of quaternion rotations, which is a symmetric vector operation:

$$\mathbf{p}' = \mathbf{qpq}^{-1}$$

Here, $\mathbf{q}$ is a quaternion rotation vector and $\mathbf{p}'$ is the resulting rotated vector. However, in Unity they are implemented to function a lot like rotation matrices:

$$\mathbf{p}' = \mathbf{Rp}$$

The inverse step is effectively abstracted away and handled by the `Quaternion` class automatically. This also means that the operation of rotating a vector in Unity is not commutative, just like rotation matrix operations.

Sampling the Tensor Field

Incorporating quaternion rotations in the SAMPLEORTHOGONAL() function makes it much easier to trace polylines. Consider an arbitrary point on the plane $\mathbf{p} = (x_0, z_0)$. To sample the tensor field at $\mathbf{p}$, we start by calculating the gradient of the underlying scalar field $f(\mathbf{p})$ numerically[2] using the definition of the derivative:

$$\nabla f(\mathbf{p}) = \Big[\frac{\partial f(\mathbf{p})}{\partial x}, \frac{\partial f(\mathbf{p})}{\partial z}\Big] = \Big[\frac{f(x_0 + \delta x) - f(x_0)}{\delta x}, \frac{f(z_0 + \delta z) - f(z_0)}{\delta z}\Big]$$

This vector gives us the direction of the eigenvector field at point $\mathbf{p}$. When defining an Oriented Point for this eigenvector field sample, we can now use the aforementioned point $\mathbf{p} = (x_0, z_0)$ as the *position*, and the direction of the gradient as the *orientation* of the point. Since we want to store the orientation as a quaternion rotation in Unity, we call QUATERNION.LOOKROTATION() and pass in $\nabla f(\mathbf{p})$ as the direction. This method points the rotation in the direction of the input vector, in this case $\nabla f(\mathbf{p})$, and maintains the same up direction as the world coordinate system by default. Additionally, we compute $|\nabla f(\mathbf{p})|$ and assign to the *magnitude* field of the point. This corresponds to R in equation 2.1.

The sample point now contains all the information about the tensor field at the given point. However we have two orthogonal eigenvector fields per tensor field. Instead of having separate calculations for the minor eigenvector field, we can simply rotate the major field by calling QUATERNION.ANGLEAXIS() and rotate 90° along the y-axis.

Tracing Hyperstreamlines & Polylines

In the implementation by Evans (2015), a numerical integration scheme based on RK4 was used. However, in implementing the trace function, we take a simplified approach and implement a method similar to the Euler method. Originally, an RK4 based trace function was planned, but during testing the single stage fixed step Euler method was deemed sufficient and chosen over the RK4. The advantage of this is that RK4 requires four stages per step in the integration process, and is therefore more computationally cumbersome.

To start the tracing, the SAMPLE() function is called on the initial starting seed. This seed is added as a public Vector3 for the class, so that we can dynamically change the first seed in the Unity editor. Note that we do not need to store the seed as an oriented point as the rotation at the seed is determined by the underlying tensor field. In fact, during tracing, all the points $P_0, P_1, \ldots, P_n$ are stored as Vector3, while a separate Oriented Point variable is used to keep track of the sampled point values. Both the current point and the sampled point is updated every iteration. The reason to keep these separate is that we can easily advance in forward or backwards using VECTOR3.FORWARD or VECTOR3.BACK respectively when advancing to the next sample point. These directions however are relative to the main coordinate system, and not the local coordinate system of the point itself. To move relative to the local coordinate system, we multiply with the local rotation, i.e. the rotation obtained from the sampled point.

[2] In our code implementation, the values $\delta x = \delta z = 0.01$ was found to be sufficiently accurate for numerical differentiation.

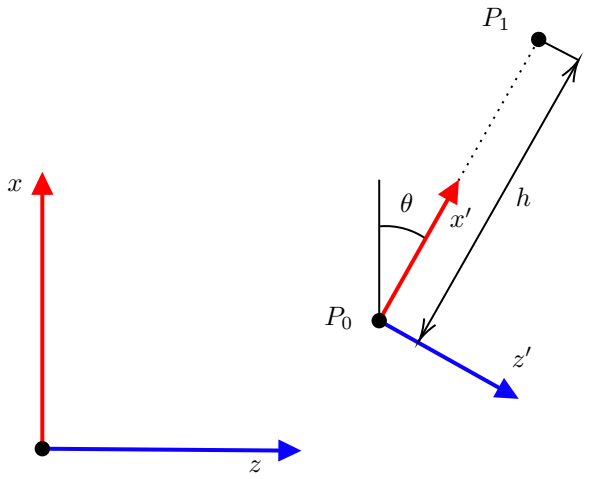


Figure 4.2: World vs local coordinate systems. The next point P_1 is obtained by moving distance h forward (along the local x-axis) relative to the local coordinate system.

The pseudocode in algorithm 1 makes up the TRACE() function. The hyperstreamline H is initialized as a list of oriented points. The current point (xyz-coordinates) are stored in C. As SAMPLE() returns the sampled oriented point P_c, we advance to the next point by method illustrated in figure 4.2 in either the forward or reverse direction and update C. At each iteration, we perform a forward conical search with angle ϕ and search distance R_s to search for existing nearby road paths, and if a collision is detected, we use the spline CONNECT() function to generate the connecting segment (described in detail in section 4.3.3), and add it to H.

4.3.3 Road Interconnection using Bézier Curves

A method of connecting to adjacent roads using cubic Bézier curves was implemented. This function was implemented as a separate `Spline` class. We also take advantage of the list data structure used in the path tracing algorithm here. Whenever the proximity check during tracing returns positive for generating a Bézier connection, we connect the points using a spline. This path is then joined to the hyperstreamline list before returning the completed road path.

In code, the SPLINE class is implemented as several functions. We utilized a cubic Bézier curve similar to the one illustrated in 2.14a, except it is naturally extended into three dimensions. Bézier curves are reliant on linear interpolation, or a *lerp*, as it is often called in context of computer graphics. A LERPORIENTEDPOINT() function was therefore implemented. Using Unity's built in methods, we can lerp both Vector3s and Quaternions natively. The magnitude of the interpolated point m_i is calculated using standard linear interpolation: $m_i = m_a + (m_b - m_a)t$, where $t \in [0, 1]$ is the interpolation parameter. When tracing a road path on the other hand, we are more interested in the *tangential*

Algorithm 1: TRACE

Result: Traces a hyperstreamline from starting point S
Input: SAMPLE(), $S, rev, length$
Output: **P**: the polyline list
$h \leftarrow$ step length
$C \leftarrow S.pos$
P $\leftarrow$ empty list of oriented points
for $i \leftarrow 0$ **to** $(i < length)$ **do**
  $P_c \leftarrow$ SAMPLE(C, m)

  if *(rev)*
    $C \leftarrow$ advance h units in the reverse direction
  else
    $C \leftarrow$ advance h units in the forward direction
  end

  if COLLISION(ϕ, R_s)
    Concatenate **P** with the result of CONNECT(P_c, P)
    return P
  end
  P $\leftarrow$ ADD(P_c)
end
return H

rotation at the interpolated point, as this adheres to the format of the traced road paths. This was solved by simply rotating the lerped point towards the line formed by the ”first layer” of interpolation in the quadratic function, as illustrated in figure 4.3. The rotation is achieved using `Quaternion.LookRotation()`, as in TRACE().

The cubic spline is lastly calculated using calls to the quadratic spline function, adding the last ”layer” of lerps. The control points for the splines were simply calculated by initializing two new oriented points, and then using calls to VECTOR3.FORWARD and VECTOR3.BACK from the start and end points respectively for the position. After some testing, a distance of 20 units from the start and end points was picked as an appropriate length. This corresponds to the distance $P_1 - P_b$ in figure 4.3. Note that both the quadratic and the cubic spline functions only return a single point for any value of t. Therefore, when connecting roads we need to iterate through a series of t values. This is done in the CONNECT() function (algorithm 2).

A fixed length of $l = 13$ steps was utilized after some testing. One drawback with using splines is that while the road paths traced by TRACE() contain uniformly spaced points, the points generated by the spline functions are not uniformly spaced even if the t-intervals are. There are several methods of discretizing Bézier curves to achieve uniform point spacing, however this is not crucial for the macroscopic detail level and was subsequently not implemented in this project.

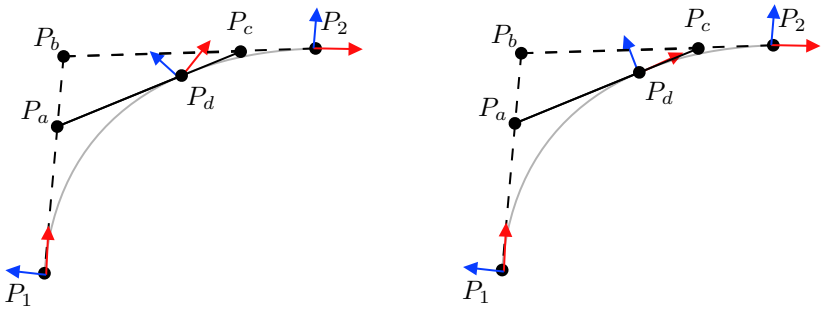


Figure 4.3: Quadratic Bézier curve and rotation alignment. By aligning the rotation at P_d along line segment $P_a - P_c$, the correct tangential rotation is achieved. In this example, P_b is the control point, and P_a and P_c are the points obtained from the first layer of interpolation.

Algorithm 2: CONNECT

Result: Connect two oriented points using a cubic Bézier curve
Input: P_1, P_2
Output: **P**, list of oriented points
$path \leftarrow$ empty list of oriented points
$l \leftarrow$ number of interpolation points
for $i \leftarrow 0$ **to** $(i < l)$ **do**
  $t \leftarrow i/(l-1)$
  $path \leftarrow$ ADD(BEZIERCUBIC(P_0, P_1, t))
end
return P

4.3.4 Mesh Generation

A meshing algorithm based on the generated road paths was implemented according to the techniques described in section 3.5. A simple way to represent the road face is to just offset each point in the road left and right relative to that point's rotation, and then use these points as basis for the vertices in the road mesh. However roads in real life are not simply flat ribbons, even though they may appear as such from a top down perspective. Therefore, a generalized way of describing the road profile was devised. Implementation wise, this is done through the `Profile` struct, which was implemented as a separate file for tidyness sake. The profile of the road is represented as an array of Vector2s, where each vector's x-coordinate represents the vertex distance from the road path, and the y-coordinate represents the elevation in the $y-$direction in world space. This way, we can alter the list of Vector2s and subsequently the road profile without having to modify the

actual meshing logic.

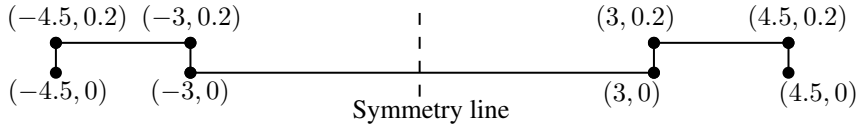


Figure 4.4: Example road profile with a 6 wide road and 1.5 wide "sidewalks" on either side. Vertex coordinates are relative to a central symmetric axis, as this puts the profile centered on the road path.

Calculating Vertex Coordinates

To extrude the mesh, the vertices are first calculated. For every point in the road path, we generate as many vertices as are in the Profile array. To calculate these positions, we simply multiply each vector in the `Profile` array ($\mathbf{p}_p$) with the road points rotation ($\mathbf{q}_r$) and add the road point position ($\mathbf{p}_r$):

$$\mathbf{v}_i = \mathbf{p}_r + \mathbf{q}_r \mathbf{p}_p \tag{4.1}$$

Thus we get the vertex at position i. If the profile array is of length w, we get w vertices per point in the road polyline, e.g. figure 3.5 would indicate a profile array of length $w = 2$ (the road path itself is not part of the vertices forming the mesh). Note that the profile array contains Vector2s, but the vertices of any given mesh is defined by Vector3s. This is solved by simply setting the z-coordinate of $\mathbf{p}_p$ equal to zero, as the array of Vector2s defines a profile in the xy-plane of the world coordinate system. Algorithm 3 describes vertex calculation in detail.

Algorithm 3: CALCULATEVERTEXCOORDS

Result: Calculates vertex positions based on the profile array
Input: $\mathbf{P}, profile, vertices$
Output: $vertices$, array of Vector3s that define the mesh geometry
for $(i \leftarrow 0, v \leftarrow 0$ **to** $(i < l)$ **do**
  for $j \leftarrow 0$ **to** $(j < w)$ **do**
    $vertices[v] \leftarrow \mathbf{P}[i].pos + \mathbf{P}[i].rot \cdot (profile[j].x, profile[j].y, 0)$
    $v \leftarrow v + 1$
  end
end
return $vertices$

The algorithm keeps track of three iteration variables in total: i, the index of the current point in the road polyline $\mathbf{P}$, v, which keeps track of the current index in the $vertices$ array, and j which keeps track of which Vector2 is to be used from the $profile$ array. Every iteration, the $vertices$ array is populated with a Vector3, calculated according to (4.1), and v is incremented. The result is the populated $vertices$ array.

Vertex Winding Order and Triangle Indices

To calculate the correct vertex indices the winding order must be taken into account. Since the profile of the road can contain an arbitrary number of vertices, a generalized method was needed. In general the mesh consists of triangles which can be grouped into squares of two triangles each. For a profile of length w we therefore get $(w-1)$ squares of two triangles, making for $2(w-1)$ triangles per point in the road path. With a path length l we get $2(w-1)(l-1)$ squares along the path. This gives us a total of $6(w-1)(l-1)$ vertices per road path. Note that these vertices need not be unique, as multiple triangles can share a vertex with neighboring triangles.

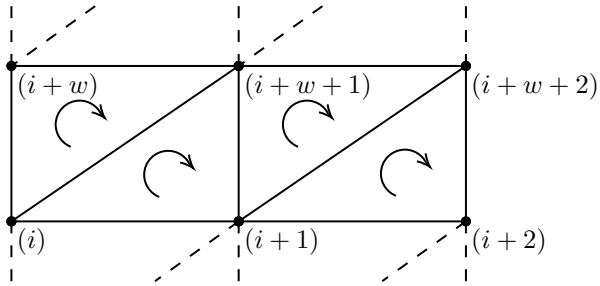


Figure 4.5: Section of the road mesh seen from above. In this example, a profile length of $w = 3$ is used, resulting in 4 triangles and subsequently 6 vertices (black dots) per row in the mesh. Vertex indices are displayed in the parenthesis and the winding order follows the arrows.

This forms the basis for the CALCULATETRIANGLEINDICES() algorithm, detailed in Algorithm 4. The triangle indices are calculated according to the principle illustrated in figure 4.5. Moving in clockwise triangles, indices are added to the $triangles$ array. The vertex iterator i_v keeps track of the specific vertices according to figure 4.5, while triangle iterator i_t keeps track of index position in the $triangles$ array. As the inner loop generates 6 indices, this effectively makes up one square. The triangle iterator i_t is therefore incremented by 6 in the inner loop, and the vertex iterator i_v is incremented by one to form the next square. Lastly, the vertex iterator is also incremented in the outer loop, as we advance to the next row in the mesh. The result is the completed $triangles$ array.

Now that we have algorithms for calculating both the vertex coordinates (3) and the triangle indices (4), we can combine these into the EXTRUDEROADMESH() function (Algorithm 5). Its function is rather simple, we start by creating variables for road width w and path length l, as well as initializing arrays for $triangles$, $vertices$ and uvs[3] with their respective data types. Calls to CALCULATEVERTEXCOORDS() and CALCULATETRIANGLEINDICES() are made, and finally these arrays are assigned to the Unity mesh object.

During testing a simple profile array of length $w = 2$ was used, generating flat and ribbon like roads. This was done to minimize the number of vertices generated, which also speeds up testing. An example of this can be seen in figure 4.6.

[3]Only a basic road texture was implemented, as not much attention was paid to texturing. The uvs array is strictly not necessary to render the mesh.

Algorithm 4: CALCULATETRIANGLEINDICES

Result: Calculates triangles according to the width and length of a road path
Input: $w, l, triangles$
Output: $triangles$ array with correctly calculated triangle indices
$i_t \leftarrow 0, i_v \leftarrow 0$
for $j \leftarrow 0$ **to** $(j < (l-1))$ **do**
  for $i \leftarrow 0$ **to** $(i < (w-1))$ **do**
    $triangles[i_t] \leftarrow i_v$
    $triangles[i_t + 1] \leftarrow i_v + w$
    $triangles[i_t + 2] \leftarrow i_v + w + 1$
    $triangles[i_t + 3] \leftarrow i_v$
    $triangles[i_t + 4] \leftarrow i_v + w + 1$
    $triangles[i_t + 5] \leftarrow i_v + 1$
    $i_t \leftarrow i_t + 6$
    $i_v \leftarrow i_v + 1$
  end
  $i_v \leftarrow i_v + 1$
end
return $triangles$

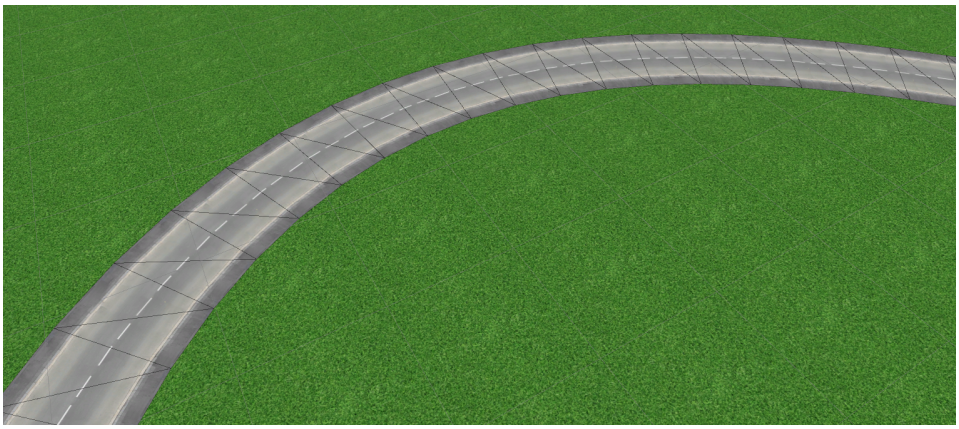


Figure 4.6: Shaded wireframe view of a section of a generated road using a profile length of $w = 2$. A simple road texture is also applied. Screenshot from Unity engine.

Algorithm 5: EXTRUDEROADMESH

Result: The finished road mesh
Input: Path **P**
Output: $mesh$
$w = profile.length$
$l =$ **P**$.length$
$profile \leftarrow$ the desired road profile, Vector2 array
$mesh \leftarrow$ empty Unity mesh object
$vertices \leftarrow$ Vector3 array of size $(w \cdot l)$
$triangles \leftarrow$ integer array of size $6(w-1)(l-1)$
$uvs \leftarrow$ Vector2 array of size $vertices.length$
CALCULATEVERTEXCOORDS(**P**, $profile, vertices$)
CALCULATETRIANGLEINDICES($w, l, triangles$)
$mesh.triangles \leftarrow triangles$
$mesh.vertices \leftarrow vertices$
$mesh.uv \leftarrow uvs$
return $mesh$

4.3.5 Generating the Road Network

Once all the underlying functions for tracing paths and generating meshes were implemented, the main road network can be constructed. This is done in the `RoadNetwork` class, which occupies a game object in the scene with the same name. Two functions were implemented: GENERATEROADNETWORK() to serve as the main iterator for initializing each road, and GENERATEROADS() which is responsible for initializing the tracing process and generating the branching road meshes.

Roads are branched from seed points in an existing road path. With GENERATEROADS() we generate two roads from the seed, one to the left and one to the right relative to the seed points rotation. Opposite roads are achieved by the reverse flag in TRACE(). To ensure these roads branch out perpendicular to the origin road, a flag was utilized to indicate if the major or minor tensor field is to be traced. This flag can be negated every time we generate new roads, and thus generate orthogonally branching roads.

Adding Candidate Seeds

Since the generation relies on a series of seed points, a a FIFO queue was utilized to store the seed points. This queue is kept as a global variable in the `RoadNetwork` class and is modified as roads are being generated. Subsequently a simple function for finding these seed points was implemented. This is achieved simply using iterative modular[4] arithmetic, as seen in algorithm 6.

[4] In our implementation the % operator was used. Technically the %-operator in C# is a *remainder* operator, but since we are only dealing with positive lengths, the function is the same.

Algorithm 6: ADDCANDIDATESTOQUEUE

Result: Seeds added to the master seed queue
Input: **P**, $interval$
for $i \leftarrow 0$ **to** $(i < \boldsymbol{P}.length)$ **do**
  if $(i \bmod interval)$
    $seeds \leftarrow$ ENQUEUE($path[i]$)
  end
end

This way we can space out branching roads using the public variable $interval$, which can be defined in the Unity editor before generating the road network. By using a FIFO queue in this manner, we also ensure that road points are generated strictly in the same order that the roads are generated.

Generating Branching Roads

To generate the branching roads, first we generate two new game objects to serve as the containers for the branching roads. This is done using calls to GENERATECHILDROAD(), which initializes a new game object, assigns it as child to the `RoadNetwork` and attaches the `Road` component. These new game objects are then passed to GENERATEROADS(), which generates the meshes for the branching roads with EXTRUDEROADMESH(). A decrementing iteration variable $iter$ acts as an upper limit for the number of generated roads. This prevents an infinite while loop due to an ever-increasing number of seed points being generated, as discussed in section 3.2.2. This variable was made public to the `RoadNetwork` class in order to make it avaiable in the Unity editor.

Algorithm 7: GENERATEROADNETWORK

Result: Completed road network
Input: $S, iter, length, interval$

CREATECHUNKMATRIX(500)
$R_m \leftarrow$ GENERATEROAD($S, iter, length$)
ADDCANDIDATESTOQUEUE(R_m)
while $(seeds.length \neq 0 \ \ \& \ \ iter > 0)$ **do**
  $seed \leftarrow$ DEQUEUE(SEEDS)
  $left \leftarrow$ GENERATECHILDROAD($seed$)
  $right \leftarrow$ GENERATECHILDROAD($seed$)
  $major = \neg seed.major$
  GENERATEROADS($left, right, seed, interval, major$)
  $iter = iter - 1$
end

4.3.6 Optimizations

Chunk Subdivision

When generating the road paths, we have described and implemented a proximity check for each iteration in each polyline. Naturally, searching the complete list of existing points at every step is an extremely costly way of searching for colliding paths. Suppose a circumstance where the existing road paths consist of N points. The next iteration would then have to calculate $N - 1$ distances and compare them all to the threshold before advancing. This quickly becomes extremely inefficient to calculate when N is very large, especially because we can assume that the majority of the points are not within proximity anyway.

To solve this, a chunk subdivision system was implemented to reduce the search space. The basic concept is that we divide the map into chunks and only search for points in the nearby chunks instead of the entire map. To implement this a matrix where each index position is a list was used. Each sublist contains only points whose x and z coordinates round to the nearest 10. This principle might be a bit unclear, so an example is in order to illustrate the principle. With a map size of 5000×5000 and chunk size of 10×10, we get a $\mathbf{L} \in \mathbb{R}^{500\times500}$ matrix of lists:

$$\mathbf{L} = \begin{bmatrix} L_{1,1} & \dots & L_{1,n} \\ \vdots & \ddots & \vdots \\ L_{n,1} & \dots & L_{n,n} \end{bmatrix}, \quad n = 500$$

Here each sublist $L_{1,1}, \dots, L_{n,n}$ represent one chunk. Suppose now we want to order the point $\mathbf{p} = (1367, 2581)$ to a chunk in $\mathbf{L}$. To find the matrix indices, we utilize integer casting to floor[5] each coordinate to the nearest 10. In our example, this would result in matrix entry $[136, 258]$ meaning chunk $L_{136,258}$. Assuming a map size of 5000×5000, and search radius of 10 chunks, this means we only search 100 units in any direction. Assuming an approximately uniform spread of road points in the map, this results in reducing search space to $\frac{(200-1)^2}{5000^2}$, an over 99% reduction.

Bounds Checking

To cap the generation algorithm, as well as make it easier to implement chunk subdivision the map was capped at 5000×5000 units. Subsequently, any hyperstreamlines that trace outside of coordinates value should be stopped and returned. In this manner a quick bounds check was implemented. The function is simple; we check the coordinates in every iteration, and if either the x- or z-coordinates are outside of the map, we return `true`, otherwise `false`.

Additionally when using a fixed step integration method, local maximas can be an issue. When the tracing function is close to a peak, it may therefore "overshoot" the peak.

[5]This is only the case with positive numbers. We cast the position to integers before rounding, as the spatial coordinates are decimal values (floats) and not just discrete integers. We opted to use int casting for consistency instead of the FLOOR() function, as this would round down when dealing with negative numbers.

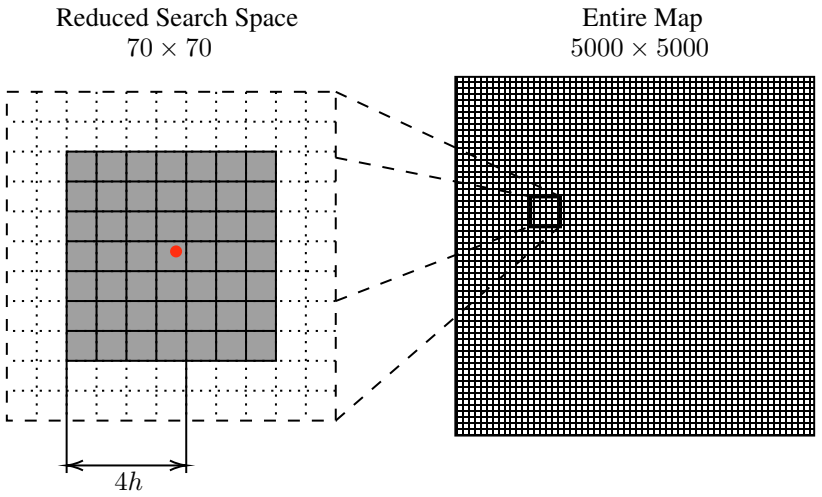


Figure 4.7: Reduced search space as a result of the implementation of chunks. Here a smaller search radius of 4 chunks is used to illustrate the principle.

The new point on the opposite side of the peak will therefore have a rotation in the opposite direction (towards the peak), and will attempt to step back over the peak. But, due to the fixed step, this will only result in another overshoot, and the process repeats. This was solved by including a magnitude check in the bounds check. Recall that the magnitude is calculated as $R = \sqrt{u_x^2 + u_z^2}$, and will therefore approach zero as we get close to a maximum point. By then including a minimum value check of this value, we can detect whether or not we are near a maximum, and avoid the repeated overshooting behavior.

4.4 Building Generator

4.4.1 Curved Buildings

The curved building algorithm was implemented as described in section 3.2.3. The buildings are generated on the basis of the already existing road network. The same list of oriented points data structure was used to define the building geometry during generation. Firstly, since the building shape is based on the road, it makes sense to use the same data structure to have access to the local orientations (rotations) of each point. Secondly, we can extract vertex coordinates from the position element of each point when defining the building mesh, similar to the meshing process of the roads.

Subpaths

Since the road meshes consist of one long list of points defining the path, an algorithm was implemented to extract a subpath. The principle of extracting a subpath is a rather trivial exercise in selecting subelements from a master list. However, there were some considerations that had to be taken into account. A method of detecting intersections along the path was needed in order to prevent buildings from being generated on top of branching roads. A method of detecting colliding buildings was also needed, in order to prevent building overlap. The function is described in algorithm 8.

Algorithm 8: GETSUBPATH

```
Result: Calculates a subpath with intersection detection and building collisions
Input: P, i_s, l, o
Output: Subpath list P_s and the index for the next subpath
P_s ← empty list of oriented points
i ← i_s
while (i < (i_s + l)  &  (i < P.length)) do
    if P[i] has more than 2 neighbors
        return (i + 1), P_s
    end
    if RAYCAST(d)
        i ← i + 1
        continue
    else
        P_s ← ADD(P[i])
        i ← i + 1
    end
end
return i, P_s
```

We iterate through the master list path **P**, starting at position i_s. From here a series of checks are done. Firstly, we check if the current point $path[i]$ has > 2 neighbors. As this means the point is an intersection, the subpath is discontinued at position i, and the next

index is returned. As we have not yet added point $\mathbf{P}[i]$ to the subpath, the subpath consists of points $[P_{i_s}, \cdots, P_{i-1}]$, as P_i is an intersection in this case. For points that are not intersections, ray casting to the left and right is utilized. In our code implementation, the Unity method `Physics.RayCast()` was used for this. This function returns `true` if the ray cast collides with a mesh collider within distance d, at which we skip the particular point at i and advance to the next index. This is repeated for the desired length l, or until the end of the path $\mathbf{P}$ is reached. The result is a subpath $\mathbf{P}_s$ which can be utlizied to form the buildings by successive path offsets.

Path Offsets

The next step in the building generation is offsetting paths to form the building geometry. A total of four paths are needed; two to form the base and two to form the top face of the building. Using combinations of these four paths, we can define each of the six surfaces of the building.

The PATHOFFSET() function (algorithm 9) handles this. Each point in the subpath $\mathbf{P}_s$ is copied to a new list $\mathbf{P}_o$ and offset a distance d left or right relative to the local rotation (along local z-axis), using the same principle as the tracing algorithm in figure 4.2.

Algorithm 9: PATHOFFSET

Result: Subsection of the input path offset by d starting at index i_o
Input: $\mathbf{P}_s, d, l$
Output: $\mathbf{P}_o$ of length l
$\mathbf{P}_o \leftarrow$ empty list of oriented points
for $i \leftarrow 0$ **to** $(i < l)$ **do**
  if *offset to the left*
    $\mathbf{P}_o \leftarrow$ ADD($\mathbf{P}[i + i_o]$ offset d to the left)
  else
    $\mathbf{P}_o \leftarrow$ ADD($\mathbf{P}[i + i_o]$ offset d to the right)
  end
end
return $\mathbf{P}_o$

Another function, OFFSETPATHUP(), based on the same principle as 9 was implemented to offset paths in the world y-direction. The reason to keep these as separate functions was to maintain a positive winding order during meshing. An $orientation$ variable is kept throughout the subpath extraction and offset process, as a means to indicate what orientation the face is going to have. This variable is also used to determine if we offset the subpath to the left or right side of the road in algorithm 9. Naturally, if we would offset paths on either side of the road and try to create a mesh between these paths without taking this into account, we would end up with one face visible from the wrong direction like in fig 4.5, as the winding order would be flipped relative to the camera.

4.4.2 Mesh Generation

To create the building meshes, the algorithms for calculating vertex coordinates and triangle indices are largely based on the same principles described in section 4.3.4. However, there is one key difference: Instead of using a single path and extruding a vertex profile along the path, the mesh vertices are defined between two paralell paths instead. This can be thought of as the construction of a suspension bridge, where the paths are analogous to the two ropes and the triangles comprises the planks making up the walking surface of the bridge. This approach has the advantage that we can use offset paths generated by OFFSETPATH() and OFFSETPATHUP() to generate all six sides of the building independently and then combining them to form the completed building mesh.

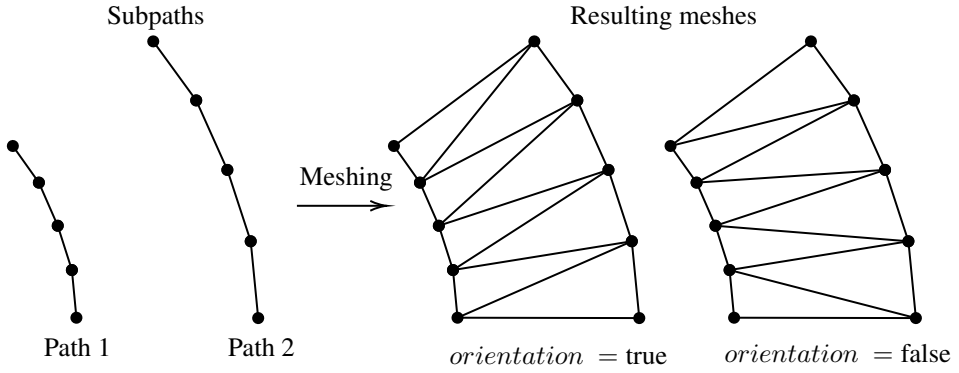


Figure 4.8: Suspension bridge meshing principle between two point paths. Notice that the triangles appear opposite in the rightmost mesh, to keep the winding order clockwise relative to the camera.

Generating Building Faces

To modify the vertex calculating code from section 4.3.4, we first start by altering the algorithm to accepting two paths instead of just a single path. Additionally, we need to pass in the $orientation$ variable. This can be seen in algorithm 10. Note that we do not need to take any rotations into account. Recall that the vertex coordinates are only positions in space (Vector3), and thus we can simply assign the position property of the oriented point to the vertex. Furthermore, the mesh length l is determined from the length of the first path. In fact, we could use either of the two paths, as these lists are always the same size. A vertex index v is used in the same manner as in algorithm 3. Since we have two paths, the final $vertices$ array naturally should contain $2l$ vertices.

Calculating Triangle Indices

Calculating the triangle indices for the building meshes is a slightly more complex operation than for the roads, due to the fact that the buildings are 3D meshes consisting of six faces, while the roads consist of only one continuos face. Since our approach involves adding multiple faces together, we also need to keep track of the index position for every

Algorithm 10: CALCULATEFACEVERTICES

Result: Get vertex coordinates from two paths
Input: $\mathbf{P}_1, \mathbf{P}_2, orientation$
Output: $vertices$ array
$l \leftarrow \mathbf{P}_1.length$
$vertices \leftarrow$ empty Vector3 array of size $2l$
if *orientation*
  for $i \leftarrow 0, v \leftarrow 0$ **to** $(i < l)$ **do**
    $vertices[v] \leftarrow \mathbf{P}_1[i].pos$
    $v \leftarrow v + 1$
    $vertices[v] \leftarrow \mathbf{P}_2[i].pos$
    $v \leftarrow v + 1$
  end
else
  for $i \leftarrow 0$ **to** $(i < l)$ **do**
    $vertices[v] \leftarrow \mathbf{P}_2[i].pos$
    $v \leftarrow v + 1$
    $vertices[v] \leftarrow \mathbf{P}_1[i].pos$
    $v \leftarrow v + 1$
  end
end
return $vertices$

face added, and shift all the triangle indices along the way for each face generated.

To store triangle indices in between each face, a dynamic list was used. We can then later use `List.ToArray()` to convert to an appropriately sized array, as Unitys mesh class does not accept list objects to store triangle indices.

In calculating the road mesh vertices, we ended up with $6(w-1)(l-1)$ indices per mesh. If we take into account that our face meshes always have width $w = 2$, the number of triangle indices needed is reduced to $6(l-1)$. In between the generation of each face, we pass in an index offset. Since the triangle indices give which vertices in $vertices$ to use for a given triangle, we need to offset this by the number of already calculated vertices. When calculating the triangle indices of face $n \in [1, 2, \cdots, 6]$ we must therefore offset the triangle index of face $(n+1)$ by the length of the current vertices array.

Algorithm 11: CALCTRIANGLEINDICES

Result: Calculates triangles according to the width and length of a road path
Input: $l, triangles, offset$
Output: $triangles$ array with correctly calculated triangle indices
$i_t \leftarrow 0, i_v \leftarrow offset$
for $j \leftarrow 0$ **to** $(j < (l-1))$ **do**
  $triangles[i_t] \leftarrow i_v$
  $triangles[i_t + 1] \leftarrow i_v + 2$
  $triangles[i_t + 2] \leftarrow i_v + 1$
  $triangles[i_t + 3] \leftarrow i_v + 1$
  $triangles[i_t + 4] \leftarrow i_v + 2$
  $triangles[i_t + 5] \leftarrow i_v + 3$
  $i_t \leftarrow i_t + 6$
  $i_v \leftarrow i_v + 2$
end
return $triangles$

Combing Building Faces

The last step in the building meshing process is to combine all six faces that comprise the curved building (algorithm 12). Each building is assembled in two stages; A series of lists are initialized to store the offset paths that make up the building geometry, and the paths are offset using PATHOFFSET() and PATHOFFSETUP(). Then the faces are combined by generating and adding vertex coordinates and triangle indices to the $vertices$ and $triangles$ array iteratively per face. Finally, the vertices and triangles of the mesh is updated in the same manner as in EXTRUDEROAD() (algorithm 5).Since offsetting paths relative to other offset paths can become a bit abstract, an illustration of this principle is given in figure 4.9.

When calculating each face, we take in two paths to make up each respective face. I.e. to generate the face facing the road, we utilize the path closest to the road $\mathbf{P}_1$ and its upwards offset $\mathbf{T}_1$. To calculate the top face, we utilize the two top paths $\mathbf{T}_1$ and $\mathbf{T}_2$, and so on. Note that there are only four offset paths, but we have six faces. Naturally we use the same path multiple times when generating faces, like $\mathbf{T}_1$ is shared by both the top face and the roadside face. However, this requires a total of 8 paths to generate the whole building. This is solved by taking into account that the front and back faces can be made up of either end points of the four main paths. This way we can generate these faces by passing in the appropriate end points as paths of length $l = 2$.

It should be mentioned that the use of the $orientation$ variable in CALCULATEFACEVERTICES() could be left out, as it is really up to a correct implementation of the triangle indices calculation to get a correct winding order regardless of vertices order. However, this was done as a conscious design choice. In flipping the vertex coordinates in this manner, we can use the exact same way to calculate the triangle indices for either

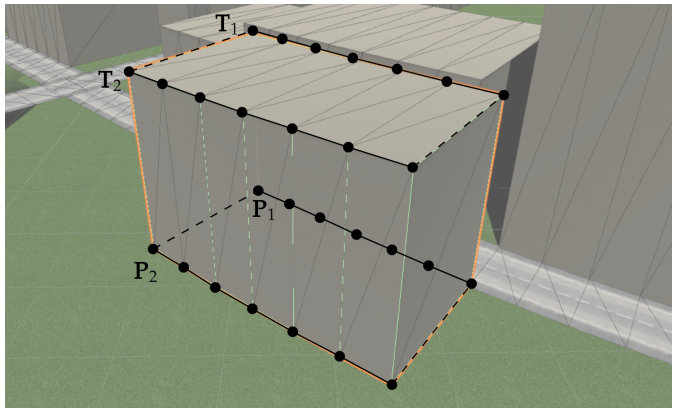


Figure 4.9: An example of the generated buildings using the meshing principles described in section 4.4. The offset paths described in algorithm 12 are highlighted for clarity. The stapled lines represent the ”paths” from the first and last elements of the 4 main paths that define the front and back faces. Shaded wireframe screenshot from Unity engine with overlayed graphics.

face orientation and still end up with the correct winding order. When we in addition need to keep track of a triangle index offset, this helped reducing the complexity of the already complex task of calculating triangle indices for multiple successive faces.

Placing Buildings

To generate buildings along the entire road network, the PLACEHOUSE() function was implemented as a means to iterate over the road network and place houses along the paths. The function takes in all the game objects of the road network, iterating over every path. The paths are subsequently subdivided and offset according to the tecniques described in algorithms 8, 9. Meshes are generated according to the CURVEDHOUSE() function (algorithm 12). Since this function is largely based on Unity specific methods for adding components, a detailed pseudocode is not given and we refer to the source code for details.

Algorithm 12: CURVEDHOUSE

Result: Generate building faces and combine into the final mesh
Input: $\mathbf{P}_s, d, w, h, orientation$
Output: $mesh$ object
$mesh \leftarrow$ empty Unity mesh object
$\mathbf{P}_1, \mathbf{P}_2, \mathbf{T}_1, \mathbf{T}_2 \leftarrow$ initialize as empty lists
$\mathbf{P}_1 \leftarrow$ PATHOFFSET($\mathbf{P}_s, d$)
$\mathbf{P}_2 \leftarrow$ PATHOFFSET($\mathbf{P}_1, w$)
$\mathbf{T}_1 \leftarrow$ PATHOFFSETUP($\mathbf{P}_1$)
$\mathbf{T}_2 \leftarrow$ PATHOFFSETUP($\mathbf{P}_2$)
for *each of the six faces* **do**
    $vertices \leftarrow$ ADD(GENERATEVERTEXCOORDS($length$, comb. of path lists)
    $triangles \leftarrow$ ADD(CALCTRIANGLEINDICES($length, triangles, offset$))
    $offset \leftarrow vertices.length$
end
$mesh.vertices \leftarrow vertices$
$mesh.triangles \leftarrow triangles$
return $mesh$

4.4.3 Optimizations & Improvements

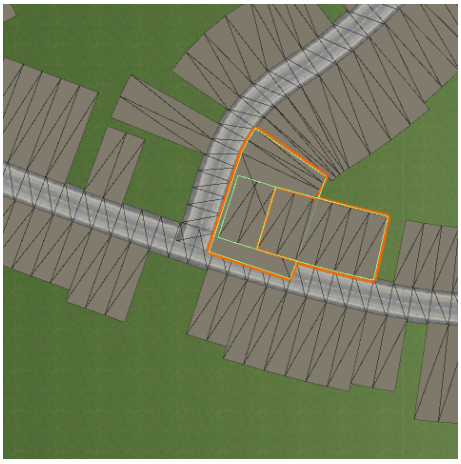
Overlap and Collision Detection

As we are generating the buildings solely based on the road paths generated, the ray casting method described in section 3.2.3 was implemented as a means to avoid building collision. A *mesh collider* component was added to each building on generation. By setting the mesh of the generated buildings to a *shared mesh* instead of a regular *mesh* object, we can use the same mesh in both the mesh renderer and the mesh collider. The function of a shared mesh is otherwise similar to a normal non-shared mesh object.

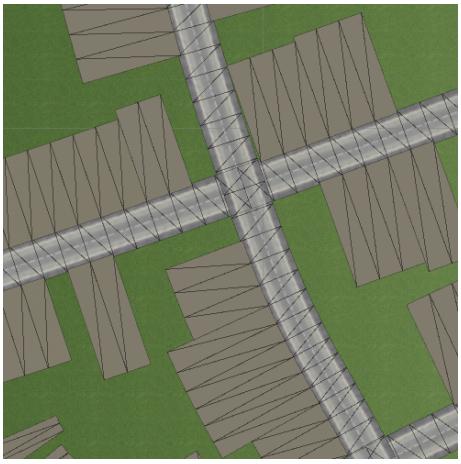
Since we generate buildings as we go along a road path, we can check for building collisions along the way during the path offsetting process. The ray casting is done some height h above each road point, but was set lower than the lowest allowable building height as to not accidentally ignore short buildings. The rays are subsequently cast orthogonally out a distance d to either the left or right direction, based on the $orientation$ variable. This way, computation time is saved as it is not necessary to check the right side of the road for collisions when generating the left side buildings, and so on.

Building Sizing

By varying the lengths of the subpaths generated from the road paths as well as the various offset distances, we can greatly vary the building sizes. This was implemented using an array of height values, which we pick from at random. This way we can distribute the heights of buildings to get more smaller buildings and a few tall skyscrapers like in a medium city, or more taller buildings and a few smaller ones like for instance the city



(a) Bad overlap detection. Overlapping meshes highlighted.



(b) Good overlap detection with no colliding meshes.

Figure 4.10: Shaded wireframe view of two generated intersections. In **(a)** the raycasting failed to detect the neighboring building and a building was placed on top of the neighboring ones. Screenshots from Unity engine.

center of a large city like New York where skyscrapers are abundant.

Some building were also skipped completely to simulate empty lots which are frequently found in urban environments. This was implemented simply as a check that skips the generation and advances to the next subpath if a certain threshold is reached, using a random number as the skip rate r. This has the added benefit of effectively acting as a population density parameter, as we can spread out existing buildings with empty areas should we want to generate a less dense city environment.

With these two techniques in addition to the seed point spacing when generating the road networks, we can generate a large variety of urban environments from densely populated large cities of skyscrapers to more scarcely populated ”rural” environments.

4.5 City Generator

A parent class to the generator functions was devised to start the generation. As the generator functions essentially function on a stand-alone basis, the function of this `CityGenerator` class is simple.

Game objects are initialized and set as children to `CityGenerator`. The `RoadNetwork` and `BuildingGenerator` scripts are attached as components to these game objects, and GENERATEROADNETWORK() is called. Next, the buildings are generated using GENERATEBUILDINGS(). The result is the generated city with road network and buildings placed along the roads.

A custom GUI editor was also implemented for the `CityGenerator` game object. This allows us to access the public variables in the class, which again influences the outcome of the city generator. Parameters include:

- **Starting Point**: The world coordinate seed for starting the generation.
- **Iter**: The number of roads to be generated. This decides the overall size of the city by placing an upper limit on the number of roads that can be generated.
- **Length**: The length of each road that is traced.
- **Offset**: Parameter that defines the offset of the underlying noise field. Since Perlin noise is pseudorandom, we can offset each coordinate using this parameter to get visually different noise fields.
- **Scale**: Scales the noise field. Higher values give less curvature in the road network.
- **Interval**: The number of points between seed points, effectively the distance between individual branching roads.
- **Skip Rate**: Percentage chance of skipping a building during generation, spacing out buildings along the road network.

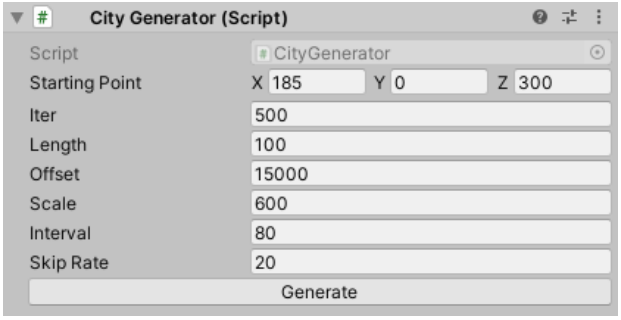


Figure 4.11: Example of initial parameters for starting the generation process.

Chapter 5

Results

In this chapter we present the results by giving various examples of the capabilities of the city generation techniques implemented in Unity engine.

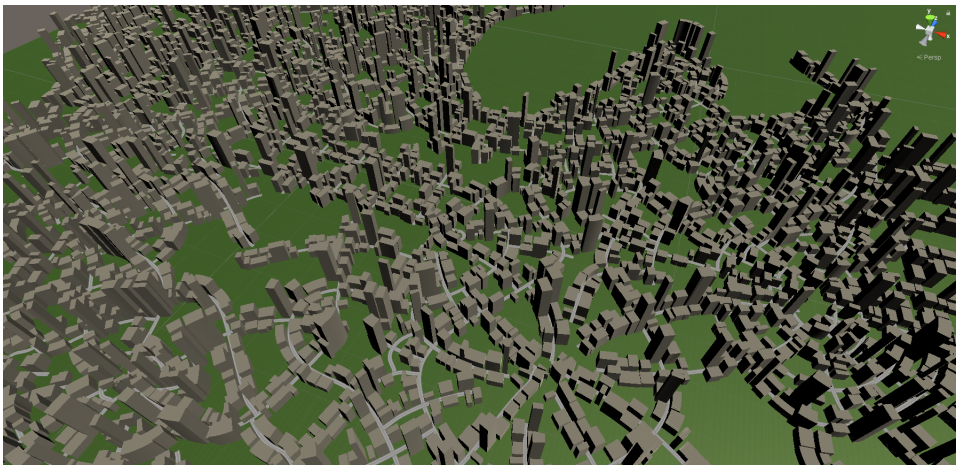


Figure 5.1: An example city generated by our system where the initial parameters from figure 4.11 are used. Screenshot from Unity engine.

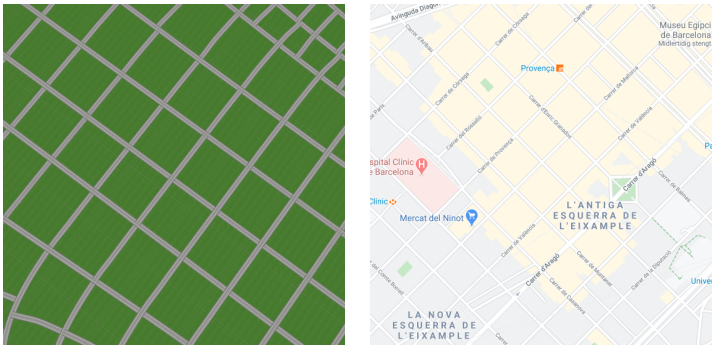
5.1 Generation Results

The implemented city generation system is capable of generating complex city environments. High variety in both the road network and the buildings generated was achieved, but the system requires refinement in order to incorporate additional functionality.

5.1.1 Strengths

Road Network Generation

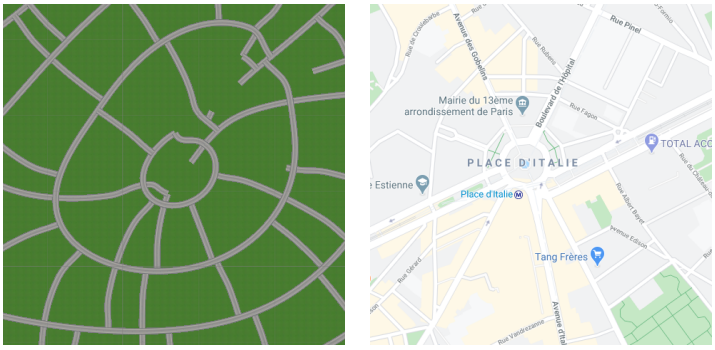
On a macro scale, the road network generation is overall quite satisfactory. The nature of the underlying Perlin noise field gives rise to a number of interesting road patterns, which was handled well by the tracing algorithms. Examples of generated road patterns can be seen in figure 5.2 and figure 5.3. We have areas in the road network that greatly resemble road patterns found in real life cities.



(a) Area of square city-like grid. Screenshot from Unity engine.

(b) Square street grid in Barcelona. Screenshot from Google Maps.

Figure 5.2: Square grid features compared to the city grid of Barcelona.



(a) Area of a circular road pattern. Screenshot from Unity engine.

(b) Place d'Italie, Paris. Screenshot from Google Maps.

Figure 5.3: Circular road features generated by our system compared with similar features found in Paris.

Another advantage inherent to using hyperstreamline tracing is that we can vary the underlying field while keeping the overall tracing principle the same, as noted by Chen

et al. (2008). This is also the case with our system, and results in high degrees of freedom when generating road networks.

The modularity of the system is also demonstrated by the simplicity of changing the underlying input field. As the Perlin noise field can theoretically be replaced with any scalar field, a simple linear gradient on the form

$$f(\mathbf{p}) = x + z \tag{5.1}$$

was also tested. By replacing the Perlin noise function with the field in (5.1), the result was a completely orthogonal street grid. An example of this can be seen in figure 5.4.

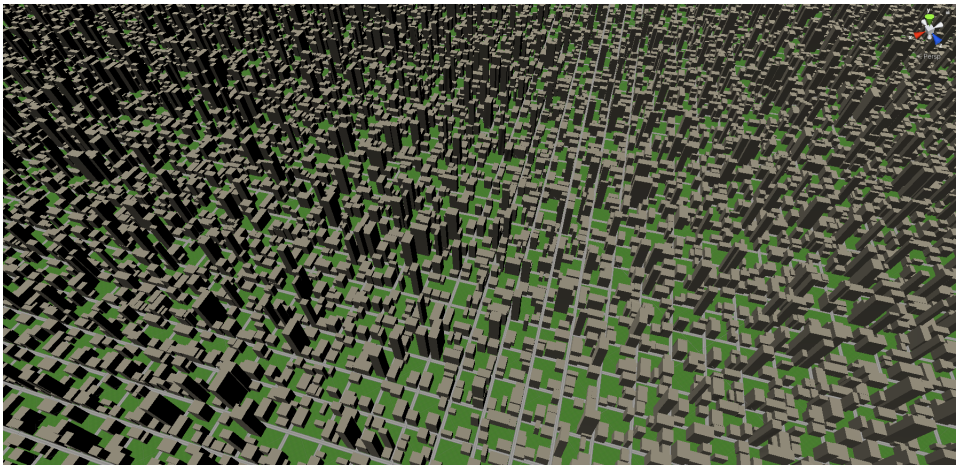
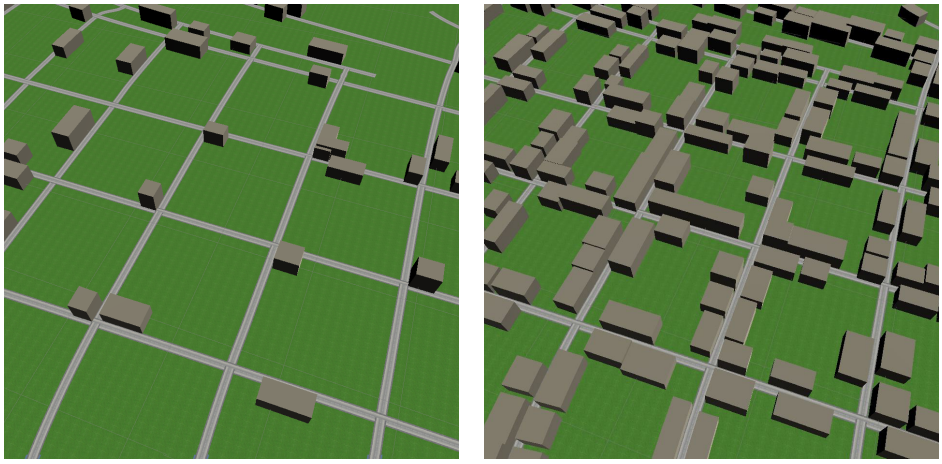


Figure 5.4: The resulting generated city when using a linear gradient field $f(\mathbf{p}) = x+z$. Screenshot from Unity engine.



(a) High skip rate ($r = 0.8$)

(b) Low skip rate ($r = 0.1$)

Figure 5.5: Differences in building skip rate with the same road network parameters. Lower skip rate results in more densely placed buildings, and this parameter can thus be adjusted to account for population density.

Building Generation

We also achieved high variety in the buildings generated by the system. By varying the sizes and placement frequency of the buildings, we can achieve different types of cities. This is mainly controlled through the branching interval and overall size of the road network, but the parameters for skip rate and building height is independent of the road network. Different combinations of these parameters allow for a wide variety of outcomes.

For instance, if we are modeling a densely populated city, we might want to have taller buildings packed more closely together. On the other hand, we can have the opposite effect by allowing a high skip rate and high branching interval for a suburban area. The results show that we can simulate a number of different types of city areas using the building skip rate and height distribution. This in turn can be used to simulate many kinds of urban environments, from densely packed downtown-like areas, suburb neighborhoods and even more rural areas where buildings are scarce. Example of skip rate differences can be seen in figure 5.5.

Since the system, being developed in Unity engine, is mainly targeted towards game development, the size of the generated game world is also an area of interest. The system, as shown in figure 5.7, is capable of generating extremely large cityscapes.

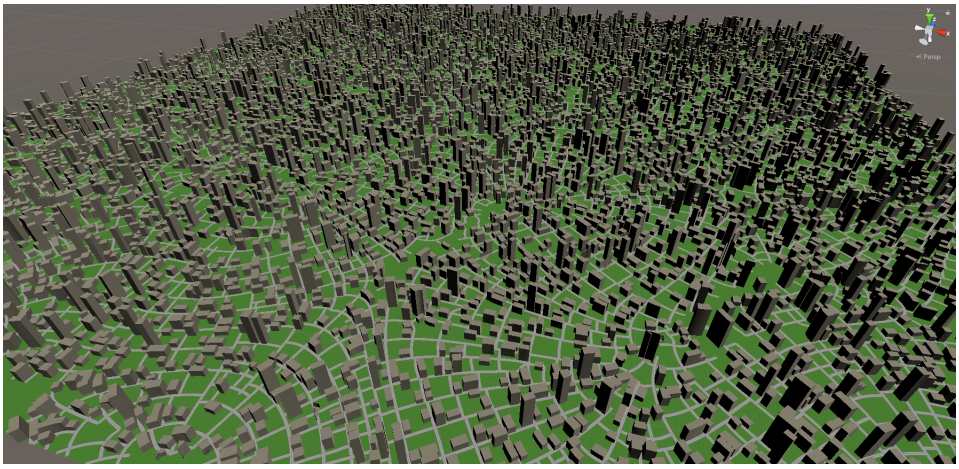


Figure 5.6: Low skip rate, low branching interval, simulating a densely populated large city. Screenshot from Unity engine.

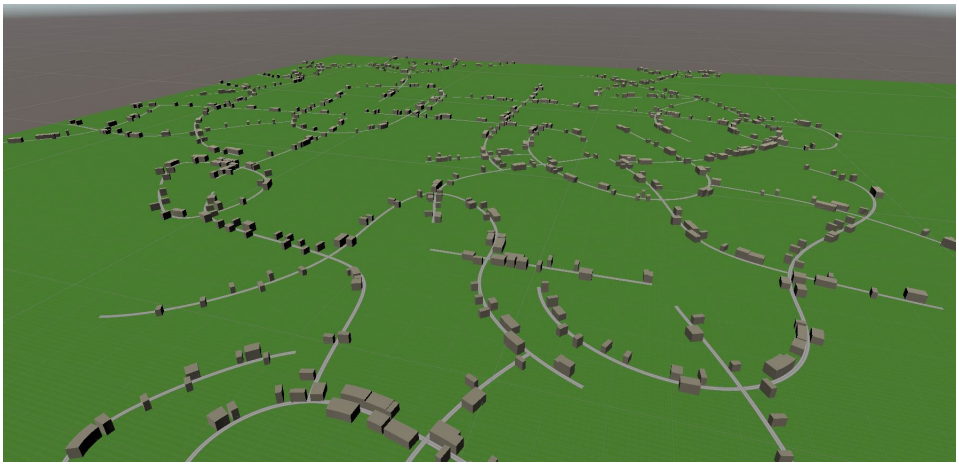
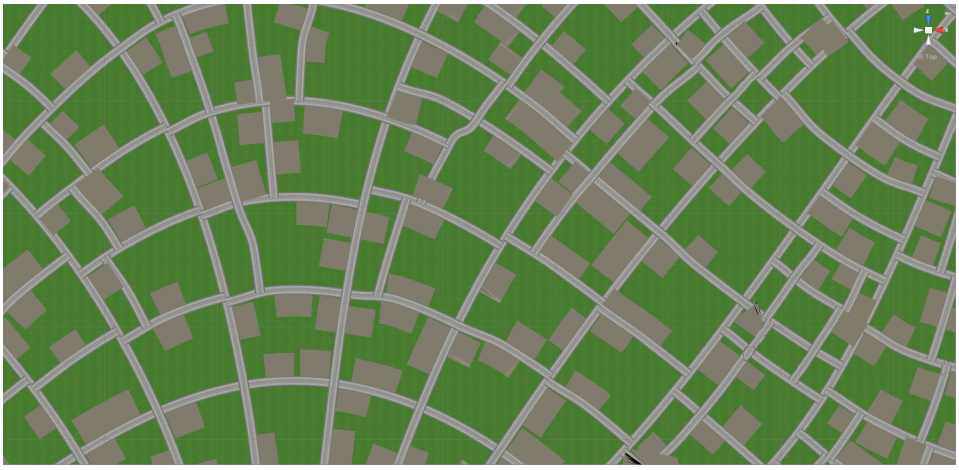


Figure 5.7: High skip rate, high branching interval and low building height to model an area of lesser population density. Screenshot from Unity engine.

5.1.2 Limitations

Currently our system only generates a single road network uniformly across the whole map. As real world cities contain different types of roads, the generator is not able to generate an accurate road network model. For a more detailed road map the implementation of different road classes such as highways, city streets and smaller private roads should be included.

(a) Scale of 200



(b) Scale of 1200

Figure 5.8: Top down comparison of noise field scale differences. Screenshot from Unity.

With smaller intervals between the road intersections (and thus denser road networks), the fixed length spline method started misbehaving. This also translated to bad building

meshes of the attached buildings generated from this part of the path. Examples of this can be seen in figure 5.9. Furthermore, at higher scale values in the noise field the splines misbehaved more frequently. This is evident in figure 5.8(a). This is a direct result of the splines control points being dependent on the traced road segments, which means at lower scale values (and thus higher curvatures), bad spline connections is observed. A solution to this can be to implement a variable length spline calculation, or investigate other methods of connecting the roads.

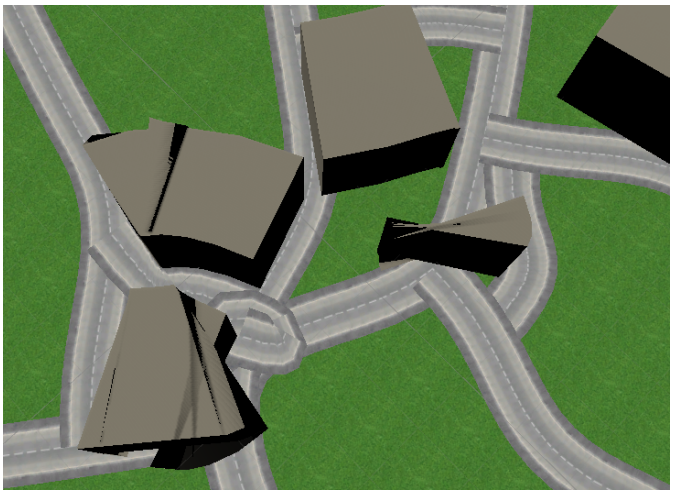


Figure 5.9: Mesh artifacting from badly generated spline connections. Screenshot from Unity engine.

As is evident in many of the figures displaying the generation of the road network, the intersections are not very realistic. The overall goal of this project was mainly to investigate methods of generation and therefore a proper method of merging intersecting meshes was not implemented. This combined with the occasional misbehavior of the spline connection algorithm means the low level detail of the generated geometry is not adequate.

Another issue that was found is that the ray casting collision algorithm tends to give false positives in some cases. This can be seen in some areas where the buildings are separated by one step length, where with proper behavior we should have buildings stacked closely together at very low skip rates. We get narrow alleys between buildings even in densely populated areas, however it was not an intended feature of the system. This does however correspond to some features found in real life cities, and the feature was subsequently kept.

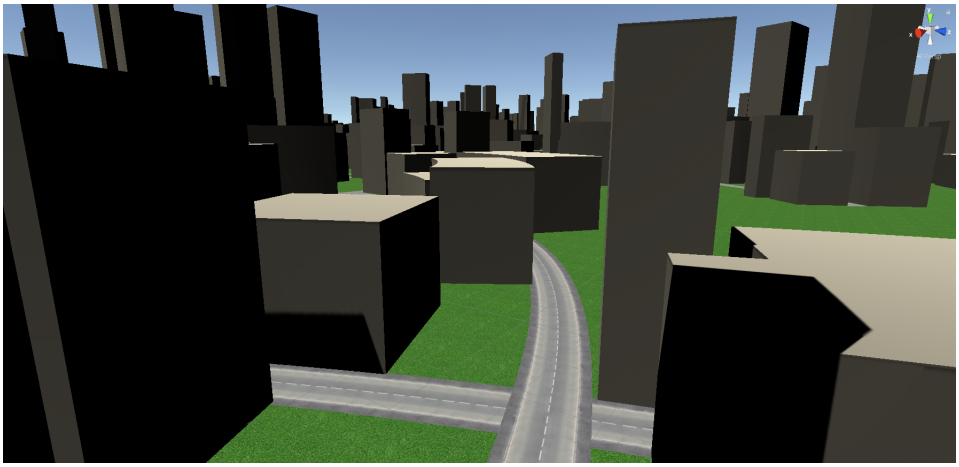


Figure 5.10: Example showcasing the (lack of) low level detail produced by the system. Screenshot from Unity engine.

5.1.3 Performance Evaluation

A series of tests were run to evaluate the generation time. Both road network generation, building generation and building generation with collision detection was tested. The results can be seen in table 5.1. The tests suggests that the most time consuming part of the generation is tracing the road network. Another observation is that the generation time scales approximately linearly with the number of roads generated. This is a favorable result, as it suggests the system is highly scalable.

	Generated Objects		Generation Times (ms)		
Iter	**Road Points**	**Buildings**	**Roads**	**Buildings**	**Total**
100	4363	448	98	92	190
500	12831	972	308	206	514
2000	38824	2752	1048	612	1660
5000	88884	6097	2365	1416	3781
10000	145980	8476	4540	2061	6601

Table 5.1: Generation benchmark of increasing city sizes. Cap on the number of branching roads (**iter**) is shown in the leftmost column. A skip rate of $r = 0.2$ and road length of 200 was used during these tests. Tests performed on the system in table 4.1.

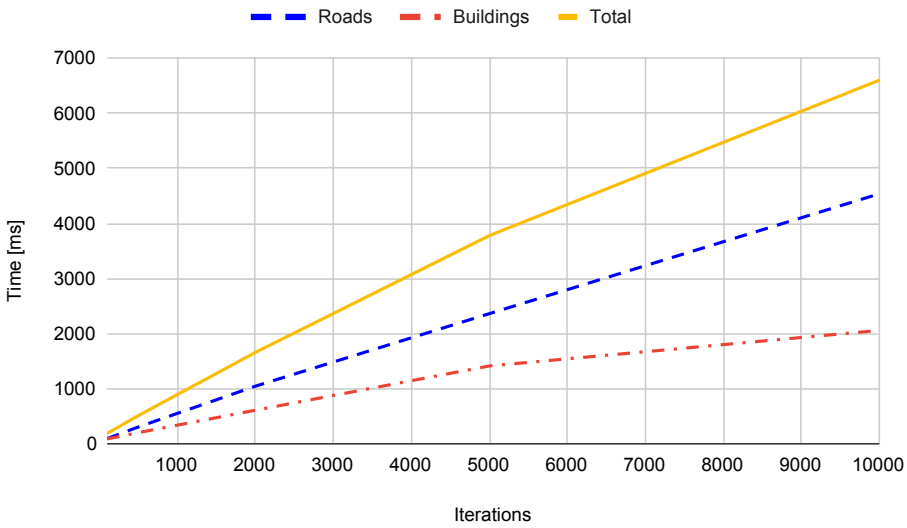


Figure 5.11: Runtime vs. number of tracing iterations. Data plotted from table 5.1.

Chapter 6

Discussion

In this project we have investigated and implemented methods of procedurally generating an urban environment in Unity engine. Along the way several choices had to be taken which led to challenges. In this chapter we discuss and evaluate our findings, and point out the significance of our findings.

6.1 Evaluation

6.1.1 Road Network Types

An extensive part of this project has been the design and implementation of the road network generator. Early on in the design phase when deciding on a technique to trace the road networks, hyperstreamline tracing was picked due to the intriguing results showed by Chen et al. (2008) and their interesting solution to tracing non-linear road paths. The tracing techniques were subsequently already cut out through previous works. We did however find some differences from existingly described tracing methods.

One thing to point out is that despite using a simpler integration technique and a fixed step method, our results were visually similar to the hyperstreamlines in Chen et al. (2008). This suggests that a fixed step method can be as viable as more accurate and more computationally cumbersome methods like RK4. This finding is significant in a game development application, as real time generation of the road networks may be of interest and naturally performance is a key factor.

Comparing our results to (Chen et al., 2008), another difference is that we used a single underlying scalar field to generate our tensor field lines instead of blending various fields. This was mainly done as the specific shape of the generated geometry was second in priority to the flexibility of the tracing methods. However, this naturally placed restrictions on the appearance of the generated city, as a single field makes the road network somewhat uniform throughout on a large scale generation. On the other hand, the extra time spent

ensuring the modular design of the network generator and trace functions makes the addition of these techniques entirely possible.

On the use of Perlin Noise

In this project we used a Perlin noise field to serve as a testbed for the road generation. To our best knowledge, this has not been done before. Using Perlin noise in this manner shows promise for interesting looking road networks. The results from using only a single noise field mimics the blended fields of (Chen et al., 2008) surprisingly well, with the drawback that the placement of the various featured road types (square neighborhoods, circular road places, etc.) is not customizable and is inherent to the noise function. Another finding that was found notable is that we were able to generate highly varied city environments with only a few parameter changes.

The overall findings suggest that using Perlin noise is viable for generating the road networks both for a densely populated city, as well as areas of lesser population density. Despite the limitations of using a single noise field, the use of Perlin noise be an interesting area of further research.

Bézier Curves in Road Generation

Another difference to (Chen et al., 2008) is the addition of road connection using Bézier curves. While the overall road connections generated was satisfactory according to specification, the results were suboptimal. In many cases, the connecting road paths did not form proper four-way intersections, which we would expect to see in a real city. This is also due to a flaw in the the overall data structure of the road network, discussed in detail in section 6.1.3.

6.1.2 Performance

A few interesting results were discovered when testing the performance of the system (table 5.1). The generation of the geometry was found to have a linear time complexity. This was rather surprising, since when increasing the number of iterations, we effectively populate a larger area in the two dimensional plane. Intuition would indicate that such an algorithm would have a polynomial time complexity, as filling twice the area would require four times the amount of filler and thus take four times as long to complete. But let us step back and take a look at what is happening during tracing. We do not in fact fill an area but rather trace one dimensional lines across a plane. Because of the superposition principle, multiplying linear operations results in a linear operation, hence the tracing algorithm can be assumed to have a linear time complexity as well. This suggests that the performance of the tracing algorithm is indeed linear to the number of iterations, and performance is more sensitive to other variables such as the branching interval or length cap.

Surprisingly, the building generated took less time to generate than the road network, despite having a more complex algorithm. This consideration alone suggests that the ray casting is a viable option for game development. However it was found to be unreliable in other areas, namely that we did get a number of false positives when detecting collisions along the way. This may be a result of the rays cast from point i intersecting with the previously finished building at point $(i - 1)$, giving a false detection and subsequently giving the "alley" features described in the previous section. In order for this method to be more viable, further optimizations need to be done to make the method more accurate.

6.1.3 Data Structures

In one area, our implementation differs from Chen et al. (2008). While the overall way of representing paths as lists of oriented points worked well, the lack of organization of these paths proved to be an issue. In the original paper they represented the road network as a graph while in our project we generated each road path independently, with no proper graph structure outside of the *neighbors* property. This caused two issues. Firstly, it did not allow us to traverse the graph properly. Traversing a graph with cycles would be beneficial, as we could use cycle detection algorithms to define city blocks, and subsequently use a lot subdivision algorithm similar to the ones described by Evans (2015).

Secondly, by having independent, unorganized paths like this, making proper intersections proved difficult, and no method for generating intersections was implemented. Since we have individual game objects for each road mesh, mesh merging algorithms may be of interest to improve intersection generation.

6.1.4 Building Generation

On a general note the path offset and extrusion method proved to be an efficient way of generating buildings. The advantage of this method is that the buildings follow the road geometry. With a road network involving curves, this results in interesting building generation. We were also able to model a variety of different building shapes using only a few simple parameters determining the path offsets.

However in some areas, the building generator did produce some results of lesser quality. While an efficient method along straight or slightly curved roads, the path offset method did not cope well with sharp turns. Combined with the occasional bad behavior of the spline connection algorithm, some less than desirable results were seen. This suggests that basing the buildings solely on the tangential direction of each point in the road does not accurately reflect buildings in real life. As a result the building generation algorithm should be revised with specifically this in mind.

6.2 Reflections

This project has been a learning experience on many different levels. Although overall satisfactory results were achieved there are some reflections to be made, there were many challenges to tackle along the way. The problem of procedurally generating city environments is not a trivial task. Many figurative balls were kept in the air at once. Translating existing methods and implementing and adapting these to Unity engine proved challenging exercises in both programming skill, planning and logical thinking.

In retrospect, the distribution of time duing the implementation phase proved difficult, with many areas being potential time sinks along the way. A lot of time was spent implementing and testing the road network generator, much of which was due to an underestimation of the amount of time it would take to develop a working prototype. This resulted in an uneven time distribution in other areas, such as the building generator and overall polish of the software. However, the extent of the time spent on developing the road tracing algorithms did not prove only negative. Great care was taken to make the software design of the tracing methods highly modular and robust, which would not have been possible with ad-hoc solutions. This allows for relatively easy integration of extension methods and additional features, which could allow for more interesting road pattern generations.

Another moment of underestimation, as discussed in section 6.1.3, was the lack of organization when it comes to the data structures. We arguably should have spent more time researching and creating a robust data structure as a base for the for the road network. Although not obvious at the time, this would likely have made it easier to implement the building generation algorithms, and might have allowed for several algorithms to be explored. On a positive note, a takeaway from this is that we exposed a key insight into developing procedural software: namely that a bad foundation may create limitations or unnecessary complications down the road.

Chapter 7

Conclusion

In this thesis we have investigated, developed and implemented a city generator in Unity engine. The development was focused around two main areas; tracing the road network and populating the road network with buildings. A hyperstreamline tracing algorithm from existing papers was introduced, and new methods for extruding buildings along the road paths was devised.

Results indicate that the implemented system shows promising capabilities. Great variety was achieved with only a few input parameters, which is highly desirable in procedural generation. We were able to simulate a number of different urban and suburban environments by varying the density of the road network and frequency of placed buildings. The use of Perlin noise produced interesting results, comparable to both existing work and real life city environments. Connecting nearby road segments with spline connections worked to some extent, but was limited by the fixed length spline implementation in some areas. A key insight that was discovered is the importance in the road network structure. We can therefore conclude that in order for the tracing methods to be a viable option for generating city geometry, a more organized data structure forming the foundation of the road network should be considered.

When it comes to assessing the systems overall viablity in game development, the system does not constitute a finished product and more development needs to be done. Lower level complexity as it stands now not adequate for generating whole game worlds. We note that procedural generation is often used to quickly generate a rough outline for a game world that is later manually adjusted, and in this manner the system performs well. The overall macroscopic detail level was found to be acceptable, and the system serves as a foundation for further work and development. Notable areas of improvement includes lower levels of detail, texturing and the addition of more features and code optimizations.

7.1 Further Works

A number of improvements for the city generation system should be considered moving forward. Many routes of further research exist, and we may consider different paths depending on what is deemed important for the future of this system. One path may be improving the lower level detail, adding intersections, more detailed buildings and texturing. Another path may be to increase the variety of the generated system even more, and introduce more parameters to give more freedom in the generation process.

7.1.1 Key Improvements

Most importantly the data structures should be revised. The road network functions as the skeleton of the generated geometry, and might be the most important part of the system. By introducing a graph based road network, we may improve both the road network generation techniques and meshing techniques. In this regard, graph traversing algorithms to discover cycles may be utilized to outline city blocks.

Another improvement we find imperative for the future of this system is generating road intersections. This problem is related to generating the mesh geometry and the overall structure of the road network. A possible area of investigation can be aimed towards generating the street graph as a whole first and then meshing the entire graph in one sweep instead of iteratively meshing each independent road segment. Another option may be extending the independent road path meshing techniques and utilize merging algorithms to merge the meshes at intersection points.

When it comes to placement of buildings, this is also a feature that may see improvement from better utilization of the generation plane. If the areas bounded by road paths can successfully be described and mapped out alternative building generation algorithms such as the lot subdivision method used in CityEngine could be of interest. This could possibly also open up the option of populating city blocks with other urban elements like parks or recreational areas. The curved building generation algorithm can also be explored further, adding more rules and restrictions to place more realistic buildings.

In the same manner that we utilize tensor fields to map out the road networks, we may use similar fields to describe other features in the terrain or city environment. Height maps could be utilized in combination with the noise field to create elevation in the terrain that corresponds to the road network, and vice versa. The use of population density maps could be utilized in the same manner to implement a dynamic branching interval to simulate varying population densities over a distance. Similar techniques could be applied to other parameters such as the building heights, sizes or even whole road networks. By classifying road network types, we may be able to for instance model both a large down town area and the surrounding suburbs with a natural transition in between.

7.1.2 Optimizations

A key aspect when it comes to game development, and especially for real time applications, is the performance of the system. During the implementation stage of this project, code optimization was not an immediate concern. Although certain optimizations were done such as the chunk subdivision and building collision detection methods, these were absolutely necessary in order to make the implementation feasible. Several areas in the generation hierarchy could be subject to performance optimization. For instance, the generation of individual game objects may not be necessary when it comes to the road network. Collecting the entire road network under a single game object may prove more efficient at runtime, and reduce generation time. Parallel processing methods may be investigated to utilize threading when calculating the geometry.

Bibliography

Bhadury, K., 2017. LSystem — GitHub. https://github.com/kbhadury/LSystem, Online; accessed 2020-04-1.

Birkhoff, G., de Boor, C.R., 1964. Piecewise polynomial interpolation and approximation. Approximation of Functions, Elsevier Publishing Company , 164–190.

Cash, J.R., Karp, A.H., 1990. A variable order runge-kutta method for initial value problems with rapidly varying right-hand sides. ACM Transactions on Mathematical Software 16, 201–222.

Chen, G., Esch, G., Wonka, P., Müller, P., Zhang, E., 2008. Interactive procedural street modeling. ACM Transactions on Graphics 27, No. 3, 301–308.

Delmarcelle, T., Hesselink, L., 1993. Visualizing second-order tensor fields with hyperstreamlines. IEEE Computer Grapichs & Applications , 25–33.

Delmarcelle, T., Hesselink, L., 1994. The topology of symmetric, second-order tensor fields. Proceedings Visualization , 140–147.

Dustler, M., Bakic, P., Petersson, H., Timberg, P., Tingberg, A., Zackrisson, S., 2015. Application of the fractal perlin noise algorithm for the generation of simulated breast tissue. Progress in Biomedical Optics and Imaging - Proceedings of SPIE 9412. doi:10.1117/12.2081856.

Egeland, O., Gravdal, T., 2003. Modeling and Simulation for Automatic Control. Marine Cybernetics.

Evans, M., 2015. Procedural generation for dummies: Road generation. https://martindevans.me/game-development/2015/12/11/Procedural-Generation-For-Dummies-Roads/, Online; accessed 2020-02-12.

Holmér, J., 2015. A coders guide to spline-based procedural geometry, in: Unite Conference, Boston, USA. https://docs.google.com/presentation/d/

10XjxscVrm5LprOmG-VB2DltVyQ_QygD26N6XC2iap2A/edit, Online; accessed 2020-02-26.

Hosier, A., 2016. Voronoi tessellation in dart. https://github.com/adamhosier/voronoi, Online; accessed 2020-04-27.

Kutz, P., 2012. Computer graphics by peter kutz. http://peterkutz.com/, Online; accessed 2020-04-01.

Lindenmayer, A., 1968. Mathematical models for cellular interactions in development. Journal of Theoretical Biology 18, 280–299.

Massive Software, 2020. Massive Software - Simulating Life. http://www.massivesoftware.com/index.html, Online; accessed 2020-03-29.

Parberry, I., 2015. Modeling real-world terrain with exponentially distributed noise. Journal of Computer Graphics Techniques 4.

Parish, Y.I.H., Müller, P., 2001. Procedural modeling of cities. ACM SIGGRAPH , 301–308.

Perlin, K., 1985. An image synthesizer. SIGGRAPH Computer Graphics 19, 287–296.

Prusinkiewicz, P., Lindenmayer, A., 1990. The Algorithmic Beauty of Plants. Springer.

Thomas, K., 2011. Perlin noise in javascript. https://asserttrue.blogspot.com/2011/12/perlin-noise-in-javascript_31.html, Online; accessed 2020-04-02.

Zhang, E., Hays, J., Turk, G., 2007. Interactive tensor field design and visualization on surfaces. IEEE Transactions on Visualization and Computer Graphics 13, No. 1, 94–107.

Appendix

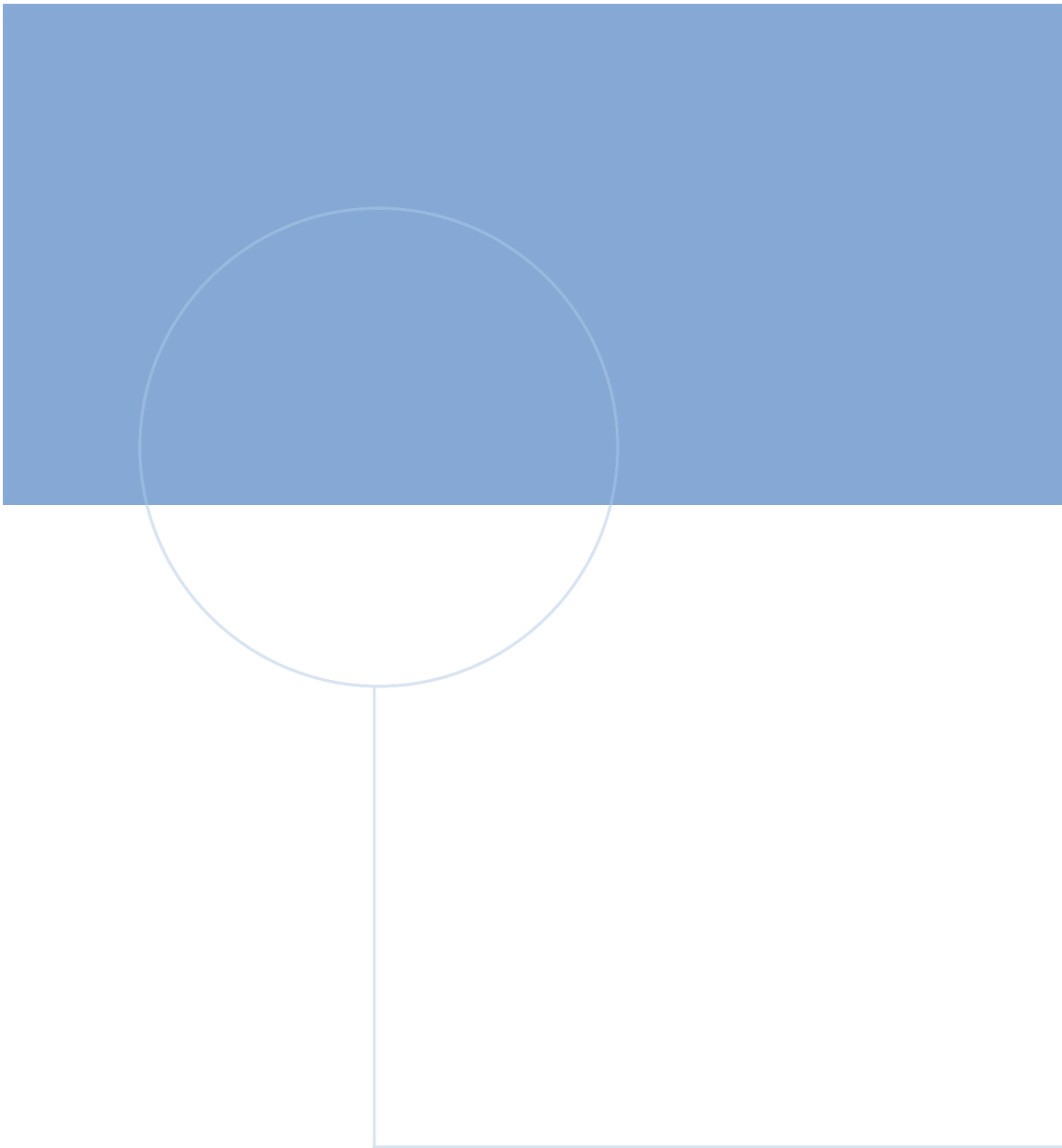
The code for this project can be found at `https://github.com/davidbmadsen/CityGen`

Installation Instructions

- Download and install Unity engine from `https://unity3d.com/get-unity/download`
- Clone the GitHub repository to a folder
- Open the cloned project as a Unity project

To start the generation, highlight the *CityGenerator* gameobject in the Hierarchy tab to the left in the Editor. The parameters for the city generator described in section 4.5 is found in the *Inspector* pane on the right with the *CityGenerator* game object highlighted.

Press the *Generate* button in the inspector to generate the city. Note that for high **iter** values, the generation may take a few seconds.



NTNU

Kunnskap for en bedre verden