Chunk-Based Procedural Generation of Urban Environments

MSc Dissertation

Gvidas Danielius
MSc Game Engineering
Newcastle University, School of Computing

ABSTRACT

Procedural Content Generation (PCG) is an essential tool for creating vast, explorable worlds demanded by modern video games and visual effects. While PCG for natural landscapes is well-established, generating performant, believable and seemingly infinite urban environments in real-time presents a significant challenge. Much of the existing research focuses on generating finite cities with global knowledge, leaving a gap in research for performant, chunk-based methods suitable for real-time applications.

This dissertation investigates and evaluates chunk-based algorithms for procedural city generation. It aims to determine the feasibility of producing compelling and performant cityscapes by comparing a novel, organic generation method against more conventional approaches. To achieve this, three distinct algorithms were developed and analysed: a conventional agent-based road generator to serve as a baseline; a simple grid-based chunk generator for high performance; and an organic generator that uses Voronoi diagrams to create more varied and natural urban layouts. A modular pipeline was also implemented to translate the 2D road networks from these systems into 3D geometry, including roads and buildings.

The algorithms were comparatively evaluated using quantitative performance metrics and qualitative analysis for their visual output. The results demonstrate that while the agent-based method produces complex, organic road networks, it scales poorly for larger city sizes. The grid-based chunk system proved highly performant and scalable but resulted in visually monotonous cityscapes. The Voronoi-based generator successfully addressed this limitation, producing significantly more varied and believable layouts with only a modest increase in computational cost, confirming its viability for real-time use.

This research concludes that chunk-based procedural generation is a feasible approach for creating infinite urban environments. The developed systems provide an array of balances between aesthetic quality and performance, and the best choice is dependent on the specific use-case of the systems.

INTRODUCTION

1.1 Context

Procedural Content Generation (PCG) has been a cornerstone of various fields such as Game Development and VFX for decades. Since the development of Perlin noise in the early 1980s, its application has expanded drastically, now enabling the creation of vast, intricate worlds on a scale previously unachievable by hand. The landscape has undergone dramatic expansion in these past decades, where players now expect vast, immersive worlds that offer extensive opportunities for exploration and interaction. Titles such as *No Man's Sky* and *Minecraft* have demonstrated the profound appeal of these game worlds, however this demand for scale presents a significant challenge to more traditional content development as designing these worlds can be labour-intensive, time-consuming and as such a prohibitively expensive process.

PCG has emerged as a powerful tool for addressing this challenge. By leveraging excess computation time, developers can generate quantities of content that are not feasible to create by hand, fostering larger worlds and more immersive gameplay.

Within the broad field of PCG, the generation of urban environments represents a unique and compelling problem. Unlike natural landscapes, cities are a product of both deliberate planning and organic growth. Their shapes are defined by a complex interplay of social and economic factors that combine to create a unique experience within each. The structure and logic of cities make them interesting subjects for procedural generation, demanding an approach that would balance believability with efficiency.

This project focuses specifically on a chunk-based/block-based approach to procedural urban generation. This decision is twofold. Firstly, from a performance perspective, generating a world in discrete and manageable chunks is critical for real-time applications. It allows for the dynamic loading and unloading of content based on the player's location, preventing the need to generate an entire city in situations where only a small portion is ever needed. Secondly, performance-oriented chunk-based city generation lacks much exploration, which creates a clear opportunity for novel research. This project focuses specifically on developing the ability to create infinite urban landscapes by employing different methods of chunk generation, as well as refining these methods.

1.2 Aim

The primary aim of this project is to investigate and evaluate different chunk-based algorithms for procedural city generation. The effectiveness of a novel approach will be compared against a more traditional road algorithm to determine the potential for producing compelling, varied and performant cityscapes.

1.3 Objectives

To achieve this aim, the following objectives will be pursued:

1. To research and analyse existing techniques in PCG that focus specifically on generating urban layouts, road networks and chunk-based world generation.
2. To develop a conventional, rule-based road generation algorithm to serve as a baseline for comparison. This system should be capable of producing plausible road networks influenced by variable parameters.
3. To design and prototype a novel, chunk-based generation system that explores an alternative approach to defining primary chunks and the subsequent placement of internal road networks and blocks.
4. To create a method for rendering the 3D geometry of road networks.
5. To establish and apply a set of evaluation criteria to comparatively analyse the outputs of developed systems.

BACKGROUND AND RELATED WORK

2.1 History of Procedural Content

The underpinnings of modern PCG can be traced back to certain foundational algorithms that allowed for the creation of controlled randomness and organic-looking complexity. Ken Perlin's development of Perlin Noise [2] was one of these and is considered a seminal moment, providing a method for generating gradient noise that became the industry standard for creating natural textures for surfaces, fire and smoke.

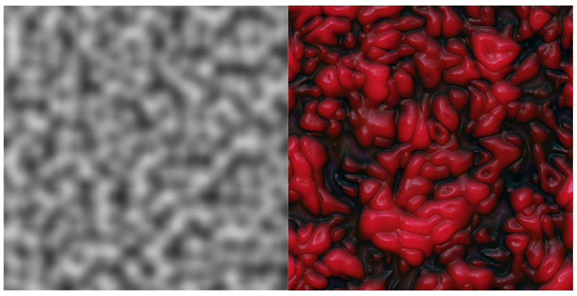

Figure 1: Perlin noise, and an organic surface generating using Perlin noise [10]

Subsequent refinements, such as Simplex noise in 2001 offered similar results with greater computational efficiency with fewer directional artifacts. These noise algorithms form the basic building blocks upon which more complex procedural systems, including those for urban generation, are built upon.

2.2 Modern Motivations and Applications

The technological landscape of content generation has changed dramatically since the 1980s. The modern surge of PCG is a response to a production bottleneck, as modern games are held to higher expectations by players and as such the demand for massive, high-fidelity and open world games has risen. Manually creating content on the scale of certain modern titles is labour-intensive, and in some cases completely infeasible.

The modern motivations for using PCG are multifaceted, focusing on enhancing replayability, fostering emergent gameplay, and enabling personalised experiences. Products such as *Starfield* and *Shadow of Mordor* demonstrate these motivations. *Shadow of Mordor*'s Nemesis System procedurally generates unique enemy hierarchies that dynamically react to the player's actions, creating personal experiences. Similarly, *Starfield* leverages PCG to initially generate over a thousand planets, populating them randomly with premade structures, which provides a vast galaxy for exploration on a scale impossible to create manually.

2.3 Procedural Urban Modelling

While the principles of PCG have been successfully applied to a wide range of content types, the generation of urban environments presents a unique and compelling set of challenges. They are fundamentally different from the generation of natural landscapes, which are often characterised by chaotic, fractal patterns and can be effectively modeled using techniques based on noise functions and recursive algorithms. Cities, on the other hand, represent a complex interplay between deliberate planning and organic growth.

There are several approaches to take when dealing with urban environments, each capturing a different aspect of a city's structure. These can broadly be categorized into the following three main paradigms.

2.3.1 Rule-Based Approaches

Among the earliest and most influential methods for procedural modelling are grammar-based systems, most notably Lindenmayer systems (L-Systems). Originally designed to model the growth of organic systems such as plants, they were adapted for urban environments by Kelly and McCabe in their work on "CityEngine" [3]. In this model, an initial starting road is iteratively rewritten according to a set of rules, generating a branching network of streets. This approach excels at creating highly structured, grid-like patterns reminiscent of planned cities but can struggle to produce the more organic and irregular layouts found in older, naturally grown urban centers.

2.3.2 Agent-based and Growth Models

An alternative approach involves simulating the growth of a city from the bottom up. Agent-based models, or space-colonisation algorithms, simulate the behaviour of individual agents that lay down roads based on local information. For example, an agent might be programmed to extend a road until it intersects with another road. Agents can be combined with templates and road patterns to influence the shape of the resulting city, as done by Jing Sun et al. [5] who use a combination of "growing" agents and density maps to create interesting and natural road networks.

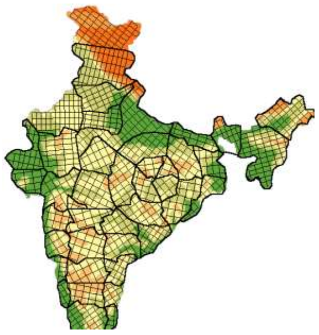

Figure 2: Road network generated by Jing Sun et al. [5]

2.3.3 Spatial Partitioning and Subdivision

A third approach focuses on the top-down subdivision of space. The area designated for the city is recursively partitioned into smaller regions, which become city blocks, parcels and lots. Techniques like quadtree subdivision are common but may result in rigid layouts, more organic patterns may be achieved by using Voronoi diagrams.

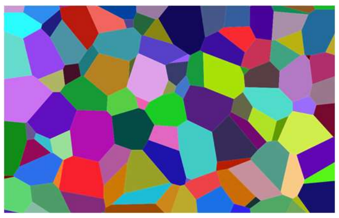

Figure 3: Voronoi diagram of a set of points [11]

Voronoi diagrams were originally defined in 1908 by Georgy Voronoy and represent the tessellation of a plane and a set of seeds, where each point in a region is closest to the same seed. The resulting partitions of this diagram can be used to generate polygonal cells that serve as natural-looking city blocks. The dual of this diagram, the Delaunay triangulation, can then be used to aid with generating the resulting network.

2.4 Real-Time Generation and Chunking

The demand for vast, explorable worlds in modern games has necessitated techniques for managing computational and memory load. Generating an entire city, let alone an infinite one, is not feasible for real-time applications. The industry-standard solution is chunk-based world generation, famously employed by games like *Minecraft*. The game world is divided into a grid of fixed-size chunks, and only chunks within a certain radius of the player are rendered

2.5 Research Gap and Justification

While previously mentioned paradigms for city generation are well-established, a significant portion of the academic research focuses on only generating finite and static cities. Early work by Greuter et al. [6] presented a viable approach to the real-time generation of pseudo infinite cities, yet this was achieved only by constraining the generation to a regular Manhattan-style grid.

Many existing methods, particularly those producing the most detailed and structured results, rely on global knowledge of the entire city map such as height and population density maps and is a luxury not available when generating a world one chunk at a time. The challenge of integrating complex urban generation algorithms into performant, chunk-based methods remain as a gap in literature.

This project directly addresses this gap. By developing and comparing a traditional growth-based algorithm against a novel chunk-based system, this research aims to explore the trade-offs and potential synergies between different approaches within the strict constraints of a real-time, chunk-based algorithm.

2.6 Further Examples of Urban Generation

Several key papers have also advanced the field by proposing and developing novel techniques. The work by Parish and Müller [1] is an important example, establishing a complete pipeline for city generation. Their system utilizes L-systems to procedurally grow extensive road networks from a set of rules, demonstrating how a grammar-based approach could generate the entire arterial structure of a city.

Addressing the specific issue of building facades, Wonka et al. [4] presented a system for this use case using stochastic shape grammar, their method recursively subdivides a building's mass into detailed architectural elements such as floors, walls and windows.

Further developing the process of street creation, Chen et al. [7] introduced a highly interactive system. It allows a user to define a high-level road network and employs a tensor field to guide the generation of smaller, local streets. This method creates road patterns that flow as well as the user sets them, with road patterns being filled within the user-defined city blocks, representing a powerful hybrid of user control and procedural subdivision.

IMPLEMENTATION

3.1 Engine Considerations

After understanding the main paradigms of Procedural Urban Generation, the implementation of a chunk-based algorithm could begin. A key consideration at the project's outset was the choice of development environment. As the research is centered on the procedural generation algorithms themselves, creating a custom engine was determined to be outside the scope of the project. A small test of each leading game engine was conducted. Initial tests with C++ OpenGL revealed that the overhead associated with managing low-level graphics programming was out of scope. Unreal Engine was also considered for its powerful rendering, however, was set aside as its complex architecture was not conducive to the iterative development the project required.

By the end, Unity emerged as the optimal choice. Its robust C# support, extensive documentation and a well-established ecosystem, combined with my existing experience on the platform, provided the necessary tools for developing this project effectively.

3.2 Conventional Road Generator

I began by developing a conventional benchmark for comparison and decided to use an agent-based growth model. This approach was selected as a well-established and frequently used method within existing research, providing a strong foundation for evaluating the chunk-based systems developed in this project.

In the implementation, the essential idea is to keep track of a set of active agents called "head nodes" which travel based on given rules, setting "road nodes" behind as they do so. The simulation progresses in discrete time steps, or "ticks", during which the generator iterates through the set of head nodes and applies the rules to it.

An agent's primary function is to extend its road segment forward, creating new nodes that form the vertices of the road network graph. To create a connected network and avoid unrealistic overlaps, each agent must be aware of its local environment. A simple approach requires each agent to check its proximity against every other node in this graph, an $O(n^2)$ operation that is computationally infeasible for large networks. To solve this, a quad tree partitioning system is employed. An agent may query the quad tree to find nearby nodes efficiently. If a node from a different branch is detected within a specific radius, the agent connects to it to form an intersection and terminates its own growth.

To introduce more organic and varied layouts, two factors influence an agent's behaviour. Firstly, a degree of randomness is applied to its direction at each step, resulting in less uniform, meandering roads. Secondly, the system is influenced by a global density map. This allows parameters, such as the probability of an agent branching to create a new road, to be modified based on the agent's location, simulating the transition between dense urban cores and sparser suburban areas.

3.2.1 Quad-tree implementation

Before the main road generator can begin, a quad tree data structure is first initialised to store all road nodes. This structure is defined by the `QuadTree` class, which serves as a public interface, storing the root `QuadNode` object, as well as any global information needed such as the width of the city space, and the maximum number of road nodes allowed within a `QuadNode`**.**

The core logic resides within the **QuadNode** class, which represents a single square region of space and is considered a "leaf" node – a node that has no children. The node remains a leaf and stores road nodes directly until the number of nodes exceeds a specific capacity, `maxRoadNodes`. Upon reaching this threshold, the `SplitNode()` method is triggered which subdivides the node into four equally sized children exactly a quarter the size and redistributes its stored road nodes among them, after which the parent node acts purely as a branch within the quad tree. This grow-as-needed- approach is highly efficient for procedural generation as the tree's detail adapts dynamically to the density of the road network.

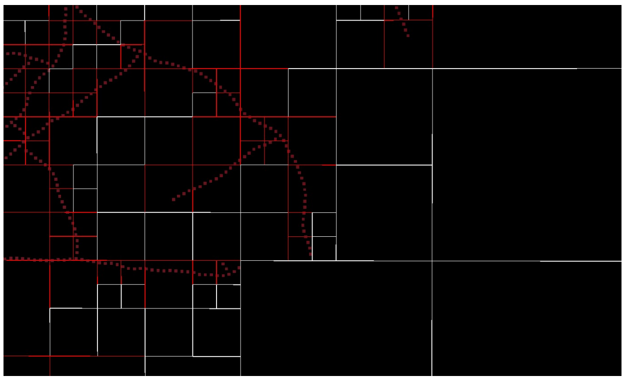
Figure 4: Populated Quad-Tree with a cell limit of eight

The primary function of the quad tree is handled by the `GetNearRoadNodes` method. When an agent queries for nearby nodes within a given radius, the tree recursively traverses its branches. Here, a critical optimization is performed at each node – the `CircleIntersection` method checks if the quad node's boundary overlaps with an agent's circular search area. If there is no overlap, the branch is pruned from the search. Only when the search reaches intersecting leaf nodes does the function query the distance to each individual road node. This two-phase process of broad and precise checking is what allows this system to scale effectively for larger road networks.

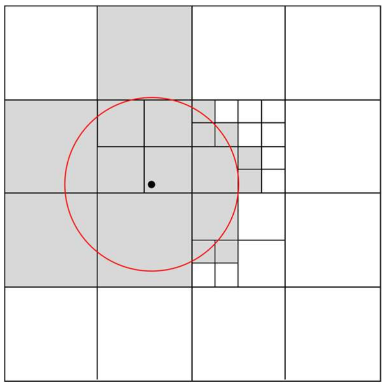
Figure 5: Chunks contacting the circle are marked for further search

3.2.2 Network Graph Representation

While the quad tree provides the spatial index, the decision was made for the road network to be stored in a graph structure. While the quad tree has a pointer to each vertex, they reside within this graph structure and are represented by the `RoadNode` class. A single road node stores not only its position in space but also a branch ID, a copy of the head node responsible for generating this node and is crucial for performance; allowing the intersection-finding algorithm to verify if a nearby node belongs to a different branch without requiring a slow graph traversal.

```
class RoadNode {
        Vector3 position;
        List<HalfEdge> edges;
        int branchID;
}

class HalfEdge {
        RoadNode to;
        RoadNode from;
}
```

Figure 6: Class structure of `RoadNode` and `HalfEdge` classes

Connections between road nodes are defined by the `HalfEdge` class, which represent a one-way connection to a neighbouring node. This decision to not have a list of whole connections comes from the gained benefits it provides for future traversal algorithms. The `CreateConnection` method implements a symmetric relationship by adding a half edge to both nodes involved, to ensure roads are inherently undirected.

3.2.3 Head Node

The generation process is driven by a collection of agents, each an instance of `HeadNode` and is independent of any other agent with its own state of various variables controlling the behaviour of the agent. This agent-based design allows complex global patterns to emerge from simple, local rules. A decision was made to implement `HeadNode` as a data-centric class, separating its state from the logic that resides within the main logic functions. This approach, where the class acts purely as a state container,

```
class HeadNode {
        Vector3 position
        Vector3 direction;
        RoadNode previousNode;
        int nodesSinceSplit;
        bool disabled = false;
        int headID;
}
```

Figure 7: Class structure of `HeadNode` class

simplifies the overall architecture and is applied repeatedly across this project. It encapsulates the necessary collection of variables that are essential for an agent to execute its rule-based behaviour: a position and direction, previously created road node, and branch ID.

3.2.4 Main Logic Controller

The `RoadGenerator` **class** controls the simulation via discrete steps/ticks. During each tick, it iterates through every active head node and executes a series of actions that grow the network. For each active head node, the algorithm applies local rules that govern its immediate behaviour which can be seen in the figure below.

The rules can be split into four actions:

- Node creation and connection: A new road node is created at the agent's current position. This new node is then connected to the stored previous node via the `CreateConnection` method.

- Intersection Detection: To avoid overlaps and ensure a connected graph, the agent must check for proximity to existing roads. Using the `GetNearRoadNodes` method within the quad tree, it queries for nearby nodes within a defined radius (set at 1.5x the standard distance between nodes to account for certain edge cases). A check is performed to ensure this node does not belong to the same branch via `headID` comparison to prevent cases where it may attach to nodes too close to itself on the branch.

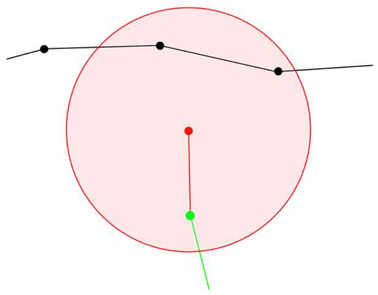

Figure 8: Intersection calculation of a head node's neighbours

- Movement and propagation: If an agent has not been terminated in the previous step, its direction is randomly altered by a random value within the `angleRandomness` parameter, responsible for creating the organic, meandering quality of the roads. The agent's position is then advanced along this new direction by a fixed `nodeDistance`.

- Boundary Constraints: A final check ensures the agent's new position is within the predefined map boundaries. If it moves outside, it is also disabled via the **disabled** boolean flag and prevents the network from growing indefinitely.

The controller also applies global rules that influence the overall structure of the network, which is handled before the local rules are executed for each agent.

The principal global rule is branching, managed by the `SplitHeadNode` method. The likelihood of an agent splitting is governed by the number of nodes since splitting, and a global density map which can be set via an image or generated at run-time using Perlin noise. At each step, the position is used to sample this density map, determining the number of nodes the agent must place before it is allowed to split. In denser areas, this threshold is lower, leading to more frequent branching and subsequently a tighter network.

When an agent meets its splitting condition, a new head node is instantiated and added to a list of all other head nodes. This new agent branches off from the original agent's last placed node `prevNode`**,** its initial direction being the parent's direction rotated by a predefined `splitAngle`. To avoid messy intersections and edge cases immediately upon splitting, a check is performed to ensure the new branch does not spawn too close to an existing road. The original agent's split counter is reset, and both agents are assigned new, unique branch IDs.

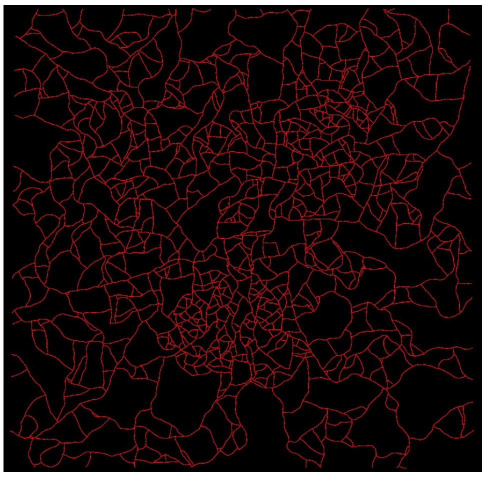

Figure 9: Generation of a road network using agent-based generator. Denser road networks are a result of noise maps

The final step, responsible for cleaning up old head nodes, is also performed. The `PurgeOldHeads` method is triggered when the number of disabled agents reaches a certain threshold. It rebuilds the list of head nodes, discarding any inactive ones. This process is a vital optimisation to ensure the main loop does not iterate over an ever-increasing number of terminated agents and improve the simulation's overall performance as the network grows.

3.3 Basic Chunk Generator

The foundation of this first chunk-based approach is inspired by the methods discussed by Greuter et al. [6]. The core concept involves partitioning the world into a grid, where each chunk is generated independently before being combined to form a larger environment. The following implementation is developed as a foundational base, establishing a pipeline of which specific components of the process can be individually refined or replaced with sophisticated algorithms later without requiring an overhaul.

3.3.1 Deterministic Chunks

For this system to create the illusion of a persistent and infinite world, two principles are key: it must be deterministic and dynamic. Each chunk must generate its contents independently, without any knowledge of its neighbours, and produce an identical result every time. This is essential as chunks are dynamically loaded and may be loaded in a different order.

To implement this, active chunks are managed in a dictionary which is keyed by a tuple of their unique **(x, y)** integer coordinates. These coordinates are then used as the primary input for a seed generation function `GetChunkSeed`, which hashes the input into a pseudo-random numerical seed for each chunk. The seed drives all subsequent procedural calculations in the chunk, guaranteeing that it will be perfectly reconstructed every time it is loaded into the scene.

3.3.2 Node Initialisation

The generation of a chunk begins with the `GenerateRoads` method, which establishes the foundational road network. First, a regular grid of road node positions is calculated based on the chunk's size and the `roadPartitions` parameter, which dictates the number of road subdivisions within a chunk.

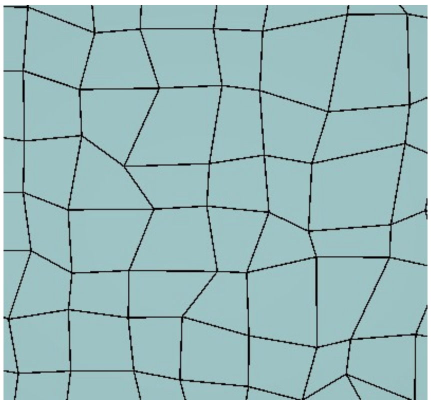

Figure 10: Extreme example of node shifting, with a `NodeJitter` parameter of 0.12

To break the uniformity of this perfect grid, and attempt to introduce a more organic appearance, each node's position is shifted by a small, random amount. A problem arises for nodes along chunk boundaries, which are shared across different chunks and thus the offset must be predetermined to maintain seamlessness at chunk boundaries. To achieve this, the built-in `Mathf.PerlinNoise` method is called with the node's absolute world position as input and guarantees that a node generated on the seam receives the exact same offset, regardless of the current chunk.

3.3.3 Connection Initialisation

Following node placement, the next step is to define the topology of the road network. As the ultimate goal of this algorithm is to generate structures, a simple undirected graph is not enough, a similar approach to the previous algorithm was taken of using half edges. This represents a single directed edge with a `from` and `to` node, and as such the layout must be generated with this in mind. For each node within the interior of the chunk, a full set of half-edges are connected to and from each neighbour. For nodes on the chunk boundary, only the inward-facing half edges are generated, as the corresponding outward-facing edges are implicitly generated by adjacent chunks, ensuring no edges are duplicated unnecessarily.

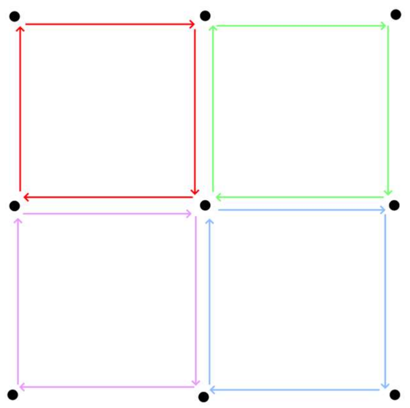
Figure 11: Connection diagram of basic 3 by 3 grid. Outer connections only have a single one-way half edge.

3.4 Generating Meshes

Once a 2D road network has been generated by either of the preceding algorithms, the abstract graph of nodes and edges must be translated into 3D geometry. The process for mesh generation has been designed to be modular; it can be applied to any road network, provided its data structure contains the necessary information, and can be modified with new algorithms to generate different shapes. To begin the process of mesh generation, the network must first be prepared properly.

3.4.1 Next Pointers

To enable the upcoming face-finding algorithm to function, the graph's topology must be modified. This is accomplished by augmenting the `HalfEdge` data structure to include a `next` pointer, designed to store a reference to the subsequent edge in a counterclockwise traversal around the `to` vertex of the half edge. This effectively creates a linked list that traces the perimeter of a face.

The algorithm to assign these points exploits a key geometric property: the `next` edge is always the one with the smallest counterclockwise angular displacement from the current edge. To calculate this, the process considers each half edge as an incoming vector to its destination (`to`) node which is then reversed to serve as a baseline reference.

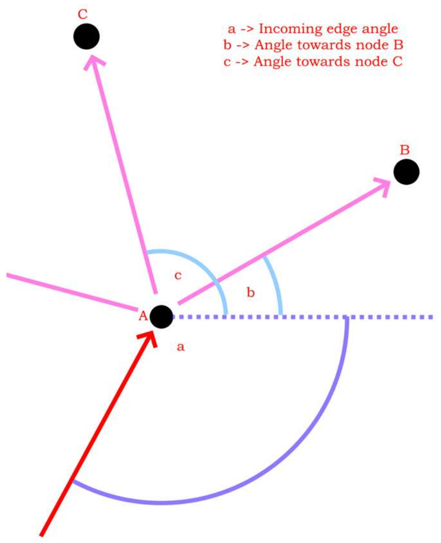

Figure 12: Diagram of angle calculation. Note that angle a is counter-clockwise from the x-axis.

The algorithm then iterates through all other edges originating from the same node, and the relative angle between the reference vector. Each potential outgoing edge is calculated using Unity's `Mathf.Atan2` function, which returns the angle of a vector in radians from the positive X-axis. The core calculation for the angular difference is:

$insert\ equation$

This formula ensures that the result is always a positive angle representing the counterclockwise turn from the reference edge. The algorithm identifies the edge with the smallest positive value and is assigned as the next pointer for the original half edge. By systematically applying this sorting process to every edge in the graph, the network is prepared for efficient face traversal.

3.4.2 Face Traversal

The goal of "discovering" every `face` – the ordered set of edges and nodes forming a closed loop – is now reduced to a simple traversal algorithm due to the previous population of next pointers. As implemented in the `GenerateBlocks` method, this process systematically iterates through every `HalfEdge` in the graph to ensure no face is missed.

A `faceVisited` boolean flag is used on each edge to prevent redundant computation. When an unvisited edge is encountered, it signals the boundary of a new, undiscovered polygon, and the algorithm begins a traversal.

The loop traverses the next pointer chain from the first edge, adding any nodes it encounters into a new `Face` object. This walk continues until the algorithm returns to its starting edge, at which point a complete polygonal face has been traced. The resulting `Face` object is stored in the `Chunk` class, and the main loop continues its search for the next unvisited edge. A notable consequence of this implementation is how it handles the unbounded outer face.

Typically, this traversal method would identify the infinite area surrounding the graph as a valid face. However, because the initial connection step deliberately omits the creation of outward facing half edges at the chunk's boundary, no such cycles can be formed. This ensures that the algorithm only ever identifies the internal polygons that represent buildable city blocks.

3.4.3 Inset Algorithm

Once the polygonal faces representing city blocks have been identified, the shape can be refined to define the building and pavement areas. This is done via a polygon inset algorithm, which creates a smaller, concentric version of the Face polygon within the original boundary. It represents the clear margin between what will become the building's footprint and the surrounding road or pavement.

The `InsetPolygon` method, described by D. R. Finley [8], takes an ordered list vertices defining a polygon and an `insetDistance` as input. Rather than the simple approach of scaling the polygon, which would not produce a uniform border, the algorithm calculates the new position of each vertex individually.

It begins by iterating through every vertex of the polygon, treating each one as a corner to be inset, via the following logic: Two vectors, `v1` and `v2`, are defined, representing the edges between the two neighbouring vertices. A perpendicular normal vector is also defined, using the formula `(x,y)->(y,-x)` when the polygon is wound in a counterclockwise fashion. The two line segments are shifted inward by the given `insetDistance` and a new corner vertex is defined precisely where the two new offset line segments intersect. All the new corners are added to a new list of polygons, labeled `insetPoints` and can be used as a base to effectively generate any shape required.

3.4.4 Final Geometry Generation

The final stage transforms the abstract **Face** and **Block** data structures into visible 3D geometry. The `GenerateStructures` method orchestrates this by processing each identifies block and calling distinct routines to generate the meshes for the surrounding road and buildings within.

3.4.4.1 Road and Pavement Meshing

The `GenerateRoads` method is responsible for creating the ground-level infrastructure and takes two polygons as input: the vertices of the outer boundary of the block, and the vertices of the inset polygon. This two-loop structure provides a robust foundation for generating the road and pavement mesh that occupies the margin between them. While this structure could support advanced features like placing curbs and street furniture, the current implementation focuses on generating a basic ground plane to demonstrate the core functionality.

To construct the mesh, the algorithm iterates along the corresponding vertices of the inner and outer loops. For each segment, a "quad" is formed by the two vertices of the outer edge and the two vertices of the inner edge. To render this in Unity's 3D engine, the quad must be triangulated following the above diagram to ensure correct winding order. For any given segment, the vertices are ordered as (`outer_p1`, `inner_p1`, `inner_p2`) for the first triangle, and (`outer_p1`, `inner_p2`, `outer_p2`) for the second. Repeating this process for every segment creates a continuous, watertight mesh strip.

3.4.4.2 Building Generation as Extruded Prisms

Buildings may be generated in any shape needed, however, to demonstrate core functionality, a simple 3D prism is extruded out of the footprint using the `GeneratePrismMesh` method. It constructs the mesh from two distinct parts: the vertical side walls and the top/bottom caps.

- Side walls: the vertical walls are constructed as a series of quads connecting the corresponding vertices of the top and bottom caps. To prevent incorrect normal interpolation, the vertices for these quads must be unique for each.
- Caps: the bottom cap is a direct triangulation of the 2D inset polygon. The top cap is similar but is duplicated vertically by the desired building height, and the winding order is reversed to ensure the top surface faces outwards.

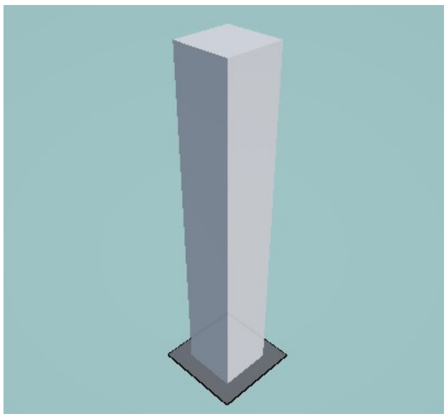

Figure 13: Mesh output for a square block

3.4.4.3 Recursive Subdivision

A more complex approach can be taken to generate separate buildings within these blocks. Rather than extruding the entire inset polygon as a single structure, a recursive subdivision algorithm can be used to create more varied and realistic urban lots. The recursive subdivision algorithm chosen is simple in that it repeats one easy operation multiple times:

```
SplitBlock():
  List<List<Vector2>> polygons
  polygons.Add(inset_points)
  for i = 0 to block_split_amount:
    List<List<Vector2>> new_polygons
    foreach polyogn in polygons:
      Vector2 normal =
GenerateNormal(polygon)
      Vector2 center =
GetAverageCenter(polygon)

      List<Vector2> polygon1 =
CutPolygon(polygon, center, normal)
      List<Vector2> polygon2 =
CutPolygon(polygon, center, -normal)

      new_polygons.Add(polygon1, polygon2)
    polygons = new_polygons
```

Figure 14: Pseudocode for block splitting algorithm

- Initial footprint: the process begins with the single inset polygon that defines the area of the block, which is added to a queue of polygons that must be cut.
- Splitting, the one easy operation: The initial polygon is then subdivided, every resulting polygon added to a new list of polygons to cut. This repeats until `blockSplitAmount` is reached. For example, one initial block splits into two plots, then four plots, then eight. The details of this are explained in Section 3.4.5
- Extrusion: The algorithm iterates through the final list of small plot polygons. For each one, a random height is calculated and the previous `GeneratePrismMesh` method is called to generate a building on this smaller plot of land.

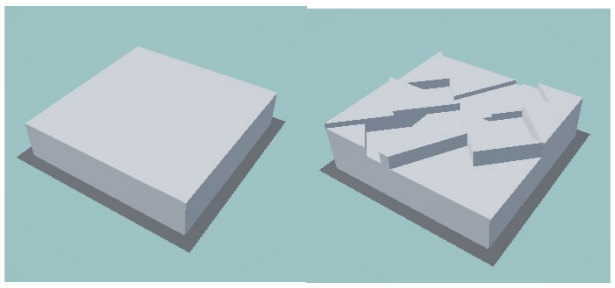

Figure 15: Before and after of block splitting algorithm

This subdivision technique results in a block being populated by multiple distinct structures, creating a more detailed and visually appealing cityscape. The final collection of road and building meshes are attached to **GameObjects** and parented to a single container object.

3.4.5 Polygon Cutting/Clipping

The core of the recursive subdivision process is the 'splitting' operation, which is achieved using a polygon clipping algorithm. This technique takes a single polygon and divides it into two smaller polygons along a defined line. The implementation is a variation of the Sutherland-Hodgman algorithm, which is highly efficient for clipping convex and concave polygons against a clipping plane.

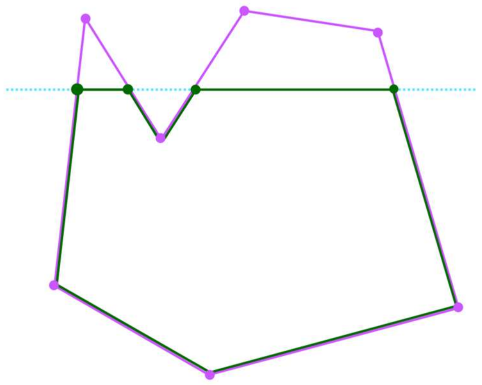

Figure 16: Diagram demonstrating the polygon splitting algorithm.

The `CutPolygon` method takes the polygon to be split and defines an infinite cutting line using a single point and normal vector. This line divides the 2D space into an inside space (the side the normal points to) and an outside space. The function's goal is to construct a new polygon containing only the portion that lies on the inside.

The algorithm iterates through each edge of the input polygon, defined by a `start` and `end` vertex, and checks if its vertices are inside the clipping plane. One of four rules is applied based on the result:

- Both vertices are inside: The edge is entirely within, the `end` vertex is added to the new polygon
- The `start` is inside, and `end` is outside: The edge crosses the clipping plane on the way out. Only the intersection point of the edge and the clipping plane is added to the list.
- Both are outside: The edge is completely discarded as it is outside the region.
- The `start` is outside, and `end` is inside. The edge crosses the clipping plane on the way in. The calculated intersection point is added to the new list first, followed by the `end` vertex.

After applying these rules to every edge in sequence, the result is a new, successfully split polygon. In situations where both sides of the polygon are needed, such as the previous chapter, this method is used twice when splitting a polygon. Once with the given normal to get the first half, and the second time with an inverted normal to get the other half.

3.5 Organic Chunk Boundaries

While the grid-based chunk system provides a robust and scalable solution for the chunk-based generation problem, its structure results in uniform and repetitive cityscapes. This can represent certain real-world cities well; however, they lack the organic complexity found in other urban environments. To address these limitations and introduce a more naturalistic design, a more advanced algorithm based on Voronoi diagrams was developed.

This approach replaces the rigid grid with procedurally generated irregular polygonal cells. A Voronoi diagram partitions space into regions based on proximity to a set of random points. By using these regions as the basis for city blocks, it is possible to generate road networks that are non-uniform, and with varied road angles and block sizes.

3.5.1 Voronoi Cell Chunks

This method builds off the structure of the grid-based chunk system, where the world is managed by a series of **Chunk** objects in a grid. However, the visual and logical boundaries of the chunks are redefined. Each chunk is assigned a single, unique site point that serves as the nucleus for a Voronoi cell and placed randomly within the chunk. To ensure the chunk is generated deterministically, the chunk's integer grid coordinates are used as a seed to generate the offset responsible for the position. Furthermore, the point is not placed completely random within the chunk's area. Instead, it is scaled down based on the `edgeGap` variable, which is responsible for pushing points away from the edge and thus preventing catastrophic edge cases in the algorithm, as well as preventing clusters of points that create undesirable sliver-like regions.

To calculate the polygonal boundary of the Voronoi cell for a given chunk, the algorithm must consider the site points of all its immediate neighbours. Due to the edge gap previously introduced, certain issues are prevented, such as the situation described in the figure above, and therefore only require the site points of a 3x3 grid surrounding the chunk.

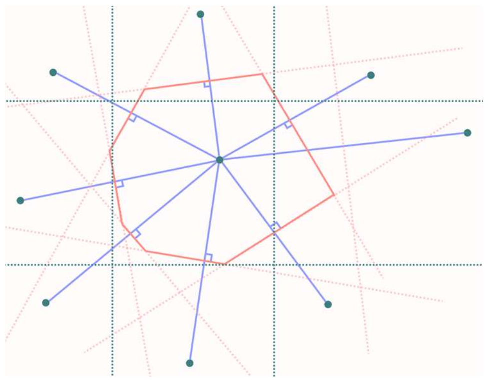
Figure 17: Polygon cutting algorithm used with every normal of each of the 8 neighbouring points.

With these set of nine points, a simple algorithm employing previous polygon cutting algorithm is used to generate the polygon representing the Voronoi cell. The fundamental principle of a Voronoi diagram is that every point within a cell is closer to that cell's site point than any other site point, and this is exploited to construct the cell's boundary.

The algorithm begins with a large initial bounding polygon, typically twice the area of the chunk itself. It then iterates through each of the eight neighbouring site points, defining a clipping half-plane for each point This half-plane represents all the space that is closer to the central site than to the current neighbouring site and is calculated using the perpendicular bisector of the line segment connecting the two site points.

The current polygon (which starts as the initial boundary and is slowly chipped away) is clipped against this half-plane using the `CutPolygon` method, keeping only the portion that lies inside it which becomes the input for the next iteration with the next neighbour. After the process is complete with all eight points, the final remaining polygon is the correct Voronoi cell for the central site.

A problem with this process can occur when points arrange themselves in such a way that the central cell becomes unbounded. This is the reason for the introduction of the `edgeGap` variable, as it forces the site point away from the edge.

3.5.2 Block Generation within Chunks

With the primary boundary of the chunk established, the next step is to subdivide this large polygonal area into smaller, individual city blocks. This is achieved by applying the Voronoi generation principle a second time, but on a more granular level and constrained within the main cell's boundary.

First, a new set of site points is procedurally placed within the interior of the parent Voronoi cell. This process is handled by the `GeneratePoints` method, which uses an approach analogous to Poisson disk sampling and ensures a minimum distance is maintained between any two points. This prevents the clustering that can lead to undesirably small or narrow blocks. The approach taken was a simple brute force method, as the number of points within a chunk would not exceed a number that would impact performance; the option to use the previously made quad-tree was dismissed as performance was lower at smaller point counts.

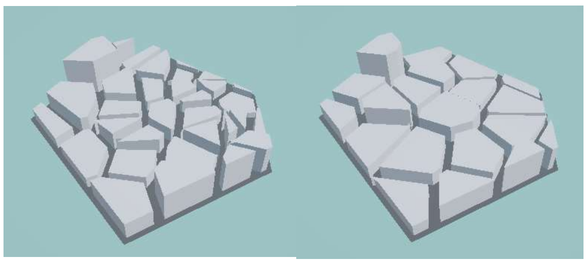
Figure 18: With and without Poisson disc point distribution. Blocks vary more in size without distribution.

To efficiently calculate the Voronoi cells and prevent the use of the polygon cutting method against every combination of points, a Delaunay triangulation is constructed. A key property of these triangulations is that their dual graph is the Voronoi diagram. This means that for any given point in the triangulation, its connected neighbours are also its only neighbours in the Voronoi diagram. The triangulation was implemented using a third-party library, called `delaulator-sharp` [9], and is used in the next step of cutting the polygon boundary of each cell.

The `GenerateBlocks` method orchestrates this process. It iterates through each of the internal site points, and using its neighbours from the Delaunay triangulation, calculates its corresponding Voronoi cell using the same approach as the main cell. A small difference is performed here. To ensure that a cell does not exceed the chunks of the parent chunk's cell, the initial boundary in the polygon cutting method is set to the boundary of this main cell. All of this is stored within a **Block** object, which stores the boundary polygon of a given block.

The mesh generation aspect is recycled from the previous grid-based chunk system and applied to each Voronoi block. A small improvement here could be to replace the splitting block algorithm with a third layer of Voronoi cells to create a possibly even more organic look to the city.

3.6 Chunk Loading and Unloading

To support the infinite world idea of this project without exhausting too many system resources, the city is generated and deleted dynamically around a player. This process is managed continuously within the `Update` loop.

The system first identifies the player's current position and determines which chunk coordinates they occupy. It then iterates through a square region of chunks centered on the player, with the size of this region defined by the `renderDistance` variable. For each chunk coordinate, it calculates the distance from the center of that chunk to the player. If this distance is within the spherical render radius, the system attempts to create the chunk via the `CreateChunk` method.

Conversely, to manage memory and maintain performance, the system must also unload chunks that are far away from the player. It iterates through the dictionary of all currently active chunks and adds that chunk to a list of chunks to delete if they are outside the `renderDistance` radius. Game objects associated with each marked chunk are kept as a reference within the `Chunk` class and are also deleted before the chunk class itself is removed.

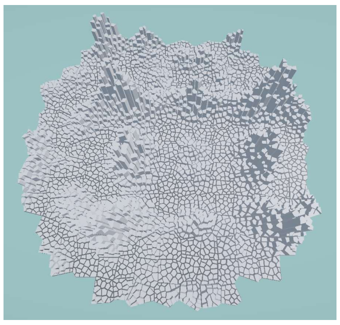

Figure 19: Circular shape of loaded Voronoi chunks.

3.7 Building Height Variation

An improvement on the random height variation can be made using Perlin noise. This can create a more realistic and structured city layout with distinct districts, such as a downtown core and suburban areas with low-rise buildings.

When generating the height of a building, its world-space coordinates (specifically the coordinates of the first vertex of the `insetPolygon` of the block) are passed into Unity's `Mathf.Perlin` function. The result is in the range 0.0 to 1.0 and is then processed using the following formula to create more dramatic and controllable height variations.

$$height = (noise * powerScale)^p * amplitude + offset$$

As shown, the resulting height is a product of several variables. First, the result of the Perlin function is multiplied by a power scale, which increases the range from 0.0 to `powerScale`. This is then raised to a power, which has the effect of pushing values below 1.0 towards 0.0, and any above towards infinity. This creates larger areas of flat buildings, and the occasional high-rise district breaking the horizon where the noise value was closer to 1.0.

EVALUATION

The purpose of this chapter is to critically evaluate the three procedural generation algorithms developed in this project: the conventional agent-based road generator, the basic grid-based chunk generator, and the organic Voronoi-based generator. The goal is to conduct a comparative analysis to identify the distinct strengths, weaknesses and overall suitability of each approach for generating performant and believable urban environments, and assess whether real-time, chunk-based procedural generation is feasible. This will be achieved by assessing the algorithms against a defined set of quantitative performance metrics and qualitative output criteria.

4.1 Evaluation Criteria

To provide a comprehensive analysis, the following criteria were established, encompassing both objective performance and subjective quality of output.

Generation time: This measures the time required to produce a usable road network. For the global agent-based method, tests are conducted on various city sizes to measure how total generation time scales with network growth. For chunk-based systems, performance is first measured at the individual chunk-level with different generation parameters.

Memory usage: This metric quantifies RAM usage of each algorithm. The total memory usage of the Unity process will be measured, as only the comparison is important and not the total usage. A baseline will also be measured to compare against, gathered without any algorithm present.

All tests were completed on a Windows desktop PC with the following specs: Ryzen 7 2700X, 32 GB DDR4 RAM, GeForce GTX 1660 Ti, Windows 10 Home. Unity version 6000.0.48f1.

4.2 Algorithm Performance Testing

Quantitative testing reveals significant performance differences between the three implemented algorithms, highlighting a clear trade-off between computation cost and visual complexity.

Tests relating to chunk performance were tested on 10,000 repetitions.

4.2.1 Agent-Based Algorithm Results

Map Width	*Map Generation Time*	*Memory Consumption*
0 (baseline)	0s	2542

256	0.057s	2551
512	0.218s	2561
1024	0.708s	2604
2048	4.162s	2952

The table presents the performance metrics for the agent-based road generation algorithm, measuring the generation time and memory consumption against increasing map sizes. The data demonstrates the performance characteristics and scaling limitations of this approach. The most critical finding is the non-linear scaling of generation time. At smaller map sizes, the generation is very fast, however as the width doubles, the time required to generate the network increases exponentially. A four-fold increase in map area from 1024 to 2048 causes the generation time to increase nearly six times.

This exponential growth is a classic characteristic of algorithms that require global awareness of the map. As more agents are active, the number of potential interactions grows rapidly.

A similar story appears when looking at memory consumption. While the usage is small at first, the differences between steps of map width grows exponentially. When map area grows four-fold between 1024 and 2048, memory consumption grows by more than six times once again.

4.2.2 Grid-Based Road Partition Results

Road Partitions	*With Mesh*	*Without Mesh*
0 by 0	1.19s	0.237s
2 by 2	8.705s	0.819s
5 by 5	48.117s	18.269s
1 by 5	10.885s	1.101s

This table evaluates the performance of the grid-based chunk generator, specifically focusing on how the internal complexity of a chunk affects generation time. The results are split into two key metrics: the time taken to generate the full 3D geometry, and the time to generate only the underlying 2D road graph.

The generation time increases dramatically as the number of road partitions grows. Increasing partitions from 2 by 2 to 5 by 5 resulted in a performance decrease of five times when generating the full mesh. The number of blocks with these partitions grows from 4 to 25, a six-fold increase.

Mesh generation seems to be a major bottle neck. Across all tests, the process of creating 3D geometry is the most expensive operation by far. I believe this is an issue with the process of creating meshes, as each mesh object is assigned a separate Game Object within the Unity scene.

This setback has implications for the algorithm's scalability in real-time scenarios, as higher grid partition sizes dramatically increase generation time. This can however be offset by generating more chunks at smaller partition sizes.

4.2.3 Grid-Based Block Splitting Results

Block Split Amount	*Time to Generate*
0	8.695s
2	20.685s
3	34.251s
6	252.141s

This table measures the performance impact of the recursive block splitting algorithm. This algorithm increases visual detail of individual blocks by the given block split amount.

The results show that the algorithm is extremely computationally expensive to operate with. The relationship between the number of splits and the generation time is exponential. This severe degradation occurs because each split doubles the number of polygons that need to be processed into meshes and each new sub-plot must undergo the entire mesh generation pipeline.

For practical applications, block split amount would need to be capped at a low value to maintain acceptable performance.

4.2.4 Grid-Based Noise Results

Generated With Noise?	*Time to Generate*
Yes	34.392s
No	34.272s

This table assesses the performance overhead of incorporating Perlin noise into the generation of meshes. The results show that the performance of adding Perlin noise to the algorithm is practically negligible.

4.2.5 Voronoi-Chunk Point Count Results

Point Count	*With Mesh*	*Without Mesh*
3	3.686s	0.473s
8	9.763s	1.285s
16	19.151s	4.389s
32	29.258s	9.206s

This table examines the performance of the Voronoi-based generator when changing the point count parameter, which determines the number of Voronoi cells generated within a single chunk. This controls the density and number of meshes to generate per chunk.

Increasing detail of Voronoi chunks seem to have a major impact on performance. A key finding is that the underlying logic for generating the diagram is computationally intensive on its own. The time for this step grows rapidly due to the complex algorithms required, such as Delaunay triangulation and repeated polygon clipping for each point to define its cell.

Despite the heavy cost of graph logic, creating 3D geometry remains the single most time-consuming part of the process.

4.2.6 Voronoi-Chunk Block Split Results

Block Split Amount	*Time to Generate*
0	19.023s
1	24.836s
2	39.170s
3	66.853s
6	457.472s

The final individual algorithm test measures the performance of applying the recursive block splitting algorithm on the Voronoi generator. Once again, the goal is to assess the cost of adding fine-grained detail to the more organic city layouts.

The results confirm the trend seen with the grid-based system but in a more extreme fashion, although the number of blocks per chunk is about double the previous system. The performance cost of splitting Voronoi blocks is expensive and scales exponentially due to similar reasons to the grid-based method.

4.2.7 Grid-Based Road Partition Memory Results

Road Partitions	*Memory (MB)*
0 by 0	4338
2 by 0	4351
5 by 5	4400

This table examines the impact of internal chunk complexity on memory consumption for the grid-based system, measuring road partition counts.

Increasing detail within chunks only seems to have a modest impact on memory usage. This implies that the bottleneck for this implementation seems to be the time required for generation, rather than memory consumption.

4.2.8 Grid-Based Block Split Memory Results

Block Split Amount	*Memory (MB)*
0	4343
2	4344
3	4351
6	5240

This table demonstrates the memory consumption associated with the recursive block splitting feature.

The memory usage scales non-linearly with the block split amount, remaining low for a few splits before increasing dramatically towards 6 splits. This occurs due to the doubling nature of the splitting method. High levels of block splitting are not only computationally expensive, but also memory intensive.

4.2.9 Grid-Based Render Distance Memory Results

Render Distance	*Memory (MB)*
8	4349
16	4545
32	7899
64	13269

The final test for the grid-based generation algorithm's memory consumption is of how memory consumption scales with render distance. This setting determines the radius of chunks to load around the player and keep active in memory.

Memory usage increases quadratically with the render distance. This is expected behaviour, as the render distance acts as a radius, while the number of chunks is proportional to the area of the circle.

4.2.10 Voronoi-Based Point Count Memory Results

Point Count	*Memory (MB)*
3	4569
8	4602
16	4675
32	4717

These results measure the memory cost of increasing block density within Voronoi chunks.

Adding more detail seems to have a surprisingly minor impact on memory. Increasing the internal point count of a cell increases linearly, as each point is only responsible for one cell within the chunk.

4.2.11 Voronoi-Based Block Split Memory Results

Block Splits	*Memory (MB)*
0	4675
2	4697
3	4792
6	6832

This test measures the memory cost of recursively splitting the organic blocks produced by the Voronoi generator.

The result is similar to other tests; they show a sharp and non-linear scaling in memory consumption. While a few splits are modest, the impact of deeper recursion becomes extremely costly.

4.2.12 Voronoi-Based Render Distance Memory Results

Render Distance	*Memory (MB)*
8	4790
16	5648
32	9849
64	19145

This test measures how memory usage for the Voronoi system scales with render distance.

As expected, the memory footprint grows quadratically with the render distance, since the number of loaded chunks is proportional to the radius of the circle around the player.

4.3 Qualitative Analysis and Visual Output

While the quantitative data reveals the performance trade-offs of each algorithm, it doesn't paint the whole picture. The algorithms must also be assessed in their effectiveness in producing believable and aesthetically pleasing urban environments. This qualitative analysis examines the visual characteristics of the output from each of the three generators, comparing their strengths and weaknesses in achieving visual variety, structural realism, and overall aesthetic appeal.

4.3.1 Agent-Based Algorithm

The conventional agent-based generator excels at producing organic and natural road networks. By simulating the growth of roads with rules for branching and intersections, this method creates layouts that can feel historically plausible, resembling cities that have grown over time rather than being planned.

The introduction of randomness in agent direction and the influence of a global density map result in flowing, meandering streets that effectively mimic the transition from dense urban cores to sparser suburbs, while avoiding repetitive patterns that can plague simpler systems and keep the network interconnected.

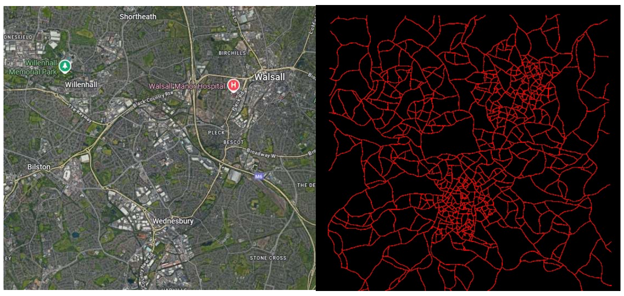

Figure 20: Comparison of Agent-based algorithm result to towns in UK.

The primary visual drawback of this method, however, is its inherent unpredictability. It can lead to layouts with many dead ends or awkwardly shaped city blocks that don't allow for easy building placement.

This algorithm serves as a good benchmark for global road network generation; it creates intricate patterns that are interesting to look at. It has an issue with generating sliver-like block sections however, which can be an issue with subsequent algorithms for generating buildings.

4.3.2 Grid-Based Chunk Algorithm

This algorithm represents a direct and functional approach to chunk-based generation. Its output is characterised by a highly structured and predictable layout, similar to modern planned cities like Manhattan.

The algorithm manages to produce clean, well defined city blocks. The underlying grid is understandable and easily navigable, and the use of Perlin noise to shift the intersection positions breaks up the perfect uniformity. When combined with the recursive block subdivision, it can create dense, detailed areas.

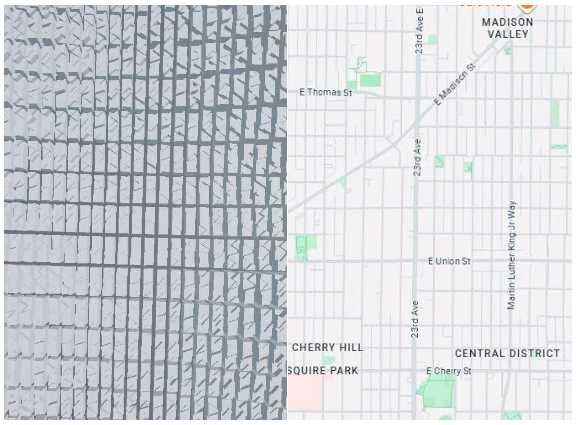

Figure 21: Grid-based Generator and suburb of Seattle, USA side by side.

Arguably, the algorithm's worst downside is its monotony. Despite the offsetting, the output is immediately recognizable as a grid which can be believable for cities known for their grid plan, but it fails to capture the character of more organically grown urban centers. The skyline may vary, but the street-level experience remains uniform wherever you are.

The algorithm's strength comes from its order and structure. It allows subsequent systems to have a predictable layout to handle by having consistent block sizes and intersection angles.

Ultimately, while it is highly performant, it produces cityscapes that are functional rather than compelling.

4.3.3 Voronoi-Based Chunk Generator

This generator was designed to modify the previous algorithm and address the aesthetic shortcomings of the grid system while remaining performant. It represents a significant improvement and a successful bridge between the gap of the other two methods.

The algorithm's biggest strength is its believability. By using Voronoi cells for both the chunk boundaries and internal blocks, it produces cities with highly organic and varied appearance. The two-tiered Voronoi approach creates structures that look planned yet natural. The irregular skylines created by applying Perlin noise to building height result in more dramatic and visually interesting shapes when set against the non-uniform block pattern.

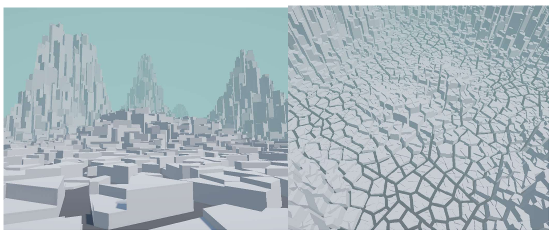
Figure 22: Screenshots of city made with Voronoi-Based Generator

Due to the nature of Voronoi diagrams and their implementation, occasional sliver-like blocks appear with very acute angles, which can be challenging to fill with buildings effectively.

The algorithm produces cities that are both functional and compelling, splitting up the monotony of a simple grid.

CONCLUSION

This project sets out to investigate and evaluate different chunk-based algorithms for procedural city generation. The primary aim was to compare a novel, organic approach against more conventional methods to determine the feasibility of producing compelling and performant cityscapes suitable for real-time applications. Through the implementation and analysis of three distinct algorithms, the research has yielded significant insights into the trade-offs between computational performance and aesthetic quality.

Each algorithm produces plausible urban environments, though their suitability is ultimately dependent on the specific use case and desired aesthetic. The conventional agent-based system excelled at creating visually complex road networks, serving as an effective benchmark.

In contract, both chunk-based systems proved viable. The grid-based generator was the more performant and scalable method, being able to generate chunks quicker, at the cost of producing a visually monotonous cityscape. The Voronoi-based generator successfully addressed this limitation, producing highly organic and varied urban layouts that avoid the repetition of a simple grid. While the quantitative analysis revealed that this generator is more computationally intensive, the difference is not by much and can be accommodated by the improved visual appeal it has to offer. This makes both chunk-based approaches viable, with the choice depending on the desired balance between visual complexity and performance targets.

Ultimately, the most suitable algorithm is dictated by the intended application, particularly within video games. The agent-based approach is ideal for pre-generating a single detailed map for a game with a fixed world. The two chunk-based systems are suited for more high-performance games, or ones that necessitate an infinite world for gameplay. This project demonstrates that all three approaches are suitable for real-world applications, and each have their own benefits and drawbacks for any specific use case.

This research was not without its limitations. The project's focus was on the high-level algorithms and road layout, leaving the rendering pipeline largely unoptimised. The buildings themselves were simple extruded prisms, lacking the detailed architectural features that bring a city to life. Furthermore, the generation was purely geometric, with no understanding of urban planning principles such as zoning.

FURTHER WORK

The limitations within this project pave the way for several avenues for future work, which could build upon this project's foundation to create even more dynamic and believable worlds.

A critical next step would be to address the mesh generation bottleneck. Instead of creating a unique `GameObject` for every mesh, techniques like GPU instancing and compute shaders could be employed to drastically reduce draw calls and improve performance.

Hybrid algorithms could be explored that combine the strengths of different algorithms. For instance, a high-level Voronoi graph could define a city's main arterial roads and districts, while a faster agent-based on grid system could fill in the local streets within each Voronoi cell. This could create a more realistic road hierarchy and more visual variety within the network.

The system could also be enhanced by moving beyond pure geometry. Natural features such as hills and rivers can be introduced, which would act as major constraints that the road network must plan around. Data maps could be generated to define land use, allowing the generator to place large, open areas like parks or farms in low-density zones, and plan building shapes along industrial-residential borders. Creating joined-up blocks like terraced houses or commercial strips may also prove beneficial to break up the repetitive nature of the algorithms.

To improve visual fidelity, a grammar-based system, such as that proposed by Wonka et al. [4], could be integrated. This would use the building footprint as a base to generate detailed facades, windows and roofs, creating a more believable and diverse cityscape.

The current system currently loads and unloads entire chunks. An LOD system could be implemented to prevent wasted resources and improve render distance. For distant chunks, building meshes could be simplified or combined into a single low-poly mesh. A simplified version of a building could also be rendered first, with full detail streamed in as the player gets closer.

One important feature that was missed from this implementation is the lack of asynchronous generation. To prevent gameplay from stuttering while new chunks are loaded, games such as *Minecraft* offload the generation of chunks to a separate CPU thread. Once the geometry and data are ready, the results can be seamlessly loaded into the main scene. This ensures a smooth user experience, even when moving quickly through the environment.

As well as varying building height with Perlin noise, it can also be used to vary the floor height of the city too. The current system uses a flat place, which can be fixed by adapting all buildings and roads to a generated heightmap.

REFERENCES

[1] Yoav I. H. Parish and Pascal Müller. 2001. **Procedural modeling of cities.** In Proceedings of the 28th annual conference on Computer graphics and interactive techniques (SIGGRAPH '01). Association for Computing Machinery, New York, NY, USA, 301–308. https://doi.org/10.1145/383259.383292

[2] Ken Perlin. 1985**. An image synthesizer**. In Proceedings of the 12th annual conference on Computer graphics and interactive techniques (SIGGRAPH '85). Association for Computing Machinery, New York, NY, USA, 287–296. https://doi.org/10.1145/325334.325247

[3] Kelly, George & McCabe, Hugh. (2007). **Citygen: An Interactive System for Procedural City Generation.**

[4] Peter Wonka, Michael Wimmer, François Sillion, and William Ribarsky. 2003. **Instant architecture**. ACM Trans. Graph. 22, 3 (July 2003), 669–677. https://doi.org/10.1145/882262.882324

[5] Jing Sun, Xiaobo Yu, George Baciu, and Mark Green. 2002. **Template-based generation of road networks for virtual city modeling**. In Proceedings of the ACM symposium on Virtual reality software and technology (VRST '02). Association for Computing Machinery, New York, NY, USA, 33–40. https://doi.org/10.1145/585740.585747

[6] Stefan Greuter, Jeremy Parker, Nigel Stewart, and Geoff Leach. 2003**. Real-time procedural generation of `pseudo infinite' cities**. In Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia (GRAPHITE '03). Association for Computing Machinery, New York, NY, USA, 87–ff. https://doi.org/10.1145/604471.604490

[7] Guoning Chen, Gregory Esch, Peter Wonka, Pascal Müller, and Eugene Zhang. 2008. **Interactive procedural street modeling**. ACM Trans. Graph. 27, 3 (August 2008), 1–10. https://doi.org/10.1145/1360612.1360702

[8] Finley, D. R. 2011. **Inset A Polygon By A Fixed, Perpendicular Distance**. Alienryderflex.com, Retrieved August 12, 2025 from https://alienryderflex.com/polygon_inset/

[9] nol1fe, 2023. **delaunator-sharp**. GitHub Repository. Retrieved August 12, 2025, from https://github.com/nol1fe/delaunator-sharp

[10] "**Perlin Noise**" Wikipedia, Wikimedia Foundation, Retrieved August 12, 2025, from https://en.wikipedia.org/wiki/Perlin_noise

[11] "**Voronoi diagram**" Wikipedia, Wikimedia Foundation, Retrieved August 12, 2025, from https://en.wikipedia.org/wiki/Voronoi_diagram