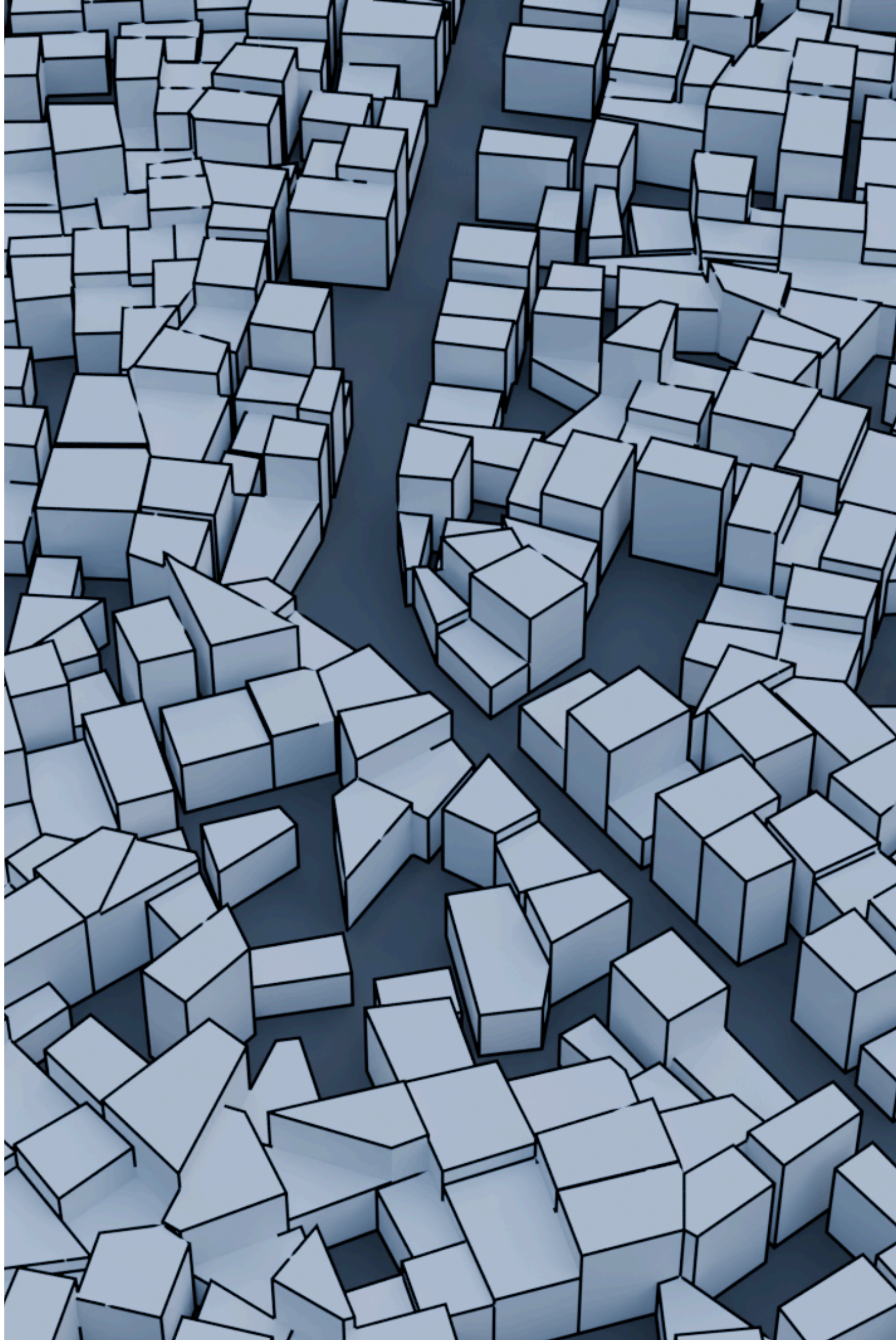




Procedural Modelling of Informal Settlements Using Voronoi-Based Spatial Subdivision

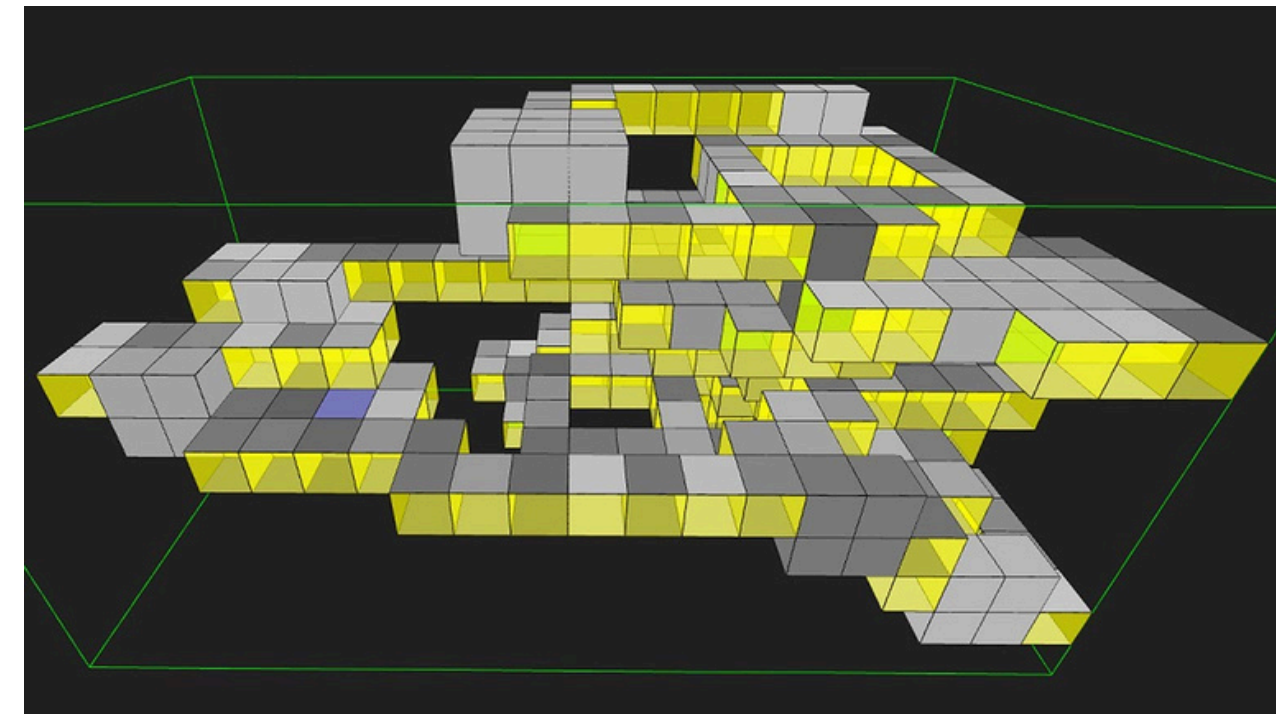
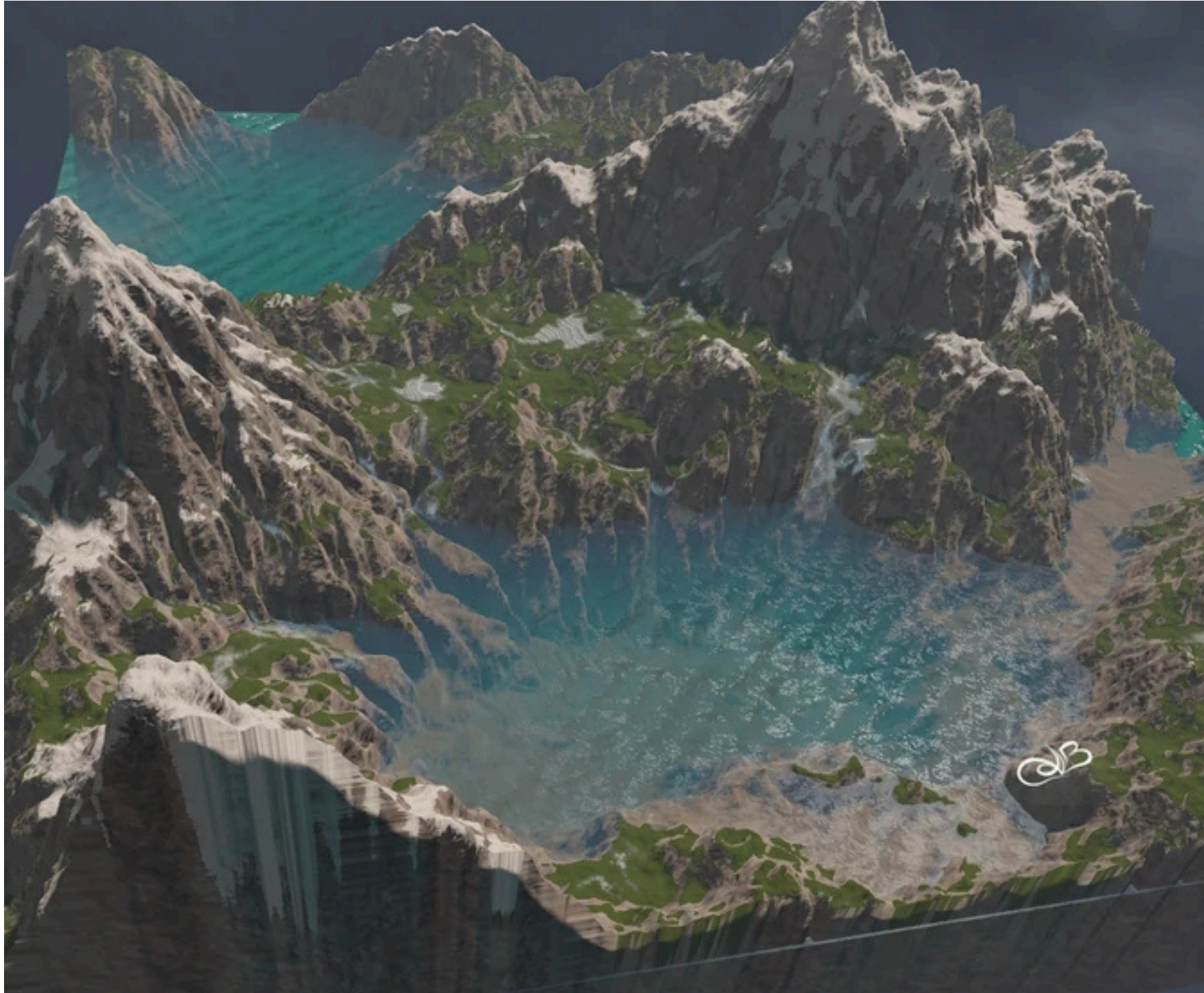
Marcel Putra
TP066419

BSc (Hons)
Computer Science



Our World is Built by Algorithms

What is Procedural Generation?



Algorithms for procedural generation and display of trees

H. Nuić*, Ž. Mihajlović*

*University of Zagreb/ Faculty of Electrical Engineering and Computing, Zagreb, Croatia
hrvoje.nuic@fer.hr, zeljka.mihajlovic@fer.hr

Abstract – Main goal of this paper is to explore in which situations should some procedural algorithm be used to generate a tree model. Each algorithm is modified so as to be able to generate a tree model whose shape resembles the required model. Space colonization algorithm, algorithm using particle flows and algorithm simulating a Lindenmayer system are compared depending on a time needed to generate a tree with similar complexity. The voxelization procedure of the model and its application in this context is explained. Method for generating a tree mesh on a graphics card using Bézier's curves is presented.

Keywords – tree; procedural generation; space colonization; L-system; particle flow; voxelization; Bézier curve

I. INTRODUCTION

Modelling and animating trees for movies, computer games and simulations are labour intensive work. Trees are most commonly represented as static objects with minimal interaction with its surroundings. Main reason is the sheer complexity of organic trees. Fully grown *common oak* can exceed the height of 40m and have more than 100,000 branches.

There are many commercially available tools which can help with modelling the tree, but the process is still time-consuming. Procedural algorithms can mitigate the time needed to create a desired tree model.

In this paper we have compared 3 algorithms. Every algorithm as an input gets voxelized model and as an output produces same abstract tree data structure. This data structure is further passed to graphics card which creates a mesh for every branch.

Procedural algorithms to some extent try to simulate the real growth of a tree, but as there are many unknown variables, it is hard to capture the real world. We have chosen to compare space colonisation algorithm, algorithm using particle flow and algorithm simulating a Lindenmayer system. Each one takes a different approach to the same problem. Space colonisation simulates a competition for space. Particle flow algorithm exploits similarities between branching structures in trees and trajectories of particle simulations. L-system focuses mainly on rules at which branching happens.

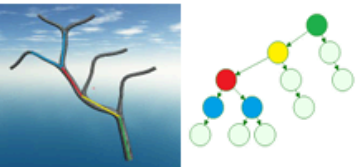


Figure 1. Rendered tree and its abstract tree structure

II. METHOD

In this section we will present the process of constructing a procedurally grown tree. Every algorithm is parametrized with a desired tree-crown size and voxelized model which we want to imitate.

Most common operation while generating a tree model with a desired shape is a test if some point is inside the model. While polygonal models for that operation have linear time complexity proportional to the size of a model, voxelized models have a constant time complexity. Main disadvantage of using voxelized models is a loss of precision, but with using 256³ voxels and trees having irregular form there was no noticeable difference in a generated tree.

Models are voxelized with a variation of an algorithm using ray casting presented in [1] and is described in the next section. Since the process of voxelization didn't take longer than a few seconds at the start of a program, there was no need for using a faster voxelization algorithm, like one presented in [2].

Every tree generator produces a tree data structure as demonstrated on a figure (1). From the perspective of a red branch, blue branch is its child, yellow branch is its parent, and green branch is its grandparent. Each node in a tree data structure stores its length and relative rotation to its parent branch. Main advantage with using abstract representation is that it reduces data that needs to be transferred to graphics card.

Abstract tree structure is converted on a graphics card to a mesh which is further rendered in a OpenGL graphics pipeline. This process is described in section VIII.

Tree Generation with Different Algorithms

Terrain Generation Using Procedural Models Based on Hydrology

Jean-David G  nevaux¹   ric Galin^{1*}   ric Gu  rin¹ Adrien Peytavie¹ Bedřich Ben  s²

¹ Universit   de Lyon, LIRIS, CNRS, UMR5205, France ² Purdue University, USA

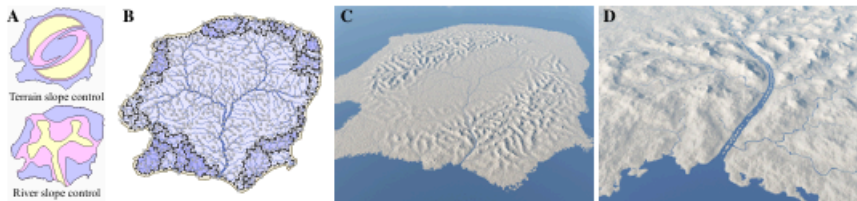


Figure 1: A) The shape of a terrain is defined by a terrain patch and two functions that control the slope of rivers and valleys. B) The river network is automatically calculated and C.D) all inputs are then used to generate the continuous terrain conforming to rules from hydrology.

Abstract

We present a framework that allows quick and intuitive modeling of terrains using concepts inspired by hydrology. The terrain is generated from a simple initial sketch, and its generation is controlled by a few parameters. Our terrain representation is both analytic and continuous and can be rendered by using varying levels of detail. The terrain data are stored in a novel data structure: a construction tree whose internal nodes define a combination of operations, and whose leaves represent terrain features. The framework uses rivers as modeling elements, and it first creates a hierarchical drainage network that is represented as a geometric graph over a given input domain. The network is then analyzed to construct watersheds and to characterize the different types and trajectories of rivers. The terrain is finally generated by combining procedural terrain and river patches with blending and carving operators.

CR Categories: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling; I.3.6 [Computer Graphics]: Methodology and Techniques—Interaction Techniques I.6.8 [Simulation and Modeling]: Types of Simulation—Visual

Keywords: procedural modeling, terrain generation, hydrology

Links: [DL](#) [PDF](#) [WEB](#) [VIDEO](#)

*e-mail:eric.galin@liris.cnrs.fr

ACM Reference Format:
G  nevaux, J., Galin,   ., Gu  rin,   ., Peytavie, A., Ben  s, B. 2013. Terrain Generation Using Procedural Models based on Hydrology. ACM Trans. Graph. 32, 4, Article 143 (July 2013), 10 pages.
DOI = 10.1145/2461912.2461986 <http://doi.acm.org/10.1145/2461912.2461986>.

Copyright Notice:
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions.acm.org.
Copyright    ACM 0730-0220/13/07-ART 143 \$15.00.
DOI <http://doi.acm.org/10.1145/2461912.2461986>

Terrain Generation with Hydrology

Procedural Generation and Rendering of Realistic, Navigable Forest Environments: An Open-Source Tool

Callum Newlands^{  } and Klaus-Peter Zauner

Electronics and Computer Science, University of Southampton, UK
{cn2g18, kpa}@soton.ac.uk



Figure 1: Sample image frames of scenes generated by the tool during runs configured with four different parameters sets.

Abstract

Simulation of forest environments has applications from entertainment and art creation to commercial and scientific modelling. Due to the unique features and lighting in forests, a forest-specific simulator is desirable, however many current forest simulators are proprietary or highly tailored to a particular application. Here we review several areas of procedural generation and rendering specific to forest generation, and utilise this to create a generalised, open-source tool for generating and rendering interactive, realistic forest scenes. The system uses specialised L-systems to generate trees which are distributed using an ecosystem simulation algorithm. The resulting scene is rendered using a deferred rendering pipeline, a Blinn-Phong lighting model with real-time leaf transparency and post-processing lighting effects. The result is a system that achieves a balance between high natural realism and visual appeal, suitable for tasks including training computer vision algorithms for autonomous robots and visual media generation. The application and supporting configuration files can be found at <https://github.com/callumnewlands/ForestGenerator>.

CCS Concepts

• **Computing methodologies** → **Computer graphics**; **Rendering**; **Mesh models**;

1. Introduction

Procedural generation and efficient rendering of vegetation and natural environments is a growing research area in both the entertainment and commercial industries. Whilst the former is often concerned with visual attractiveness and generation speed [Mor16, Int20, Lan20], the latter often focuses on real-world accuracy and generation flexibility, for purposes such as environmental forecast modelling or simulating real-world locations [HSAKD98, Kom20, Len20, Vir20].

We here present an open-source Forest Generator (available with sample configurations at [New22]) for procedurally generating and rendering realistic forests which can be virtually navigated in real-time, or through asynchronous connections with external software.

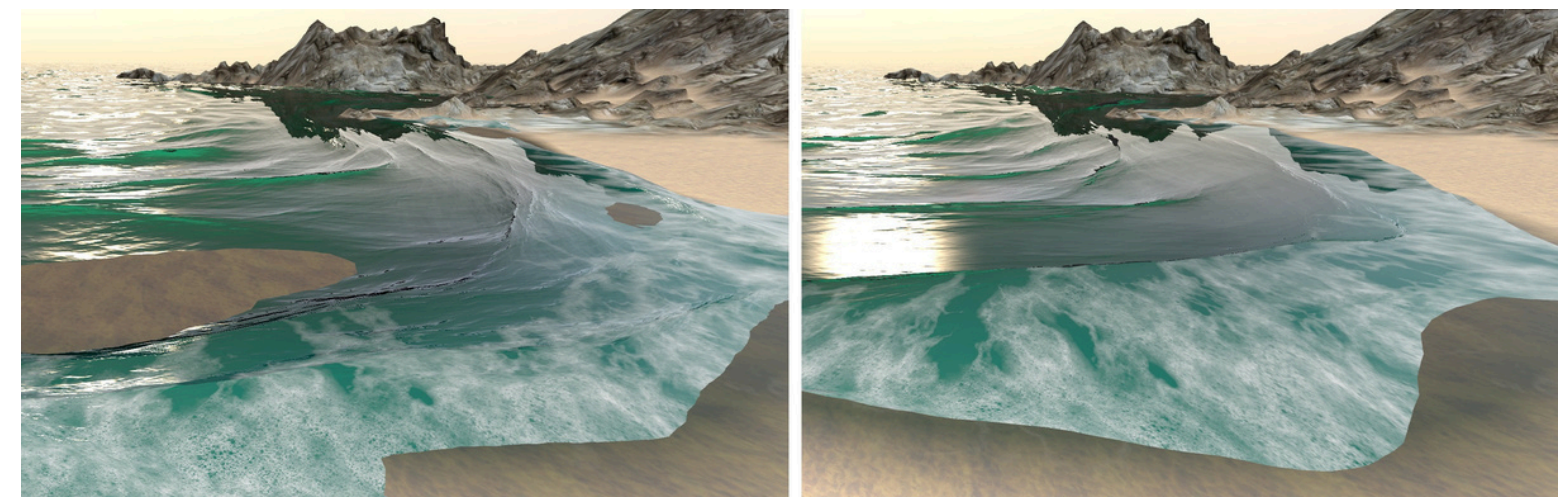
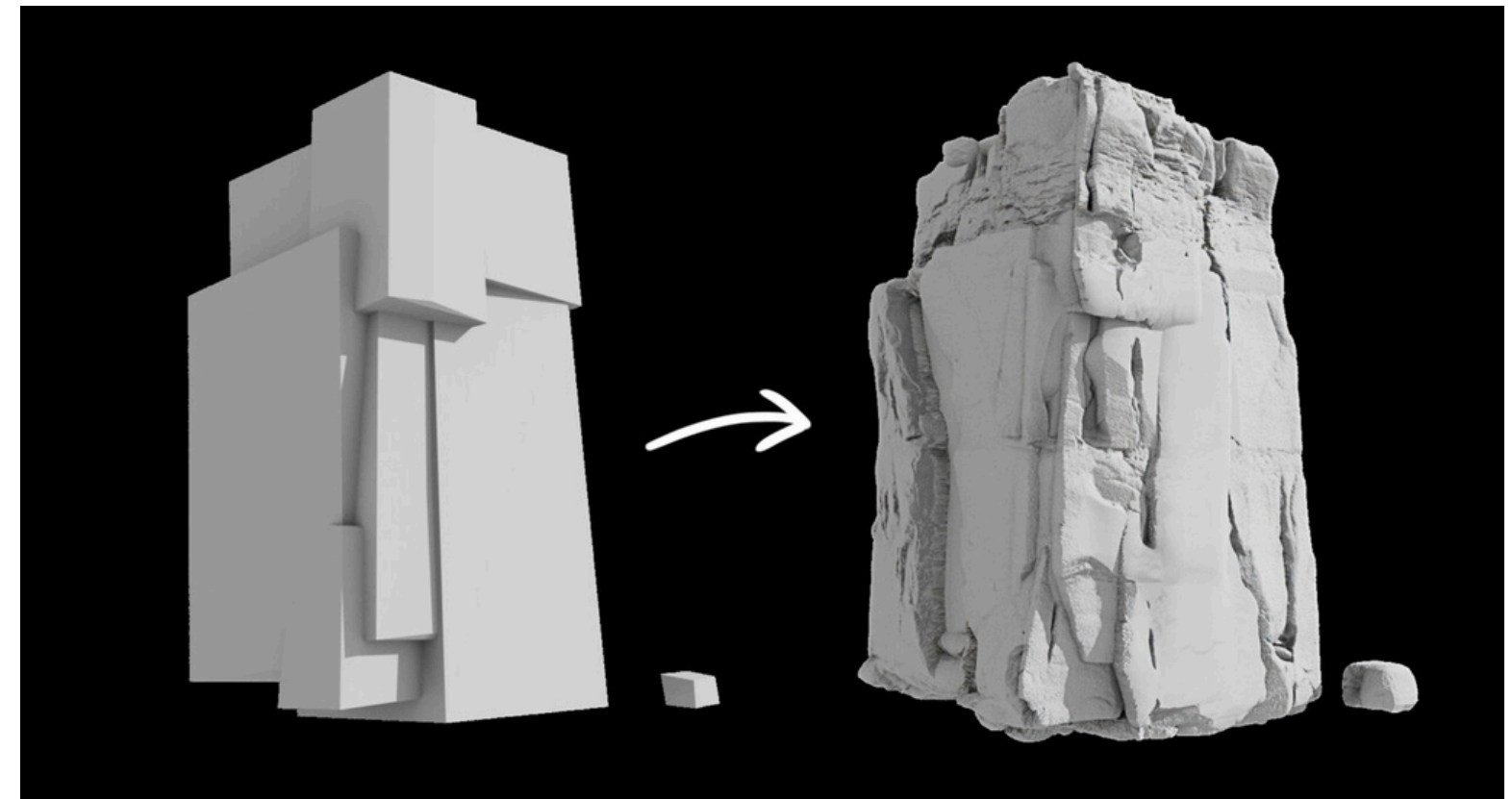
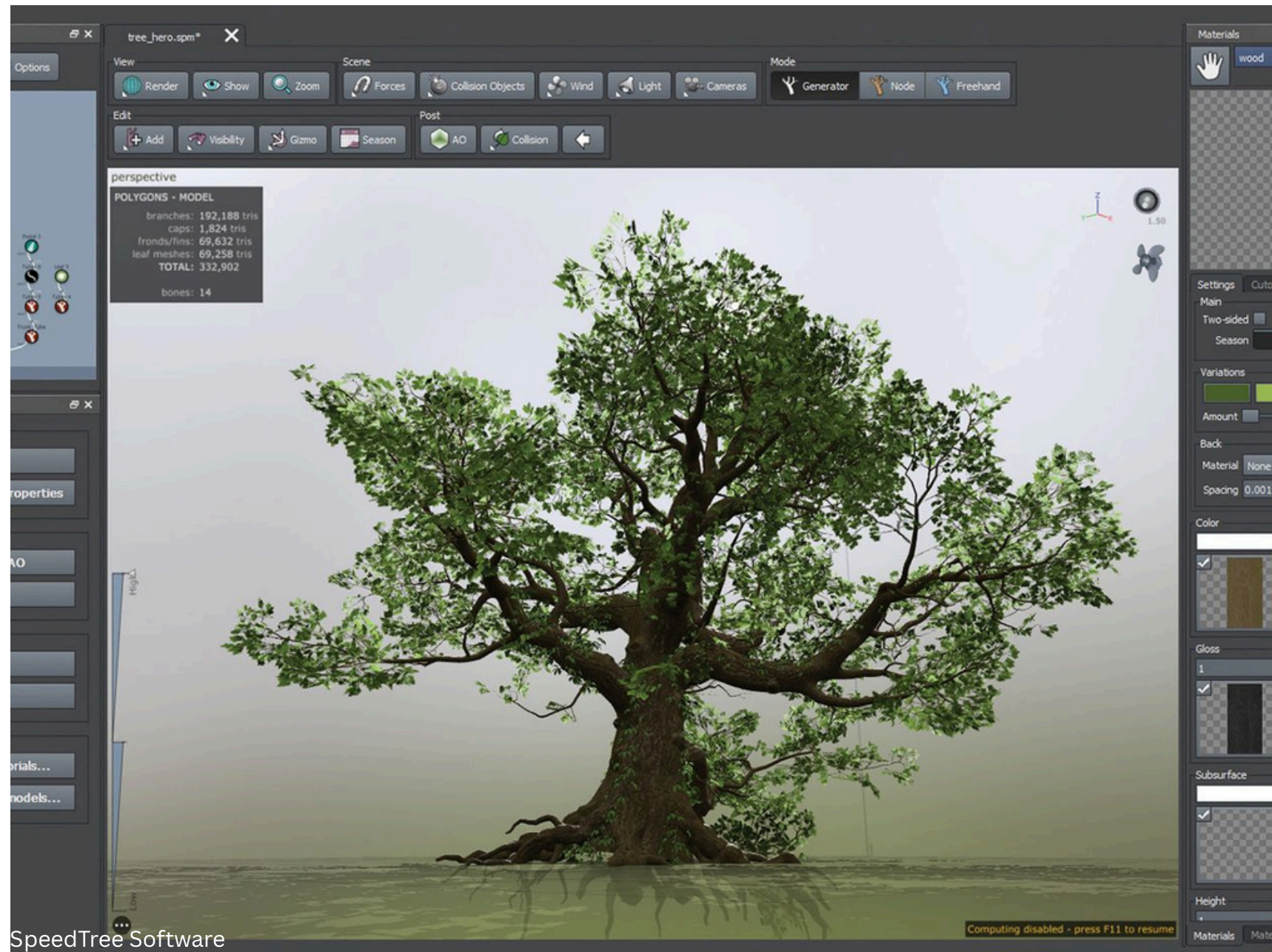
The system takes a middle-ground perspective between visual and real-world accuracy, resulting in an approach that is ideal, for example, for training the computer-vision systems of forest navigating robots [HRS13, SNK⁺20, TGZ20].

The main contributions of this article are two-fold, firstly, a brief review of the existing approaches and literature in several aspects of forest generation and rendering (Section 2)—including vegetation generation, rendering optimisation, ecosystem simulations and tree-specific illumination—and secondly, the presentation of a system for generating and rendering realistic forest environments. Section 3 gives an overview of the system architecture and control flow—which uses L-systems for representing and generating a wide range of tree types (Section 4), along with a tree-specific ecosystem simulation for realistic scene distributions (Sec-

Navigable Forest Generation

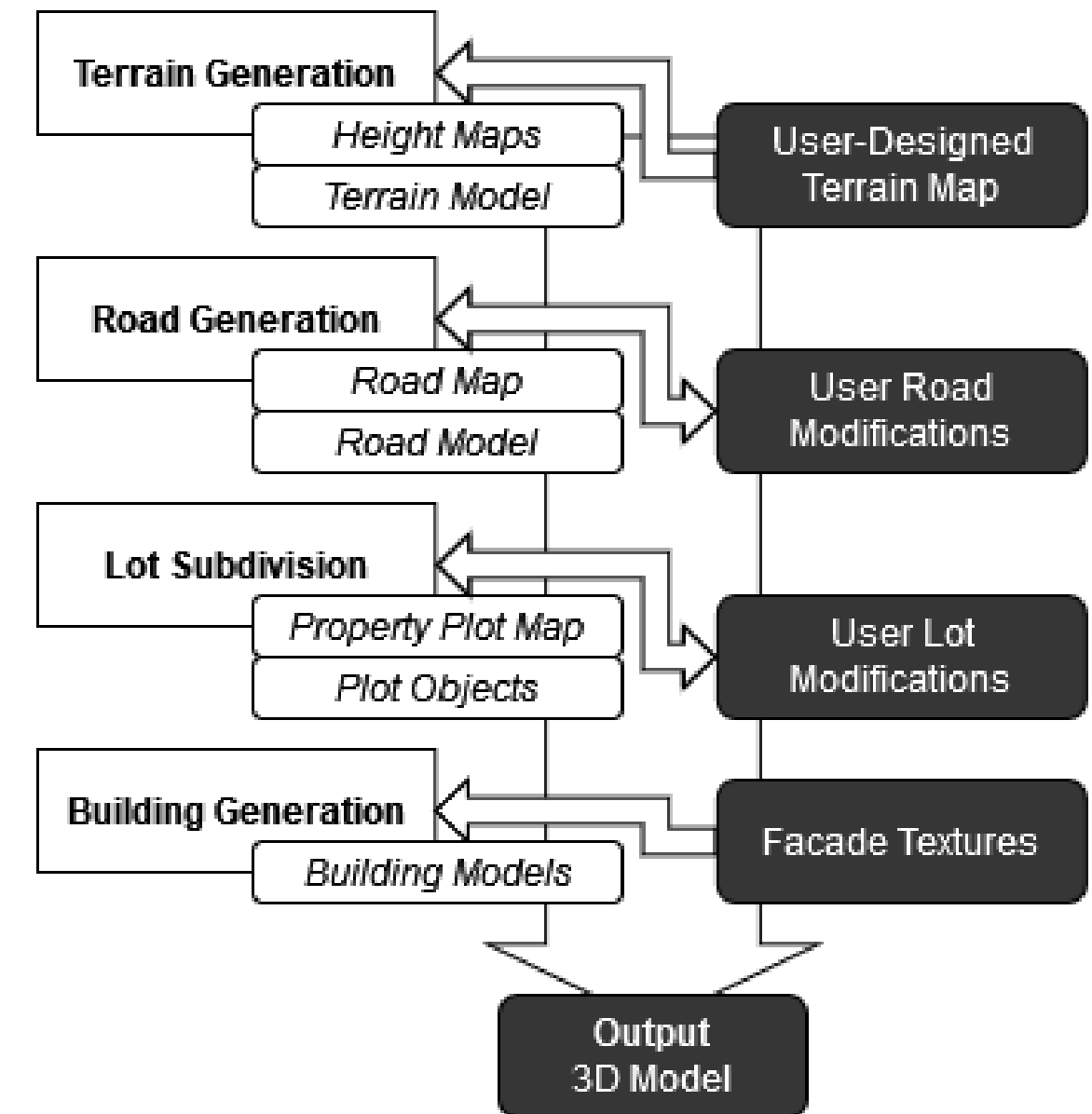
ProcGen in the Industry

How can this research field be applied?

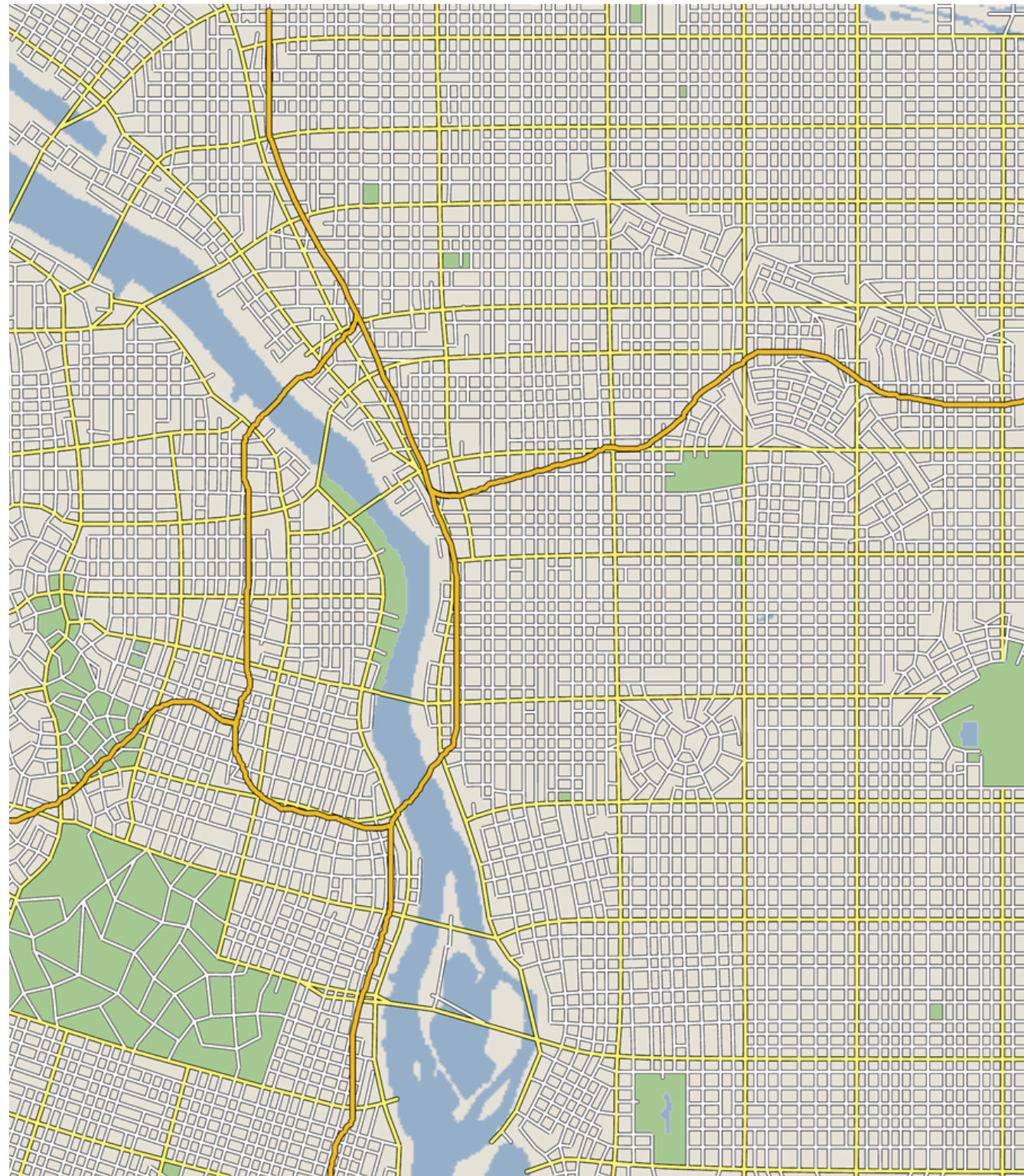


The Challenge? Cities!

City generation is one of the most complex procedural tasks, requiring the integration of multiple systems.



The Research Gap 1



“Interactive Procedural Street Modeling” from SIGGRAPH2008

VS



What differences can you spot between these 2 images?

The Research Gap 2



- Over 1 billion people live in informal settlements (SDG 11)
- Existing procedural tools assume formal planning—grids, zoning, uniformity.
- They fundamentally fail to capture the organic, irregular, and emergent morphology of slums.
- This creates a critical tooling void for urban planners, researchers, or modelers who may work on these slums

The Research Gap 3

Below is a list of existing city generators. All of them focus on urban spaces and structured spaces

01 Tensor Fields for Roads

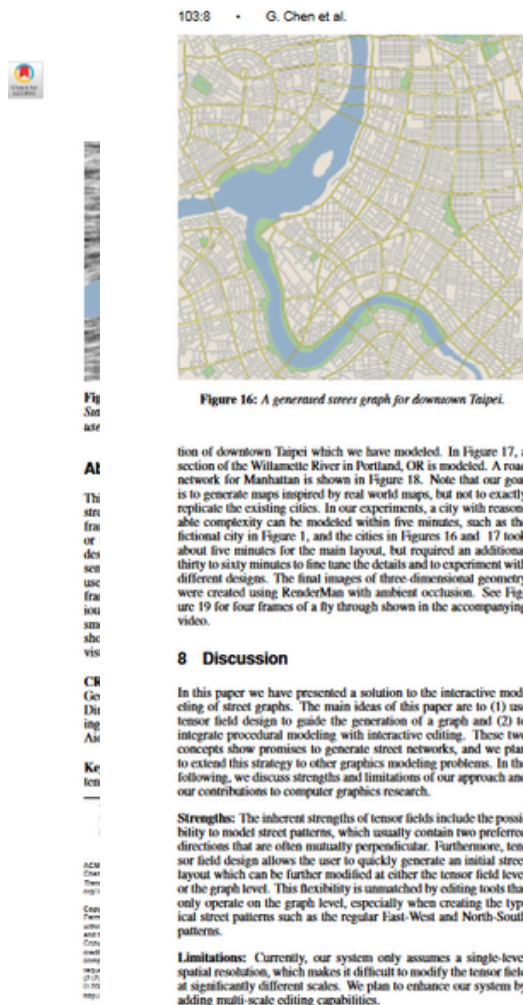


Figure 16: A generated street graph for downtown Taipei.

tion of downtown Taipei which we have modeled. In Figure 17, a section of the Willamette River in Portland, OR is modeled. A road network for Manhattan is shown in Figure 18. Note that our goal is to generate maps inspired by real world maps, but not to exactly replicate the existing cities. In our experiments, a city with reasonable complexity can be modeled within five minutes, such as the fictional city in Figure 1, and the cities in Figures 16 and 17 took about five minutes for the main layout, but required an additional thirty to sixty minutes to fine tune the details and to experiment with different designs. The final images of three-dimensional geometry were created using RenderMan with ambient occlusion. See Figure 19 for four frames of a fly through shown in the accompanying video.

8 Discussion

In this paper we have presented a solution to the interactive modeling of street graphs. The main ideas of this paper are to (1) use tensor field design to guide the generation of a graph and (2) to integrate procedural modeling with interactive editing. These two concepts show promises to generate street networks, and we plan to extend this strategy to other graphics modeling problems. In the following, we discuss strengths and limitations of our approach and our contributions to computer graphics research.

Strengths: The inherent strengths of tensor fields include the possibility to model street patterns, which usually contain two preferred directions that are often mutually perpendicular. Furthermore, tensor field design allows the user to quickly generate an initial street layout which can be further modified at either the tensor field level or the graph level. This flexibility is unmatched by editing tools that only operate on the graph level, especially when creating the typical street patterns such as the regular East-West and North-South patterns.

Limitations: Currently, our system only assumes a single-level spatial resolution, which makes it difficult to modify the tensor field at significantly different scales. We plan to enhance our system by adding multi-scale editing capabilities.

Lacks a full framework for lower density city generation

02 Fast Building Generation

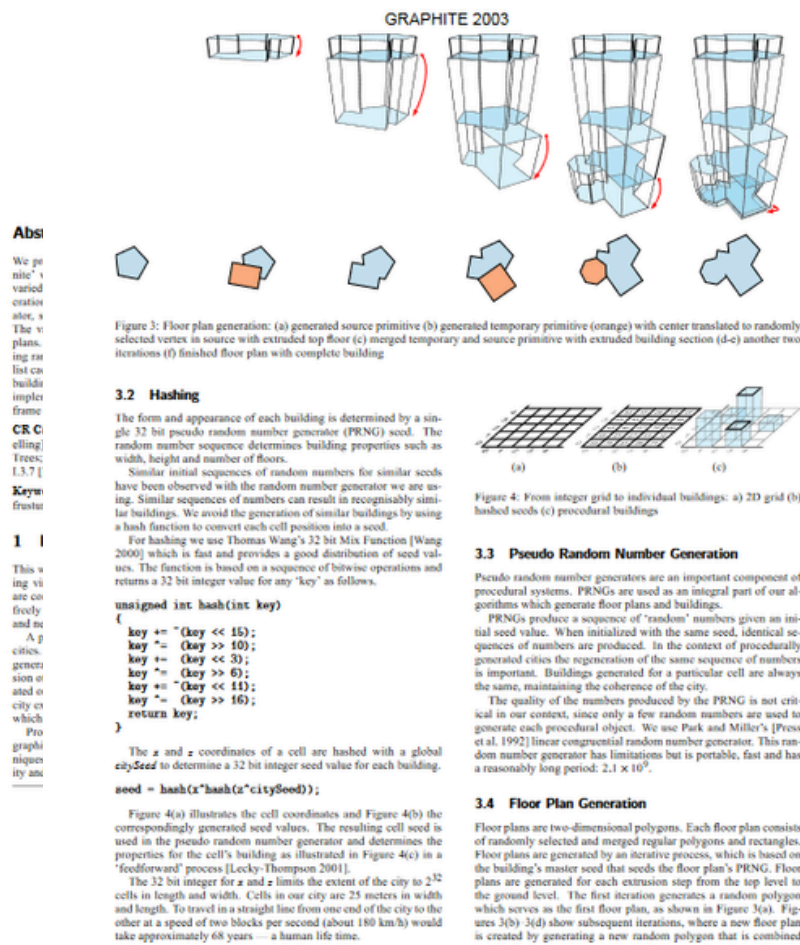


Figure 3: Floor plan generation: (a) generated source primitive (b) generated temporary primitive (orange) with center translated to randomly selected vertex in source with extruded top floor (c) merged temporary and source primitive with extruded building section (d-e) another two iterations (f) finished floor plan with complete building

3.2 Hashing

The form and appearance of each building is determined by a single 32 bit pseudo random number generator (PRNG) seed. The random number sequence determines building properties such as width, height and number of floors.

Similar initial sequences of random numbers for similar seeds have been observed with the random number generator we are using. Similar sequences of numbers can result in recognizably similar buildings. We avoid the generation of similar buildings by using a hash function to convert each cell position into a seed.

For hashing we use Thomas Wang's 32 bit Mix Function [Wang 2000] which is fast and provides a good distribution of seed values. The function is based on a sequence of bitwise operations and returns a 32 bit integer value for any "key" as follows.

```
unsigned int hash(int key)
{
    key += ~(key << 16);
    key ^= (key >> 10);
    key += (key << 3);
    key ^= (key >> 6);
    key += ~(key << 11);
    key ^= (key >> 16);
    return key;
}
```

The x and z coordinates of a cell are hashed with a global citySeed to determine a 32 bit integer seed value for each building.

seed = hash(x*hash(z*citySeed));

Figure 4(a) illustrates the cell coordinates and Figure 4(b) the correspondingly generated seed values. The resulting cell seed is used in the pseudo random number generator and determines the properties for the cell's building as illustrated in Figure 4(c) in a "feedforward" process [Locky-Thompson 2001].

The 32 bit integer for x and z limits the extent of the city to 2^{32} cells in length and width. Cells in our city are 25 meters in width and length. To travel in a straight line from one end of the city to the other at a speed of two blocks per second (about 180 km/h) would take approximately 68 years — a human life time.

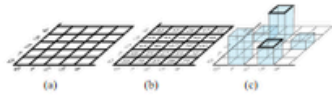


Figure 4: From integer grid to individual buildings: (a) 2D grid (b) hashed seeds (c) procedural buildings

3.3 Pseudo Random Number Generation

Pseudo random number generators are an important component of procedural systems. PRNGs are used as an integral part of our algorithms which generate floor plans and buildings.

PRNGs produce a sequence of "random" numbers given an initial seed value. When initialized with the same seed, identical sequences of numbers are produced. In the context of procedurally generated cities the regeneration of the same sequence of numbers is important. Buildings generated for a particular cell are always the same, maintaining the coherence of the city.

The quality of the numbers produced by the PRNG is not critical in our context, since only a few random numbers are used to generate each procedural object. We use Park and Miller's [Press et al. 1992] linear congruential random number generator. This random number generator has limitations but is portable, fast and has a reasonably long period: 2.1×10^9 .

3.4 Floor Plan Generation

Floor plans are two-dimensional polygons. Each floor plan consists of randomly selected and merged regular polygons and rectangles. Floor plans are generated by an iterative process, which is based on the building's master seed that seeds the floor plan's PRNG. Floor plans are generated for each extrusion step from the top level to the ground level. The first iteration generates a random polygon which serves as the first floor plan, as shown in Figure 3(a). Figures 3(b) 3(d) show subsequent iterations, where a new floor plan is created by generating a new random polygon that is combined

03 Fully Fledged System

4 Building Construction

The building construction stage of Citygen is accomplished in three stages. Firstly the enclosed regions are extracted from the secondary road graph by applying the Minimum Cycle Basis algorithm as described previously in Section 2. Secondly the lots are identified by splitting the regions into minimal tracts or parcels of land suitable for development. Thirdly and finally the buildings are selected based on their control parameters and are constructed in the boundaries of the lots and textured accordingly.

4.1 Blocks

Blocks represent the enclosed regions of the secondary road network. The role of the block is to add any extra geometry such as footpaths, signposts, traffic lights or post boxes onto the region before lot subdivision occurs. Currently only footpaths are implemented by applying a simple inset to the boundary polygon.

4.2 Lot Subdivision

The lot division process operates on the region boundary defined in the city block. A subdivision algorithm is applied based on that described by Moller et al. in [15] but without the requirement for regions to be convex. Both concave and convex boundaries are supported in our implementation and as a result it is possible to extract valid lots from complex regions and even facilitate the presence of filament roads. This enables us to apply the technique to suburban areas and support filament roads.

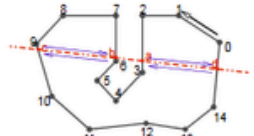


Figure 12: Lot Division Algorithm

The algorithm works by subdividing each region and operates recursively until a target lot size is reached. Each split operation is executed using the perpendicular of the longest side. Steps required to complete a single region split are as follows:

1. Identify the longest side beyond the target lot size and calculate the perpendicular from a deviated midpoint to split the polygon. The deviation introduces reproducible noise for more natural looking lots.
2. Calculate the intersection points between the line and the boundary and store them in the order encountered with each polygon index included. (a,2, b,7, c,9, d,14)
3. Process the intersection segments enclosing those outside the polygon and construct a data structure to provide a graph for the polygon. (c=b, a=d)
4. Merge the intersection graph with the polygon and extract the cycles from the graph.

The lot division algorithm works fine for Manhattan like regions where the input data is generally regular and block shaped. However when the algorithm was initially tested on suburban road networks the resulting lots were angular and irregular. To combat

this and provide realistic lots for suburban networks a modification was made to the lot division process in which division was first prioritized along sides with road access. As a result of this change almost all lots were oriented perpendicular to their access road reflecting the conditions found in the real world. In a similar fashion the value of lots for development is calculated taking appearances into account as used in the real-estate industry. Lots without road access are not considered suitable for building development and are discarded or destined for green space.

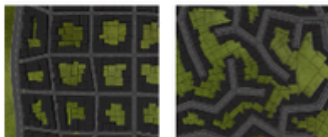


Figure 13: Down-town and Suburban lots in Deling View

4.3 Building Generation

Buildings are generated on the lots created by the lot subdivision process. Hints are attached to each neighbourhood as part of the control parameter set and advise the building generation algorithm on what class of building to be generated and how it should be positioned on the lot. Down-town buildings attempt to make maximum use of lot space while suburban buildings retreat in from their road access side and also from each other to make room for a garden area. Industrial building retreat in from their boundaries to emulate the green space found in industrial estates.

Building construction begins with the positioning of the building footprint on the lot by in-setting the polygon boundary of the lot. After the footprint has been calculated it is then extruded upwards to produce a solid object. The geometry of the buildings is very basic but with the aid of normal maps we can simulate extra geometry on the building. Figure 14 shows a typical texture tile and associated normal map for a building.

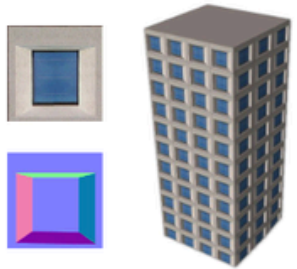


Figure 14: Building Texture, Normal Map and the resulting building after the material has been applied

Utilizes generative rule sets extensively, resulting in less finer user control



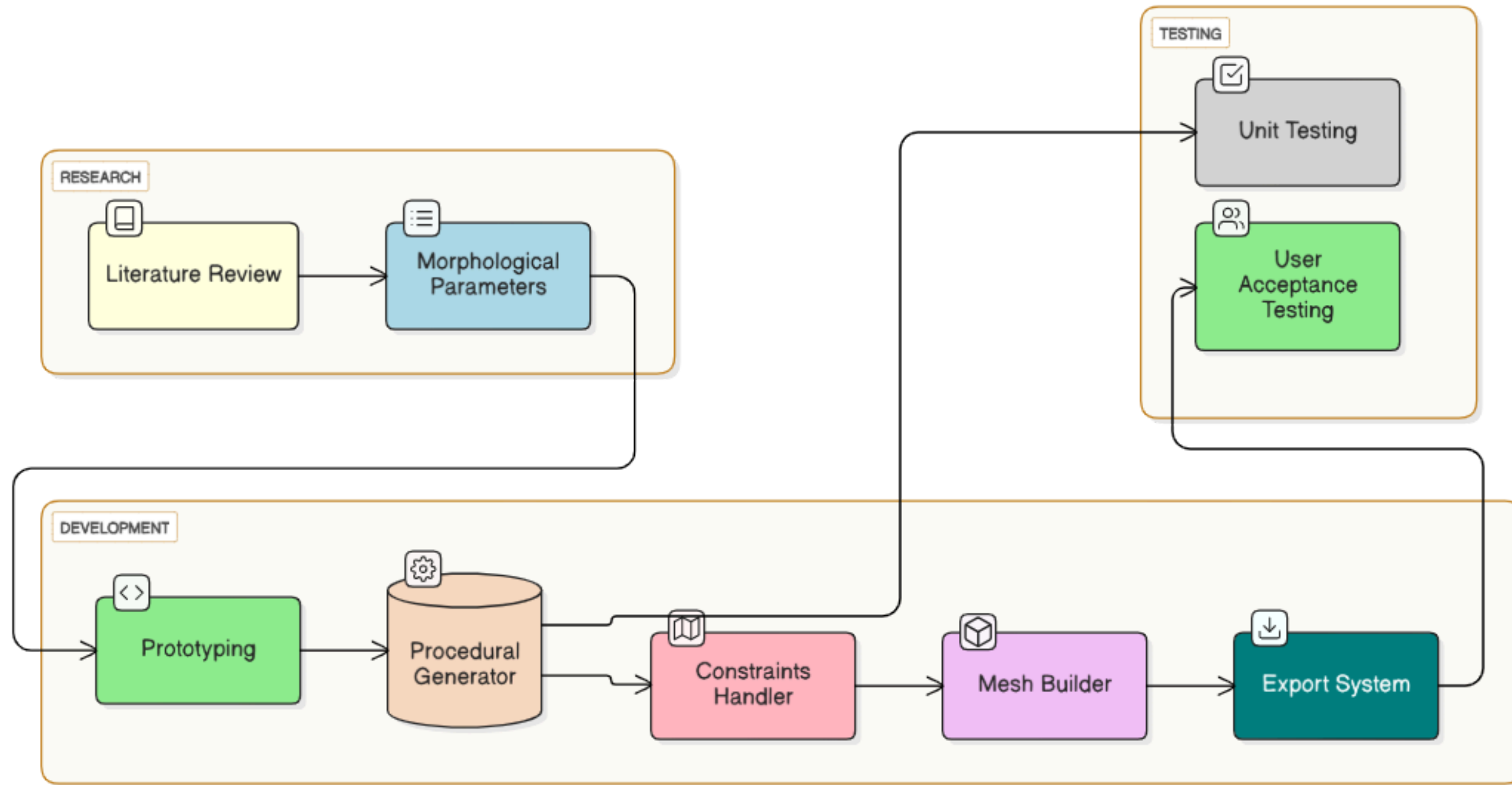
Problem Statement

The current state-of-the-art in procedural urban generation cannot realistically simulate slums.



Objectives

- To analyze and extract spatial and morphological patterns from existing slum research and datasets.
- To design and implement a procedural generation system that simulates slum-like settlement patterns using rule-based logic.
- To enable user-defined spatial inputs (e.g., road networks, barriers) as constraints within the generation process to guide slum formation.
- To evaluate the realism and variability of generated layouts against documented case studies or morphological benchmarks.



Project Plan

Research and Data Synthesis

Parameters were derived from a systematic analysis of 10 key academic studies as foundation values for my model

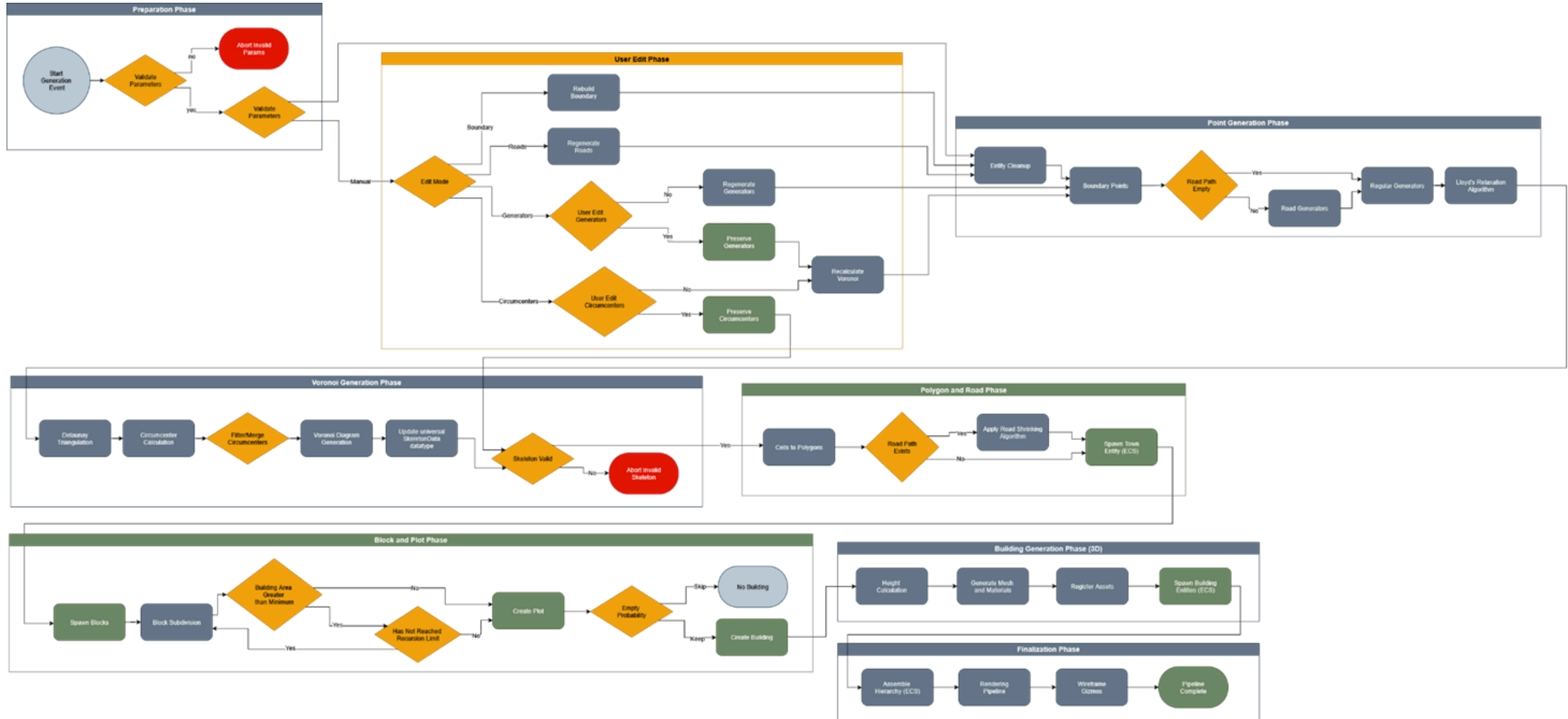
Focus Areas: Building Density, Street Geometry, Clustering, and Textural Variability.

Methods Used: Object-Oriented Image Analysis (OOA), Satellite Imagery (Quickbird, IKONOS), SAR data, and Spatial Metrics

Morphological Category	Key Finding from Literature	*Model Implementation*
Building & Footprint Density	• Very high density (Klotz et al.)	• Cell Fill Probability: 65-85%
	• Patch sizes: 0.25 - 198 ha (Wurm et al.)	• Voronoi Clustering: Log-normal distribution for patch sizes
Street & Alley Geometry	• Irregular, organic patterns (Klotz, Kuffer)	• Organic Road Generation
	• Narrow alleys: 0.5m - 1.5m wide (Wurm et al.)	• Alley Width Parameter: 0.5 - 1.5m
Patch Clustering Behavior	• High clustering (15-20%) (Klotz et al.)	• Cluster Density Parameter: 15-20%
	• Aggregation within 10m (Gruebner et al.)	• Cluster Distance: 10m aggregation rule
	• Lack of green space	• Minimal "Empty" Lots
Textural Variability	• High variability in roof types & materials (Klotz, Kuffer)	• Algorithm for Irregular Footprints
	• Irregular building shapes	• Variable dimensions, angles, and fragmentation

Algorithm Design 1

(Outside of Prototyping)

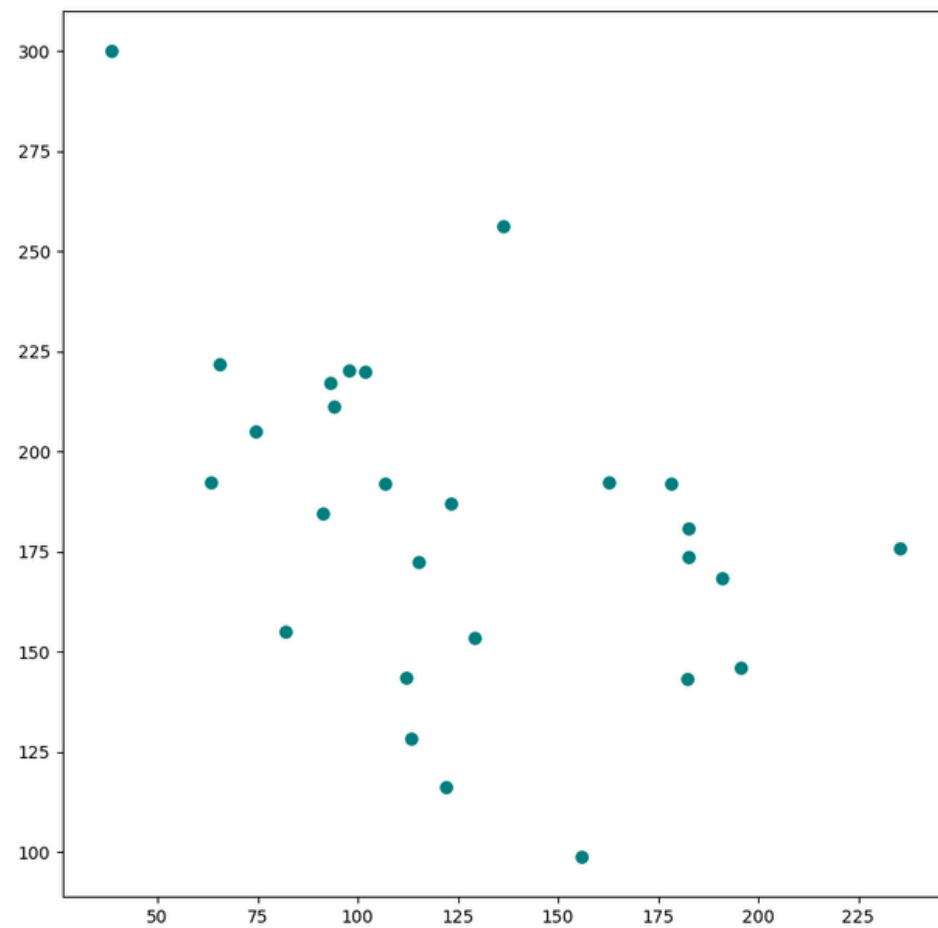


Algorithm Design 2

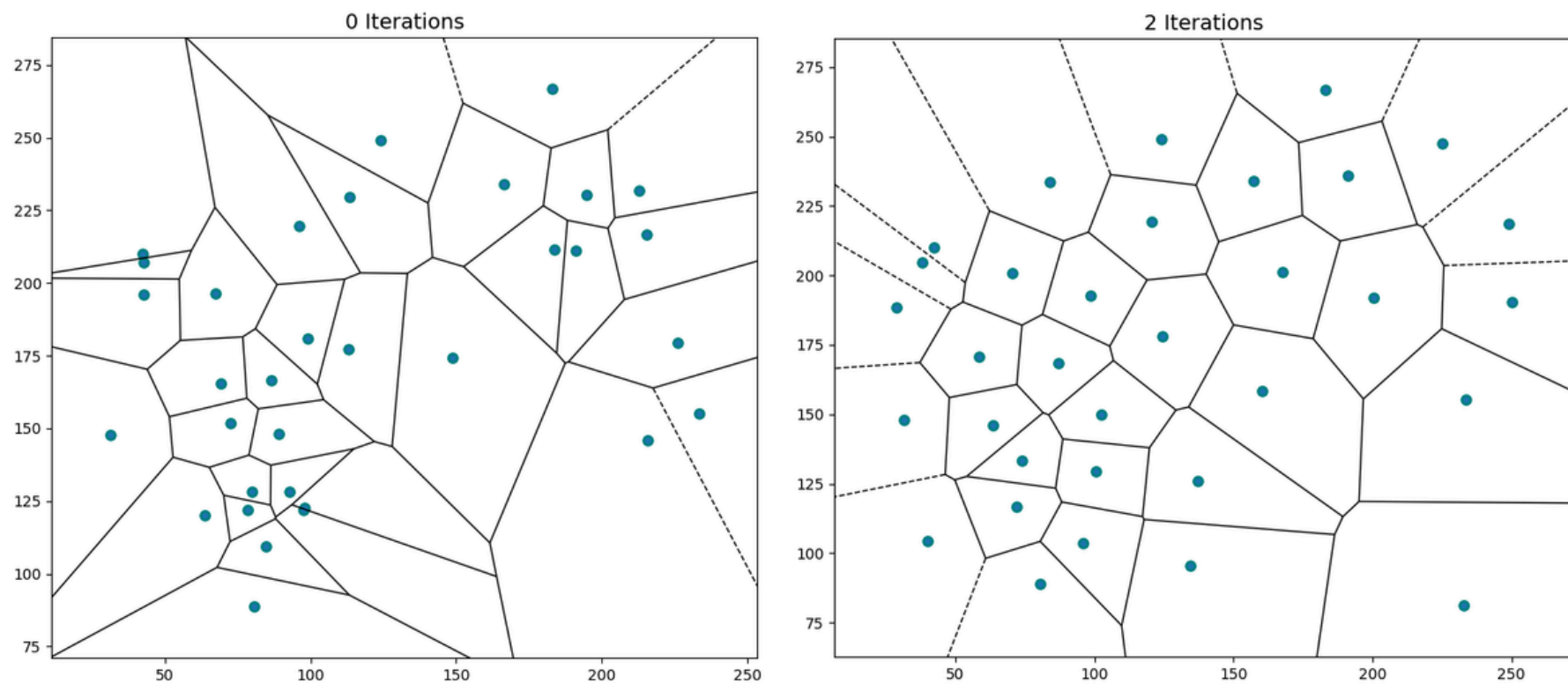
The algorithm transforms simple points into complex, organic settlements through a multi-stage spatial subdivision process. Illustrated through the Python prototype:

1. **Input:** User-defined Boundary & Parameters.
2. **Seeding & Relaxation:** Generate & relax points (Lloyd's algorithm).
3. **Voronoi Skeleton:** Create the cellular structure.
4. **Recursive Subdivision:** Split cells into individual building plots.
5. **3D Mesh Generation:** Extrude footprints into volumetric buildings.
6. **Output:** 3D Model ready for export.

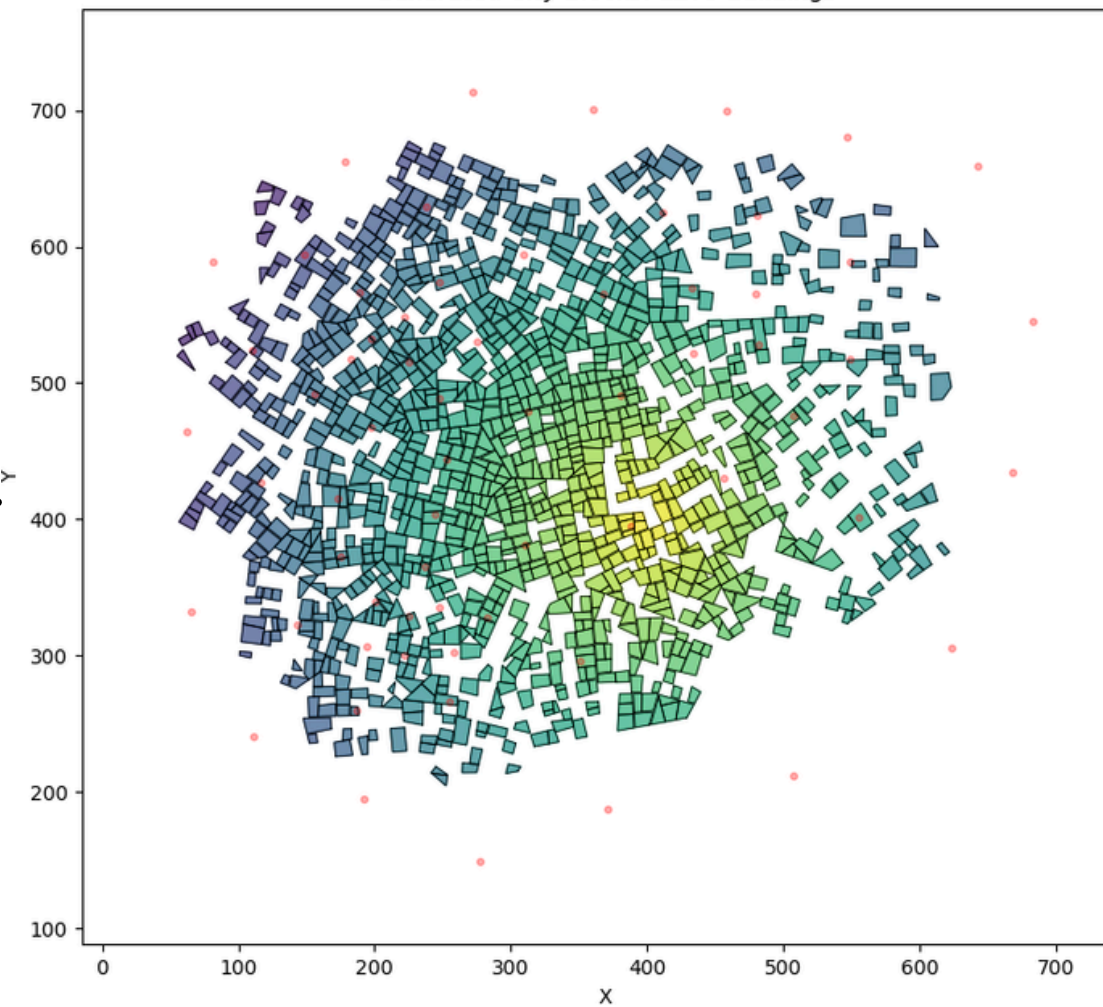
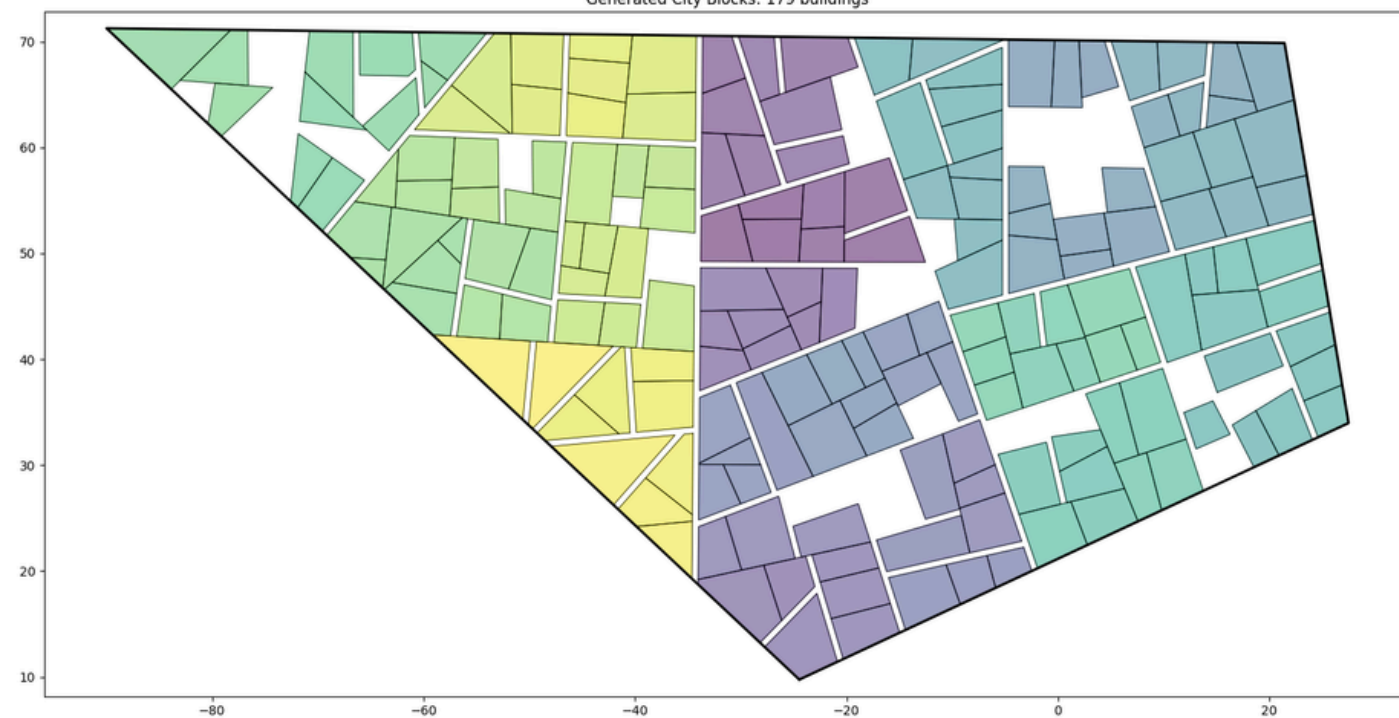
Site Distributions from Multi-Cluster Model

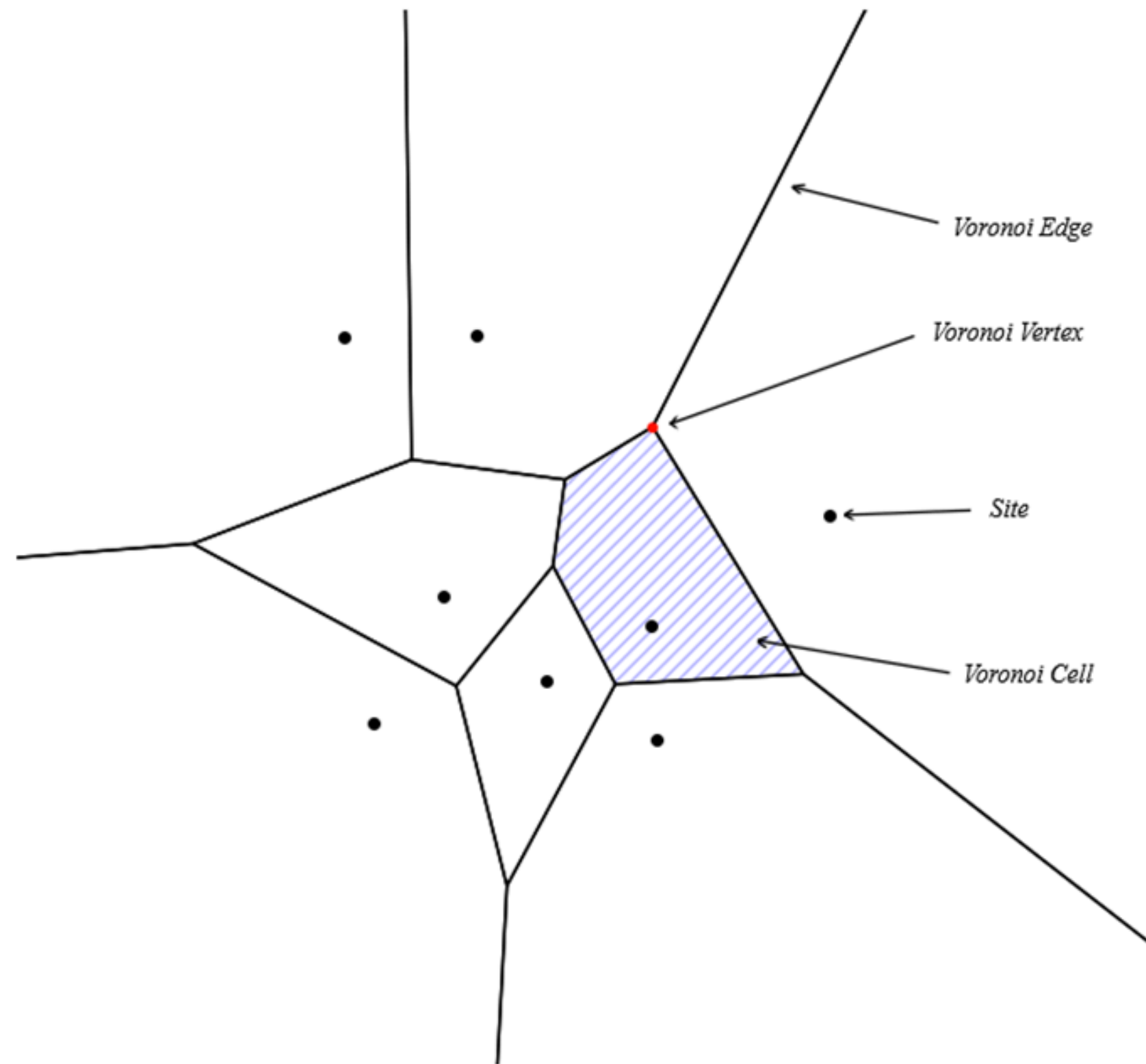


Lloyds Relaxation Algorithm

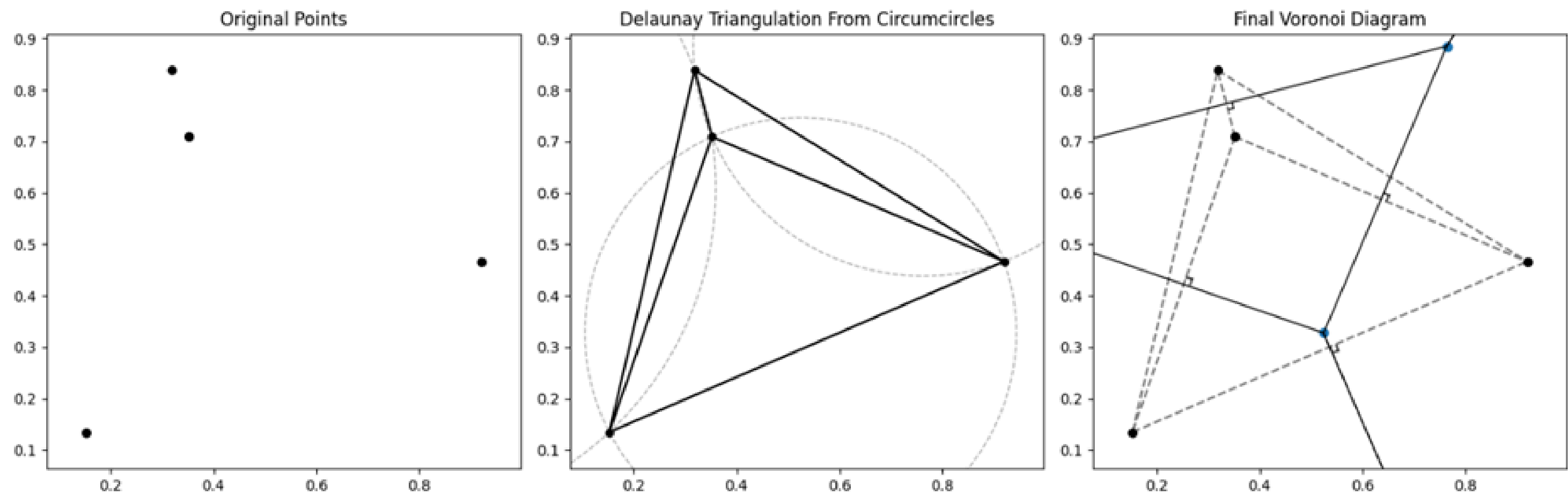


Generated City Blocks: 179 buildings

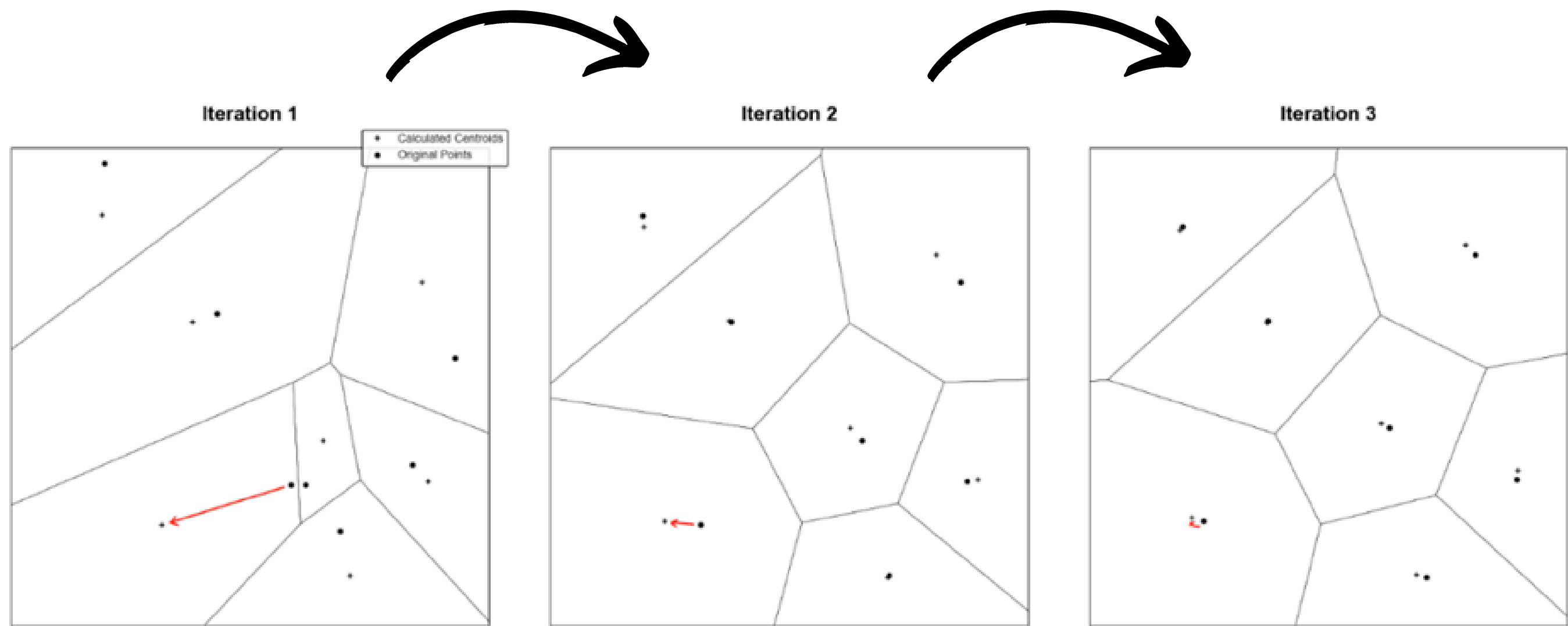




Components of a Voronoi Diagram



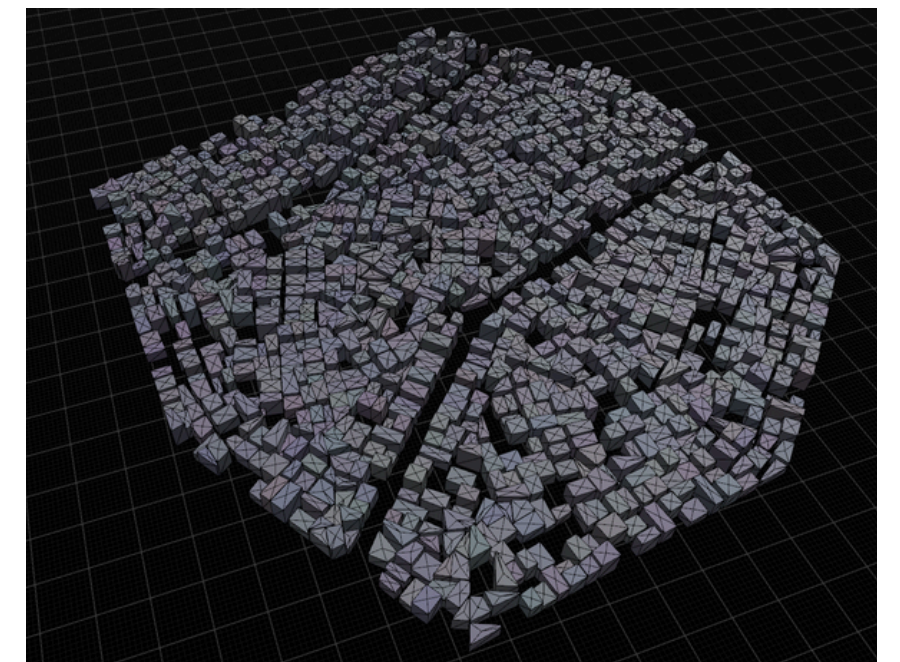
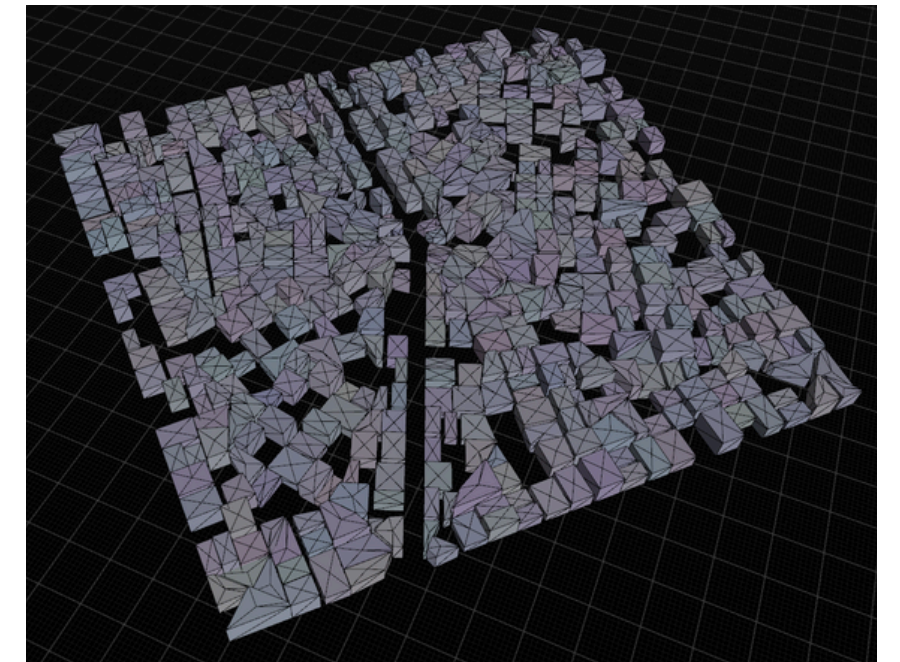
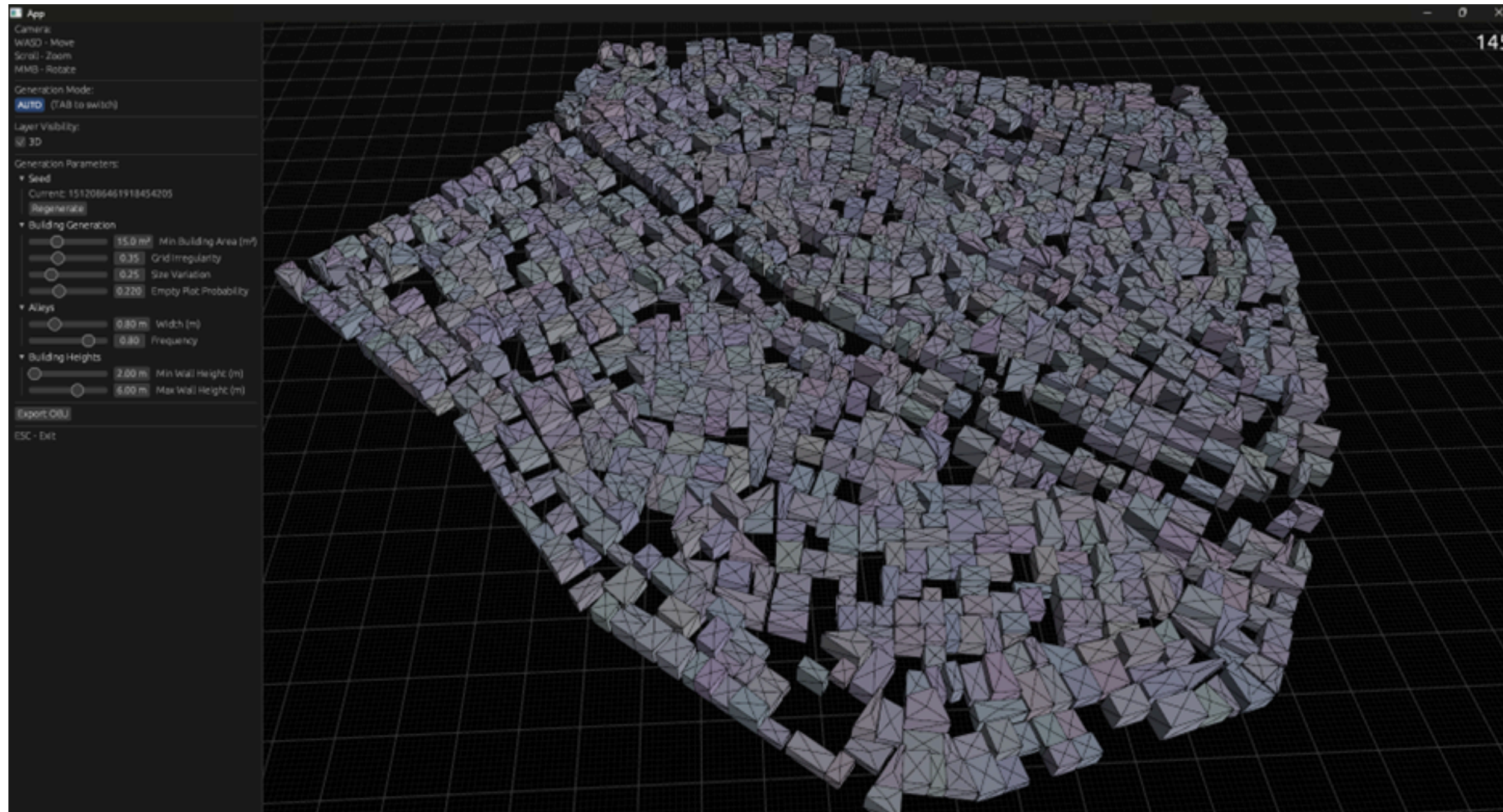
Delaunay Triangulation Algorithm



Lloyd's Relaxation Algorithm

My Software

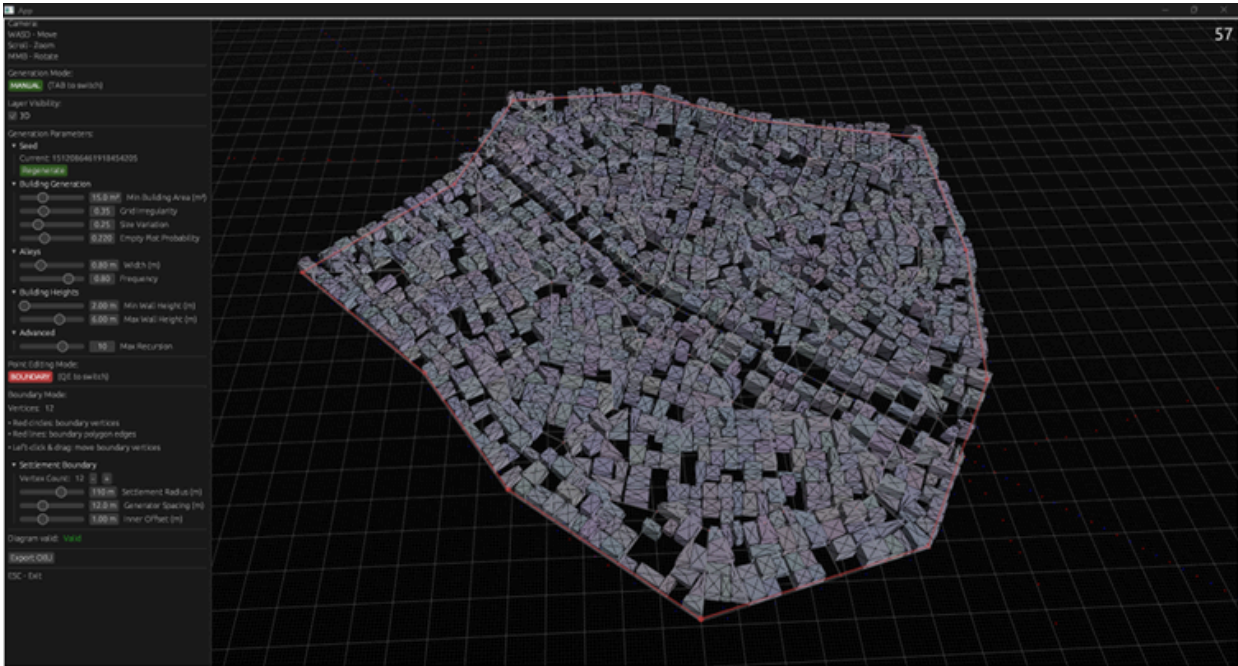
Instant Generation at Your Fingertips!



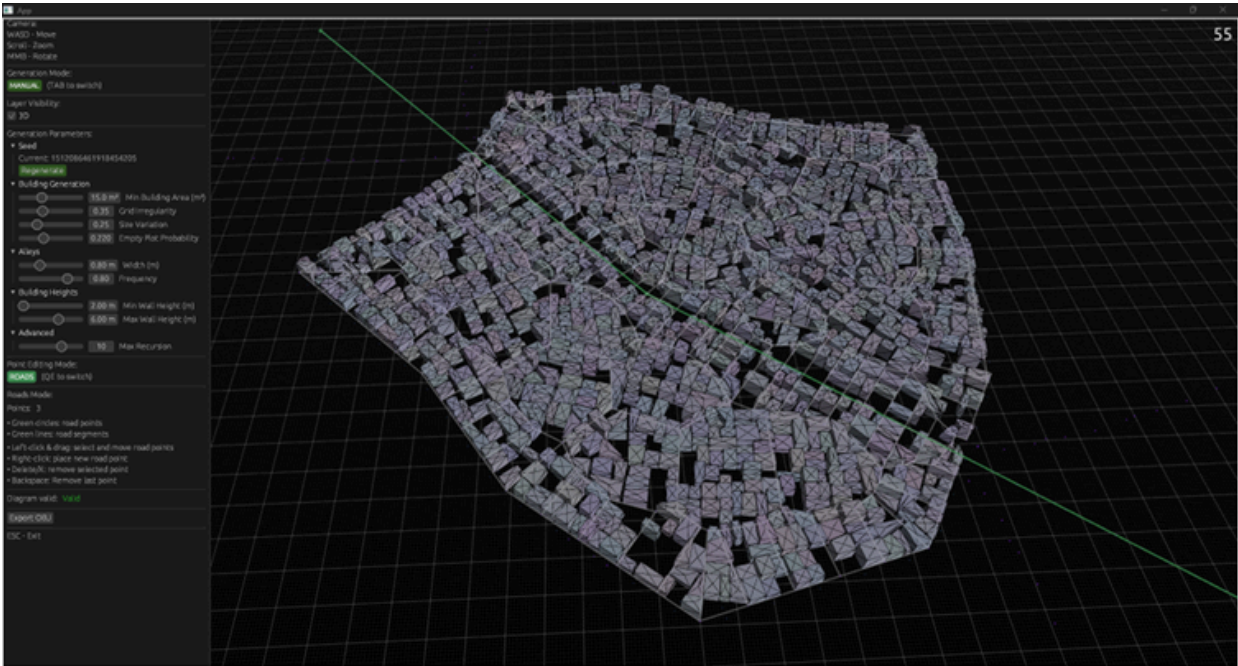
Basic-Mode UI

Multiple Edit Modes

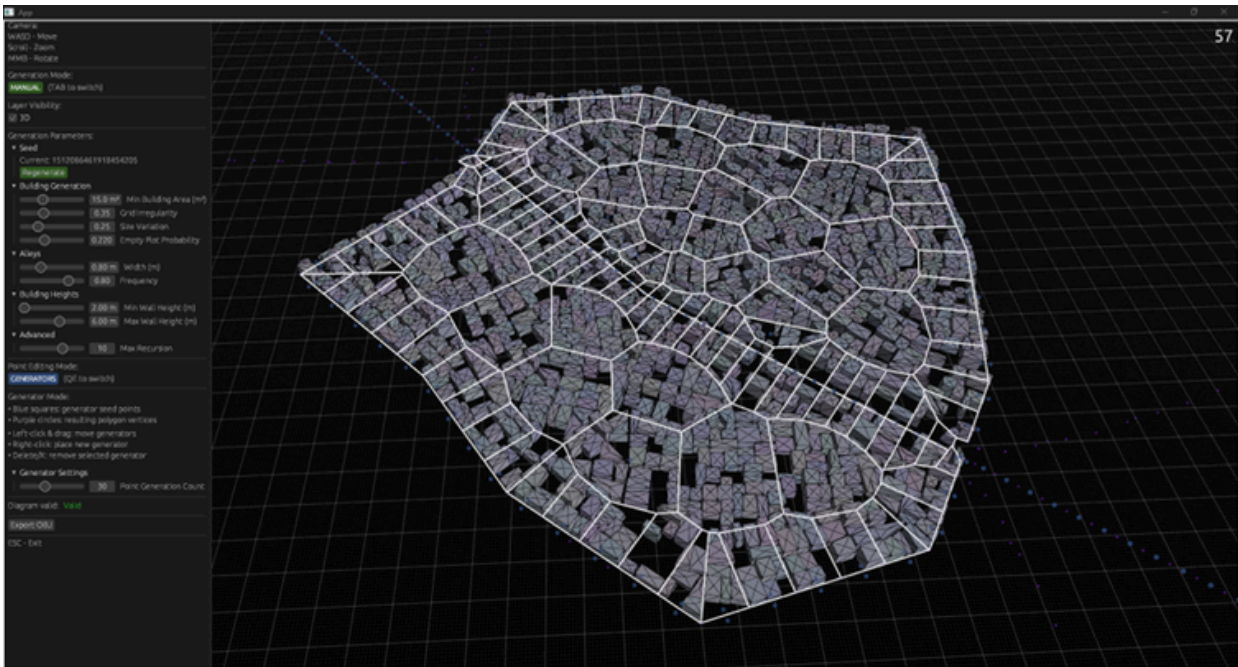
Advanced User Control Capabilities



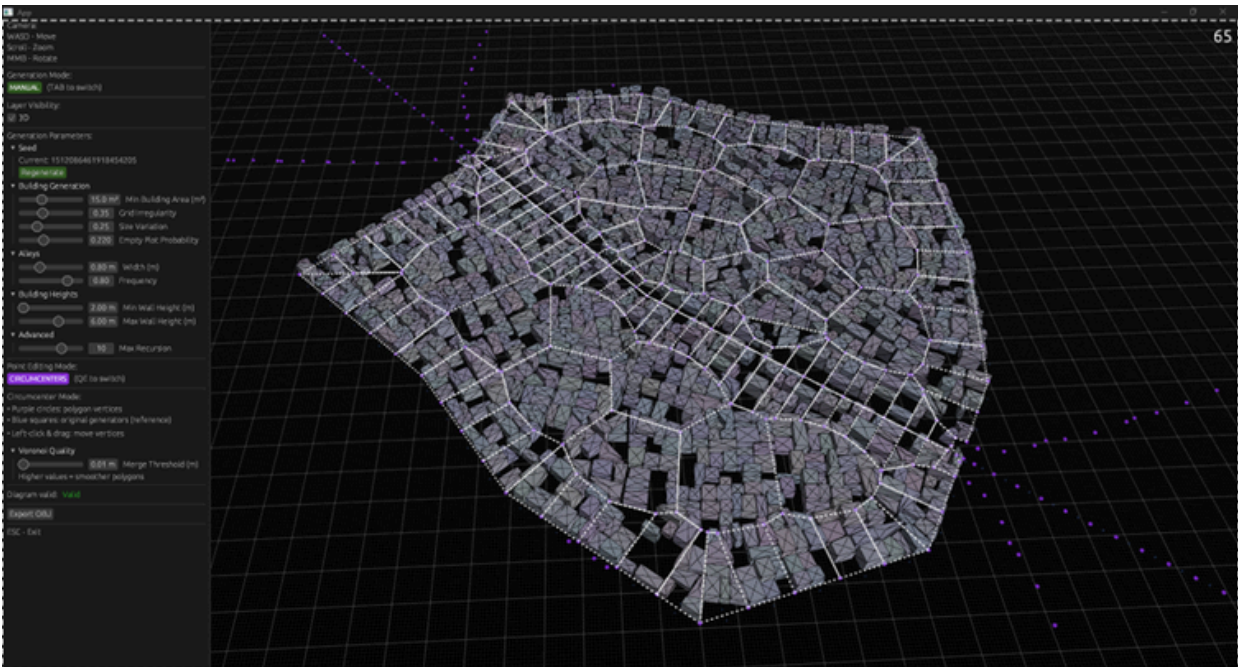
Boundary Mode



Road Mode



Seed Mode



Circumcenter Mode

High-Level

Low-Level



Demo Time!

Live Software Demonstration

Unit Testing

Unit Testing was validated through 25 suites covering the core modules. Rust's built-in unit test functionalities were used

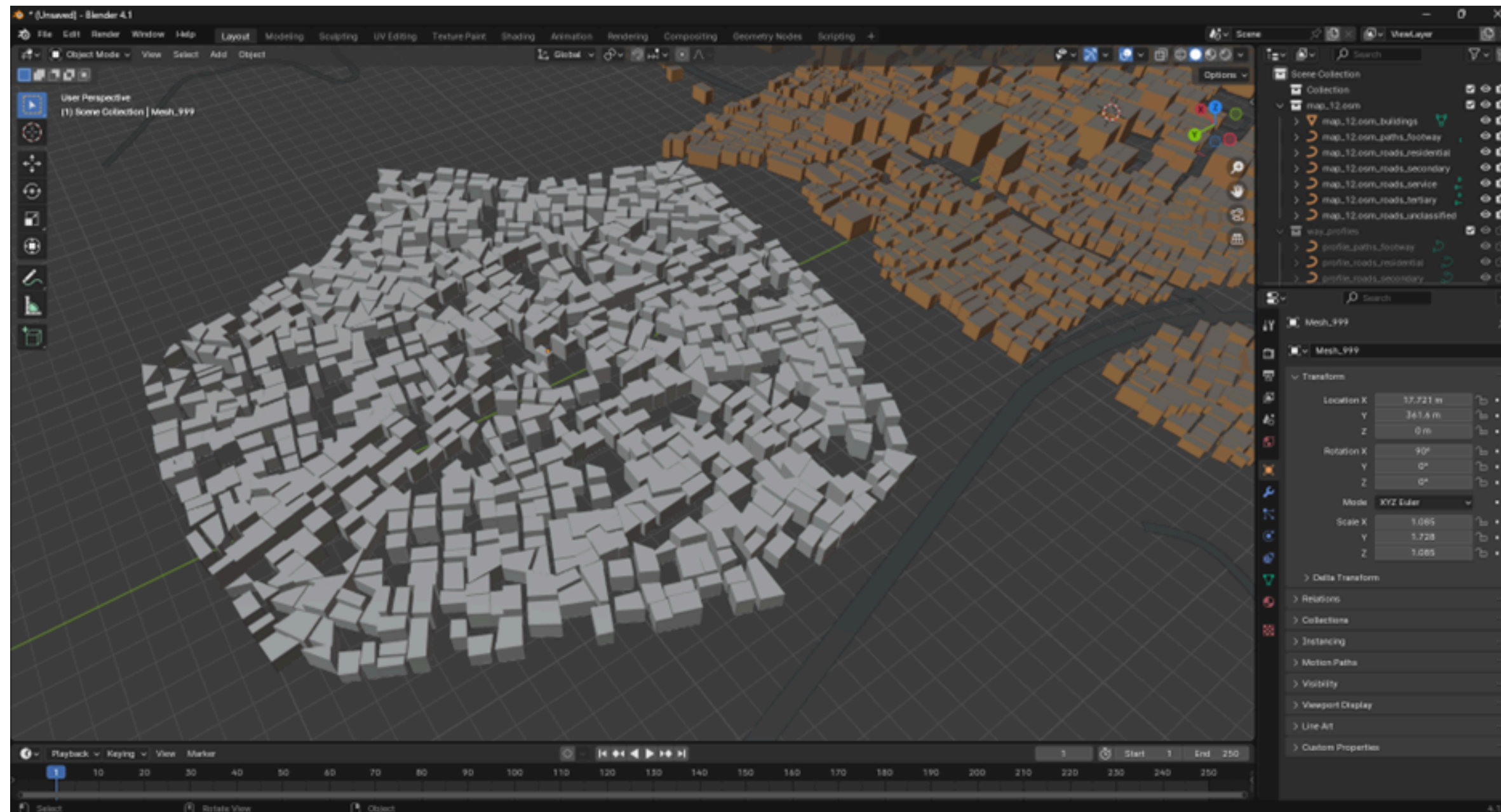
```
running 25 tests
test test::boundary::tests::test_boundary_generator_placement_constraints ... ok
test test::boundary::tests::test_generate_boundary_generators_creates_points ... ok
test test::export::tests::test_mesh_creation ... ok
test test::export::tests::test_mesh_has_positions ... ok
test test::boundary::tests::test_generate_road_generators_creates_points ... ok
test test::boundary::tests::test_generate_road_generators_short_path ... ok
test test::boundary::tests::test_generate_boundary_polygon_minimum_vertices ... ok
test test::export::tests::test_mesh_has_indices ... ok
test test::export::tests::test_mesh_vertex_count ... ok
test test::boundary::tests::test_generate_boundary_polygon_creates_vertices ... ok
test test::boundary::tests::test_generate_boundary_generators_empty_boundary ... ok
test test::mesh::tests::test_empty_polygon_has_zero_area ... ok
test test::mesh::tests::test_polygon_area_calculation ... ok
test test::export::tests::test_export_header_format ... ok
test test::mesh::tests::test_polygon_centroid_calculation ... ok
test test::export::tests::test_obj_vertex_format ... ok
test test::mesh::tests::test_polygon_to_building_creates_3d_mesh ... ok
test test::mesh::tests::test_polygon_to_layer_zero_creates_mesh ... ok
test test::mesh::tests::test_polygon_to_layer_zero_empty_polygon ... ok
test test::voronoi::tests::test_point_in_polygon_inside ... ok
test test::voronoi::tests::test_point_in_polygon_on_edge ... ok
test test::voronoi::tests::test_point_in_polygon_outside ... ok
test test::voronoi::tests::test_voronoi_cells_have_minimum_points ... ok
test test::voronoi::tests::test_vpoly_creates_skeleton_data ... ok
test test::voronoi::tests::test_vpoly_handles_single_generator ... ok

test result: ok. 25 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.01s

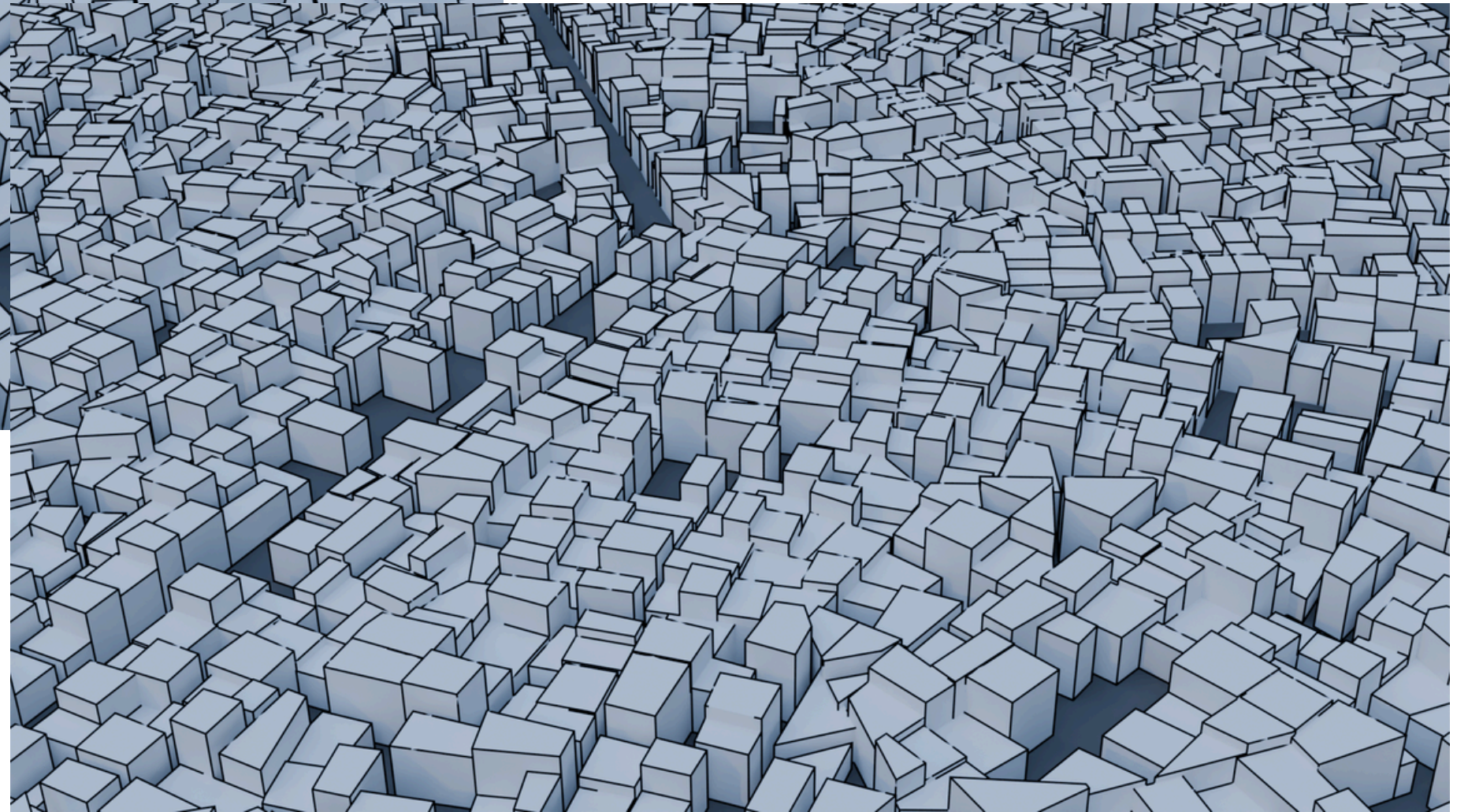
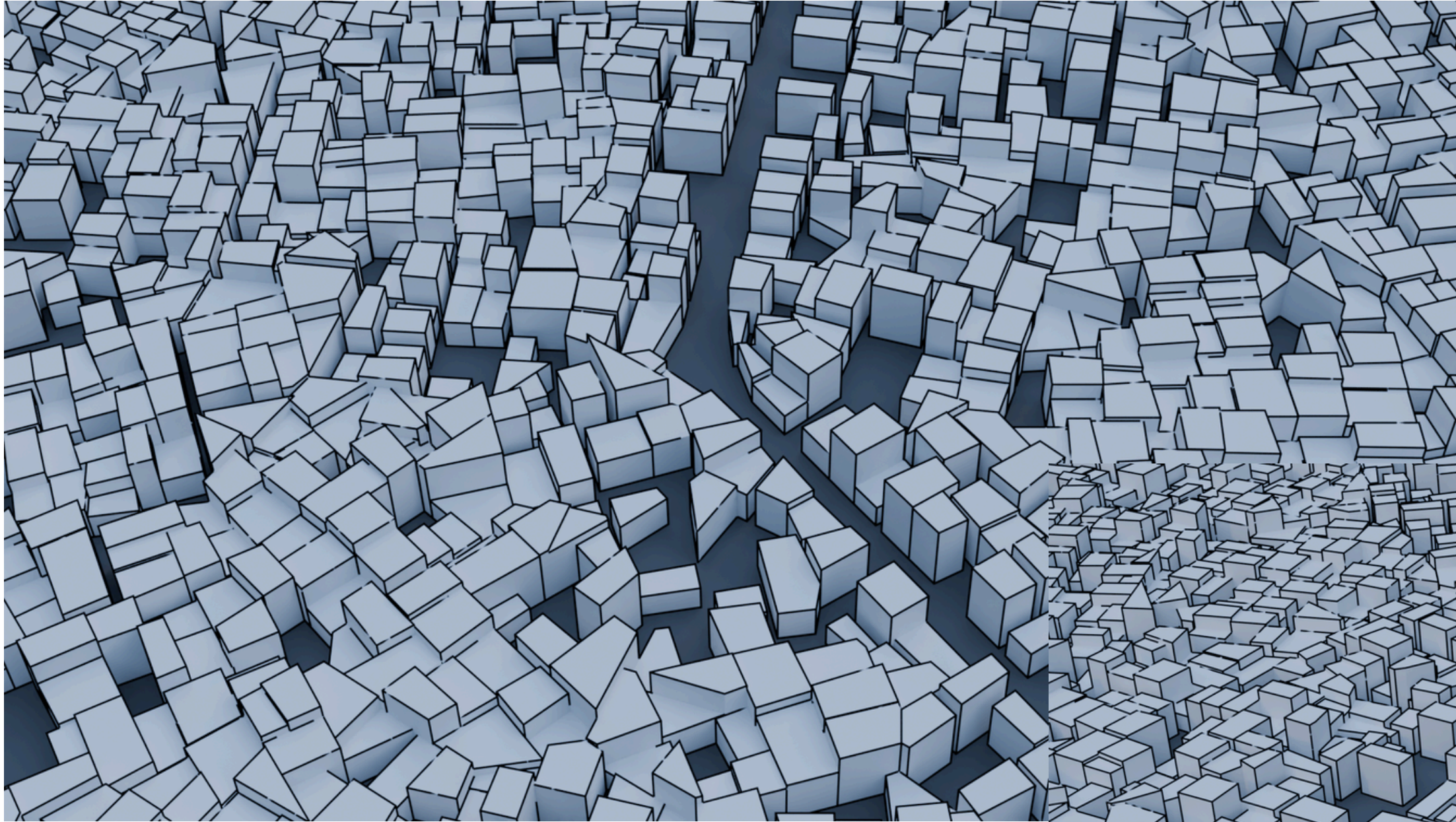
D:\projects\rust\slum_gen_v3>
```


Export and Integration

The system exports standard OBJ files, allowing for integration into industry-standard 3D modeling tools and software (CAD-inclusive).



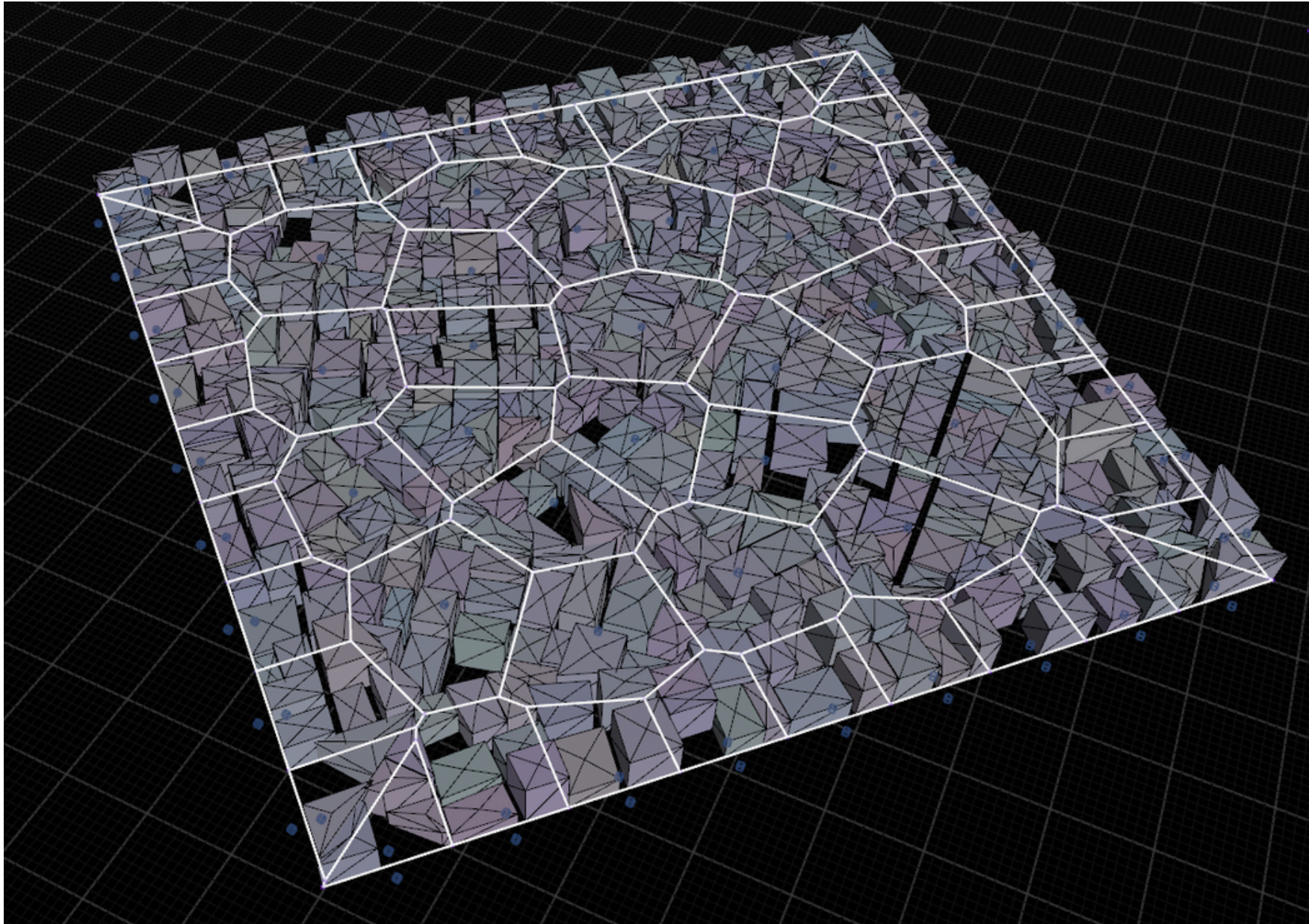
This means that models created with the software can directly be used!



3D renders of my models created
using Blender 4.1

Limitations

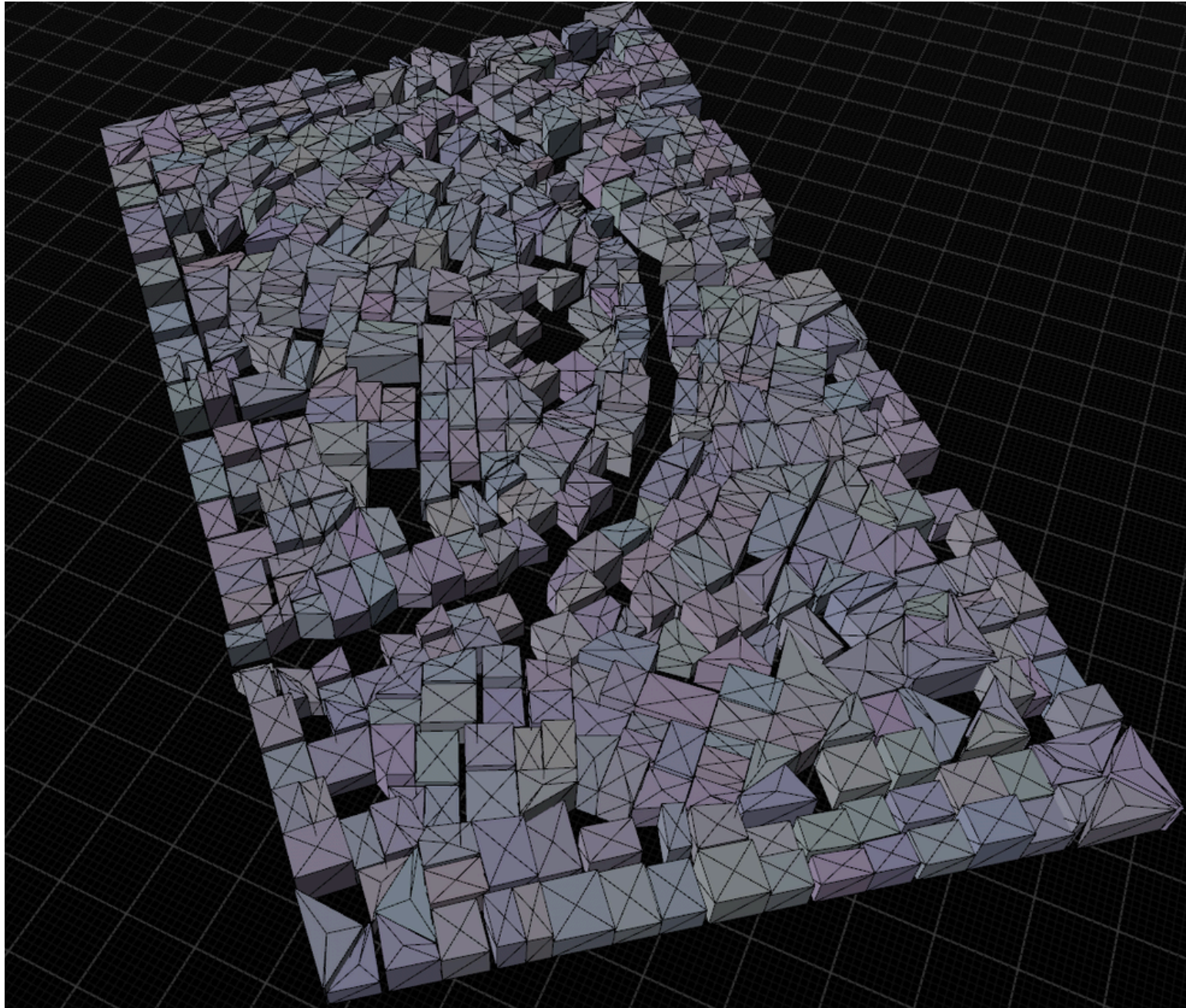
While technically complete and functional, this system in its current state serves as a foundation for more sophisticated urban modeling tools.



1. **Algorithm Refinement:** Core Voronoi/spacing logic can be optimized for more better organic patterns and smoother models
2. **Simplified 3D Representation:** Basic building extrusions lack architectural detail
3. **Alley and Pathing:** Due to how the generation pipeline is structured, alleys do not connect to each other

Future Work

Ways this software can be built upon/improved in future research or development



1. **Advanced 3D Modeling**

- a. Implement L-systems for detailed facades and roof structures
- b. Add procedural texturing for realistic materials (corrugated metal, brick, etc.)

2. **Intelligent Street Generation**

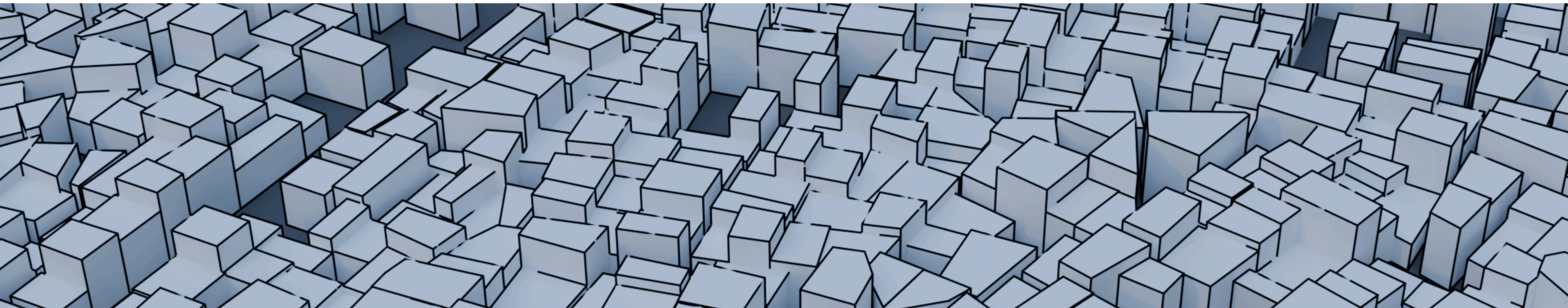
- a. Integrate L-systems or tensor fields
- b. Develop hierarchical road networks with proper pathfinding

3. **Extended Feature Set (Parameters)**

- a. socio-economic data layers
- b. environmental factors (drainage, topography)

Conclusion

- A novel procedural generator for slums/informal settlements was successfully developed
- It fills a critical gap by simulating organic, high-density morphologies ignored by existing tools (non-grid structures)
- The system is technically robust, and can integrate with professional 3D software workflows.
- This tool provides a valuable bridge for urban planning, research, and visualization; providing a solid foundation for future work



Q&A

Queries regarding the work are
welcome