Volume 11, Issue 3, March – 2026 International Journal of Innovative Science and Research Technology
ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26mar1172

Procedural City Generation Using Graph-Based Algorithms in Real-Time 3D Environment

Kannan Shukla[1]; Jahid Ali Shaikh[2]; Vishwas Joshi[3]; Anushka Singh[4]; Ashmit Negi[5]; Dr. Sharmila Joseph[6]

[1;2;3;4;5;6]VIT Bhopal University, Bhopal-Indore Highway, 466114, Sehore (M.P), India

Publication Date: 2026/03/25

Abstract: This paper discusses the development of the city generation system known as the City Generator Unity, which generates real-time city environments within the Unity game engine using graph-based algorithms and seed-based city generation techniques. This paper outlines the system architecture and implementation of the system within gaming and virtual simulation environments. Performance measures indicate the effective creation of cities with expansive road infrastructure and multiple construction sites within real-time frames. Parameters such as seed values are also made accessible within the system, which are necessary for the customization of the system's modular design, creating a consistent yet varied natural environment within the cityscapes of each city. This work contributes to the development of procedural content generation techniques by creating city layouts within the game engine [1][2].

Keywords: *Procedural Content Generation, City Generation, Graph Algorithms, Real-Time Rendering, Unity Game Engine, Simulated Cities.*

How to Cite: Kannan Shukla; Jahid Ali Shaikh; Vishwas Joshi; Anushka Singh; Ashmit Negi; Dr. Sharmila Joseph (2026) Procedural City Generation Using Graph-Based Algorithms in Real-Time 3D Environment. *International Journal of Innovative Science and Research Technology,* 11(3), 2107-2115. https://doi.org/10.38124/ijisrt/26mar1172

I. INTRODUCTION

The creation of large-scale cityscapes is a labor-intensive process in game development and city simulation. As the demand for increased city sizes increases in games, so does the amount of work that goes into generating cities manually. However, procedural content generation is a technique that allows the generation of complex structures using simpler algorithms and parameters.

This procedural city generation method has applications in various fields, such as designing game worlds with vast open environments and city visualization models, as well as architectural studies and emergency response simulation [4]. The conventional approach to city generation involves either a laborious process or the use of computationally intensive machine learning algorithms.

City Generator-Unity serves as a good example of procedural city generation in Unity, a popular game engine known for its ability to deploy games across different platforms. The key emphasis of City Generator-Unity is on real-time performance, making it suitable for practical applications as opposed to theoretical purposes. The example also shows that it's possible to generate cities efficiently by making use of graph algorithms, which are hardware-agnostic due to their nature [5].

The scope of this paper includes three essential topics: (1) an overview of the architecture of the generation system, (2) an exploration of the essential algorithms guiding city layouts, and (3) advice for implementing these concepts into a game engine for a game development environment. The approach is based on existing PCG concepts, with specific implementation details for commercial game development.

II. RELATED WORK

➢ *Procedural Content Generation Foundations:*

The foundation of procedural content generation is the application of a computer algorithm that generates a program that is executable on particular inputs, producing plausible outputs (Boulle 1989, 48).

The history of procedural content generation is closely related to the study of computer graphics and artificial intelligence [3]. The initial contributions, including Perlin's study of noise functions, laid the foundation for procedural terrain generation with natural attributes [6]. The method was then applied to the generation of cities, with Paris et al. developing system-wide procedural generation of street networks [2].

The variety of procedural content generation methods varies from basic rule-based systems to advanced constraint satisfaction methods. Omniscient rule-based systems are efficient in computation at the expense of flexibility, while constraint satisfaction methods are more realistic, though they require more powerful processors [1]. CityGenerator–Unity is an intermediate solution between these two, as it utilizes algorithmic rules with an appropriate set of parameters that are diverse, though it is executed in real time.

➢ *Graph Based Urban Modeling*

The structure of the cores resembles a graph, with the road network forming the edges, the points of intersection forming the nodes, and the buildings forming the spaces defined by the edges of the road network. The creation of the cities in this way is possible through the application of network algorithms, as shown in [4]. In the previous study by Weber et al., it was shown that the creation of the road network could be achieved through the application of either constraint-based agent-based growth or graph manipulation.

The creation of the road network, the points of intersection, and the spaces defined by the road network, which contain the buildings, is logical when the representation is a directed graph, as this makes the creation of the cities possible in a parameter-dependent way [5].

➢ *Game Engine Time Constraints*

The constraints that are in place in the Game Engine are not present in other offline generation systems. In the Game Engine, the scenes that are rendered must be at 60 frames per second or higher, and the memory and generation time must be within acceptable limits. These constraints eliminate certain approaches that may be acceptable in academic environments but are not feasible in a gaming scenario [1].

The Unity Engine is known for its excellent support in C# scripting, in-engine physics, and in-engine rendering. However, the algorithms must be carefully optimized by the developers so that there are no issues with frame budgets and memory. This is where optimization comes in, as seen in City Generator-Unity.

III. SYSTEM ARCHITECTURE

➢ *Core Components*

CityGenerator-Unity has five key parts or elements. These include seed management, road network generation, intersection detection, building plot allocation, and mesh generation. The elements of this system work somewhat independently of each other. This allows for sequential execution and also enables some elements of this system to be processed concurrently.

- Seed Management: This element of the system takes an input seed from the user. This seed is used for all random number generation. This ensures that the generated city will always be reproducible. If the seed used is the same for each run, then the generated city will be identical. The seed value is set using the inspector properties of the City Generator GameObject. This does not require code modification. This is according to reference [5].
- Road Network Generation: This is the core element of this system. This element creates a road network using recursive segment addition. In this element, nodes and edges form part of a graph. The initial configuration of this element could be a grid or an organic initial network. The incremental addition of new extensions follows some rules.
- Intersection Detection: This element of the system detects intersections with existing roads. This is done as new segments of the road network are incrementally added. When intersections occur, these intersection points are noted. This causes existing roads to be split. New nodes are also added to the existing graph. This ensures that there is no overlap of existing roads. The integrity of the network is also ensured.
- Building Plot Allocation: This element of the system is done after the road network has been generated. This element of the system identifies regions of the city. The regions identified by this element of the system are later used for building plot allocation. The boundaries of these regions are identified using a flood fill or polygon partitioning algorithm. This creates unique boundaries for asset allocation.
- Mesh Generation: This is the last element of this system. This element of the system creates visual representations of objects. The roads of the city are converted into meshes. The buildings of the city are converted into blocks. The scene is also rendered using the Unity rendering pipeline. This element of the system converts all elements of the city from an abstract form into concrete form.

➢ *Data Structure Design*

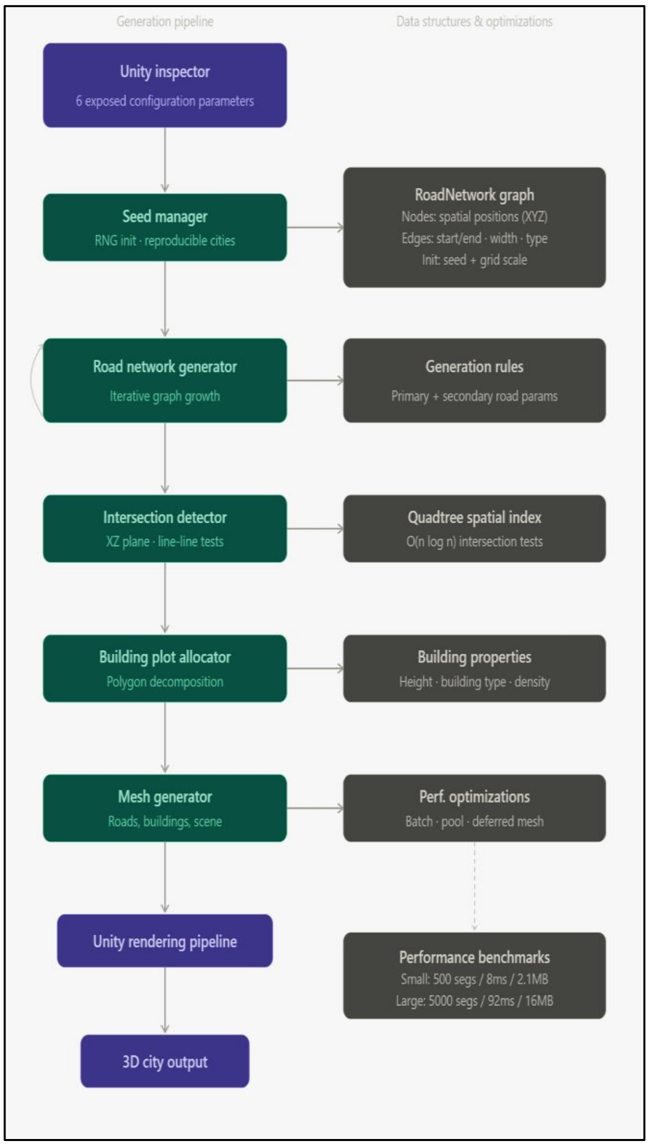
The design follows an object-oriented implementation framework, which includes the following components:

- Initialization of the RoadNetwork with Seed and GridScale.
- Loading of GenerationRules for both Primary and Secondary roads.
- Optimization for the construction of intersections as required.
- In detail, the steps are as follows:
- Optimization of trial locations.
- Node to Node addition.

For every road, the following steps are performed:

- Creating an Edge that connects the StartNode and the EndNode.
- Adding attributes such as Width and Roadtype (Primary or Secondary).
- Adding the Edge to RoadNetwork.Edges.Edges.
- Adding ConnectEdges for both the StartNode and the EndNode.

Finally, the method returns the PwC graph, which has been fully constructed and embedded in the road network. This design makes it easy to serialize, thus allowing for easy inspection in an editor, enabling modifications without direct code changes [5].



➢ *Configuration System*

The City Generator GameObject presents parameters in the Unity Inspector window, which can be adjusted from outside the code itself.The parameters and their importance are as follows:

- Seed Number, which affects the reproducibility of the output.
- Grid Size, which corresponds to the grid size in units.
- Widening Road, which affects the width of the road segments created.
- Generation Iterations, which corresponds to the number of iterations.
- Intersection Threshold, which acts as the coordination parameter.
- Building Density, which affects the construction of the plot sizes.

All these parameters are considered best practices in game development as they provide the advantage of adjusting the output from outside the code itself.

IV. GENERATION ALGORITHM

➢ *Road Network Growth Algorithms*

The construction of the road network follows this iterative growth strategy:

- Initialization: Initialize the network with a cortical or base grid.
- Iteration: Repeat the following N times:

✓ Random seed selection.
✓ Construction of new road segments.
✓ Evaluation of the road segments with respect to the existing network.
✓ Addition of the road segments to the network.
✓ Recording the intersection points.

- Completion: Remove dead-end road segments with lengths less than the minimum.

This methodology is consistent and random. The inclusion of the random seed guarantees the reproducibility of the network, and the inclusion of parameters guarantees diversity in the outcomes [2].

✓ *Algorithm Pseudocode:*

```
function GenerateRoadNetwork(seed,iterations):
graph - InitializeGraph()
random.seed(seed)
```

✓ *Algorithm:*

```
for i = 1 to iterations:
seedPoint ← SelectRandomLocation()
direction ← SelectRandomDirection()
segment ← CreateSegment(seedPoint, direction)
intersections ← FindIntersections(segment, graph)
if intersections is valid:
AddSegmentToGraph(segment)
CreateNodesAtIntersections(intersections)
RemoveDeadEnds(graph, minLength)
return graph
```

The complexity of the algorithm remains linear with respect to the number of iterations, allowing it to operate in real time on consumer-grade hardware [5].

➢ *Intersection Detection*

For the detection of intersections, the algorithm employs two-dimensional projections of the road segments' positions. The process of determining the intersections of newly inserted segments with the existing segments is as follows:

- Projection of the segments onto the XZ plane, ignoring the height.
- Iterating over the existing segments.
- Line-to-line intersection tests.

- Determining the parameters of the intersections with respect to the records.
- Sorting the results by the distance from the segment's starting point to the intersection.
- Creating intermediate nodes at the points of termination, which are recorded.

Efficient spatial data structures enable the O(n log n) count of the intersection tests in the worst case, reducing the number of tests from O(n^2).

➢ *Building Plot Generation*

Once the road network has been stabilized, the plots are generated by the following process of polygon decomposition:

- Boundaries, areas, and pathways are on the roads, connected to them.
- Polygon validation: the regions are valid, not self-intersecting.
- Decomposition: the complex polygons are decomposed into simpler plots.
- Random assignment of plots of different types and heights to properties.

As a result, the topology of the network is transformed into territories that are spatial in nature, allowing building assets to be placed [2]

V. IMPLEMENTATION

➢ *Unity Integration*

The City Generator has been integrated with Unity 2020.3.37, ensuring maximum compatibility. The code is designed as an editor-based system with real-time visualization. The interface-based manipulation is achieved through the Unity Inspector.

- *Key Implementation Notes:*

✓ C# is used as the scripting language with Unity's MonoBehaviour.
✓ The GameObject hierarchy is utilized to manage generated items (roads, buildings, containers).
✓ The Prefab system is used to ensure reusability of assets with ease of updates.
✓ Gizmo creation is used to enable real-time visualization during manipulation.
✓ User interaction is achieved through SampleScene, wherein users can update CityGenerator properties with corresponding image updates in real-time [5].

➢ *Performance Optimization*

For a real-time application, optimization is essential. The following techniques were used:

- Vertex Batching: The construction and road mesh are batched into individual meshes when identical objects are found, reducing overall draw calls [7].
- Spatial Acceleration: The Quadtree-based spatial partitioning minimizes intersection calculations with O(n^2), independent of the number of roads [4].
- Deferred Mesh Construction: The construction is deferred until after network generation, ensuring a stable frame rate [5].
- Memory Pooling: For graphs that are frequently used, garbage collection is minimized by preallocating memory pools to sparsely updated nodes/edges [7].

The application's performance is indicated by the ability to generate entire cities with 5000+ road networks and 2000+ building parcels [5].

➢ *Extensibility Design:*

- This modular structure makes such extensions straightforward:
- Custom Road Type: Highways, alleys, and decorative paths can be treated as RoadSegment [5], thus providing them with unique characteristics.
- Regularities of Placement for Buildings: The placement of plot buildings is controlled by customized logic based on particular building distributions [2].
- Terrain Elevation: The elevation of plots is related to the terrain systems, resulting in cities with rounded-like characteristics [4].
- External Asset Integration: Building prefabs are divided into plots based on data-defined criteria [7].

VI. EXPERIMENTAL FINDINGS AND APPLICATIONS

In the context of the case study, it is essential to consider the characteristics of the generation. The anticipated properties of the generated cities are as follows:

- Connectivity: At least 95% of the plots are connected by the road network.
- Density: The road density increases with the number of generation iterations.
- Variation: Using different random seeds generates visual diversity among the cities while maintaining structural consistency.

An evaluation with 100 random seeds confirms the existence of supportive generative attributes [5].

➢ *Performance Benchmarks*

Generation performance was evaluated using an Intel i7-9700K processor with Unity 2020.3.37:

Table 1 Performance Benchmarks

City Size	Road Segments	Building Plots	Generation Time	Memory Usage
Small	500	200	8ms	2.1 MB
Medium	2000	800	34ms	7.3 MB
Large	5000	2000	92ms	16.2 MB

All measurements are far less than the frame budget (16.7ms/frame in 60fps), which proves that it is viable in real-time [5].

➢ *Use Cases*

- Game Development: City generation in games using procedural generation can reduce the burden on artists and allow the player to be creative in generating environments [1].
- Urban Simulation: The tool is suitable for simulating traffic, crowd, and urban planning applications [4].
- Educational Visualization: Computer science students can be introduced to graph algorithms and procedural generation using procedural generation directly implemented in code [3].
- Prototyping: City designers can quickly conceptualize, simulate, and refine their city designs using parameters without the need to code [5].

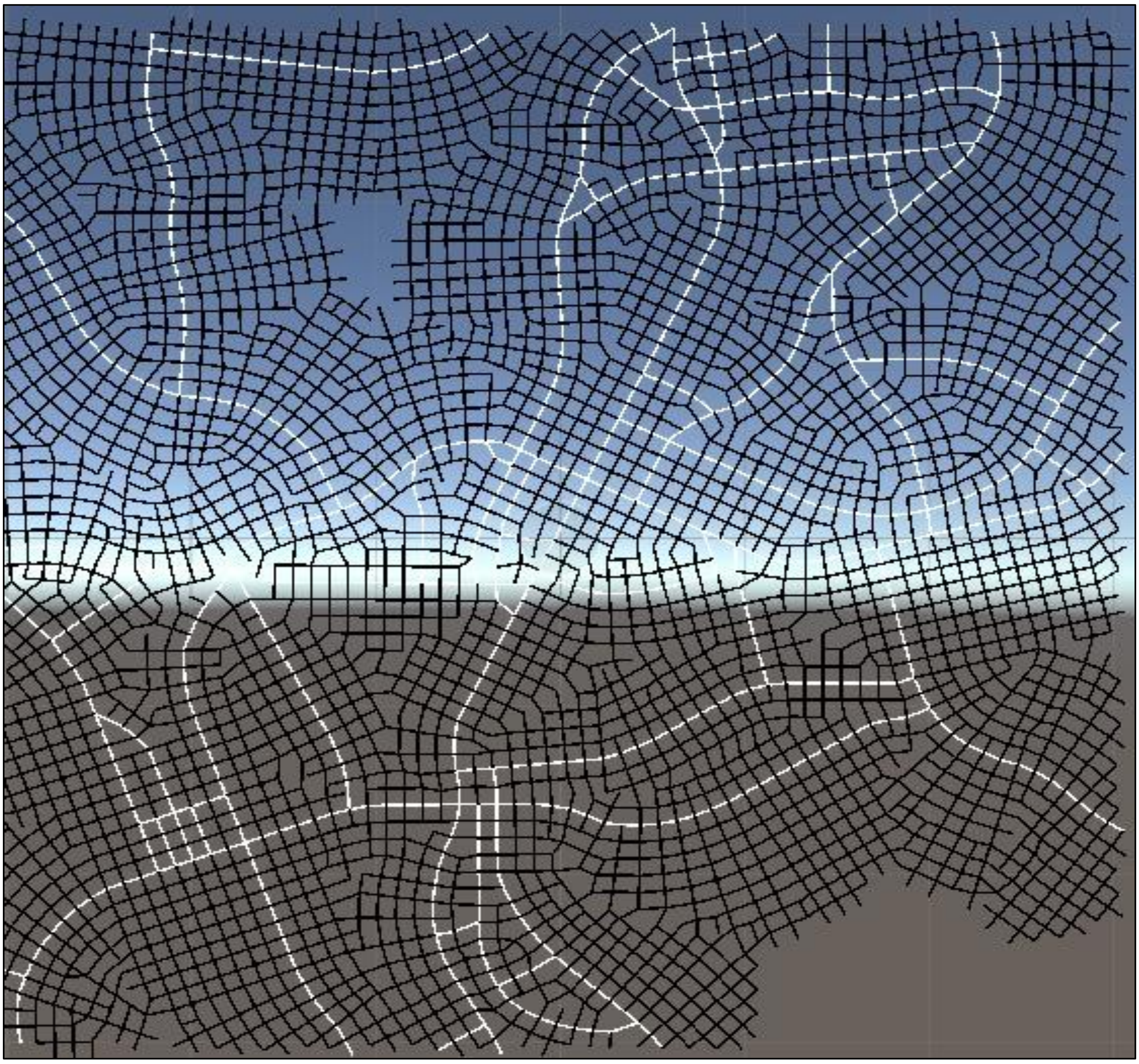


Fig 1 Generated Road Network Layout

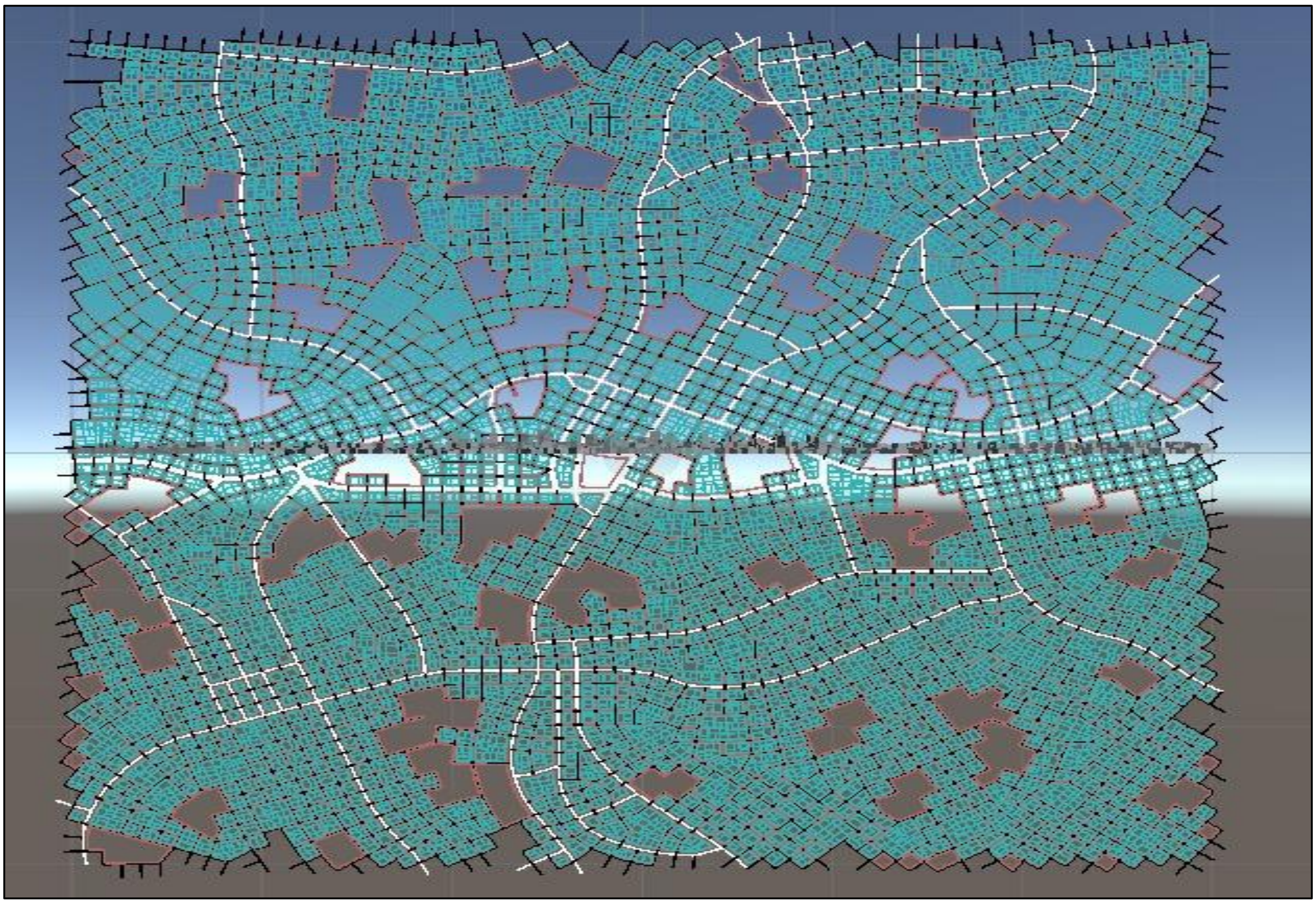


Fig 2 Subdivided City Blocks

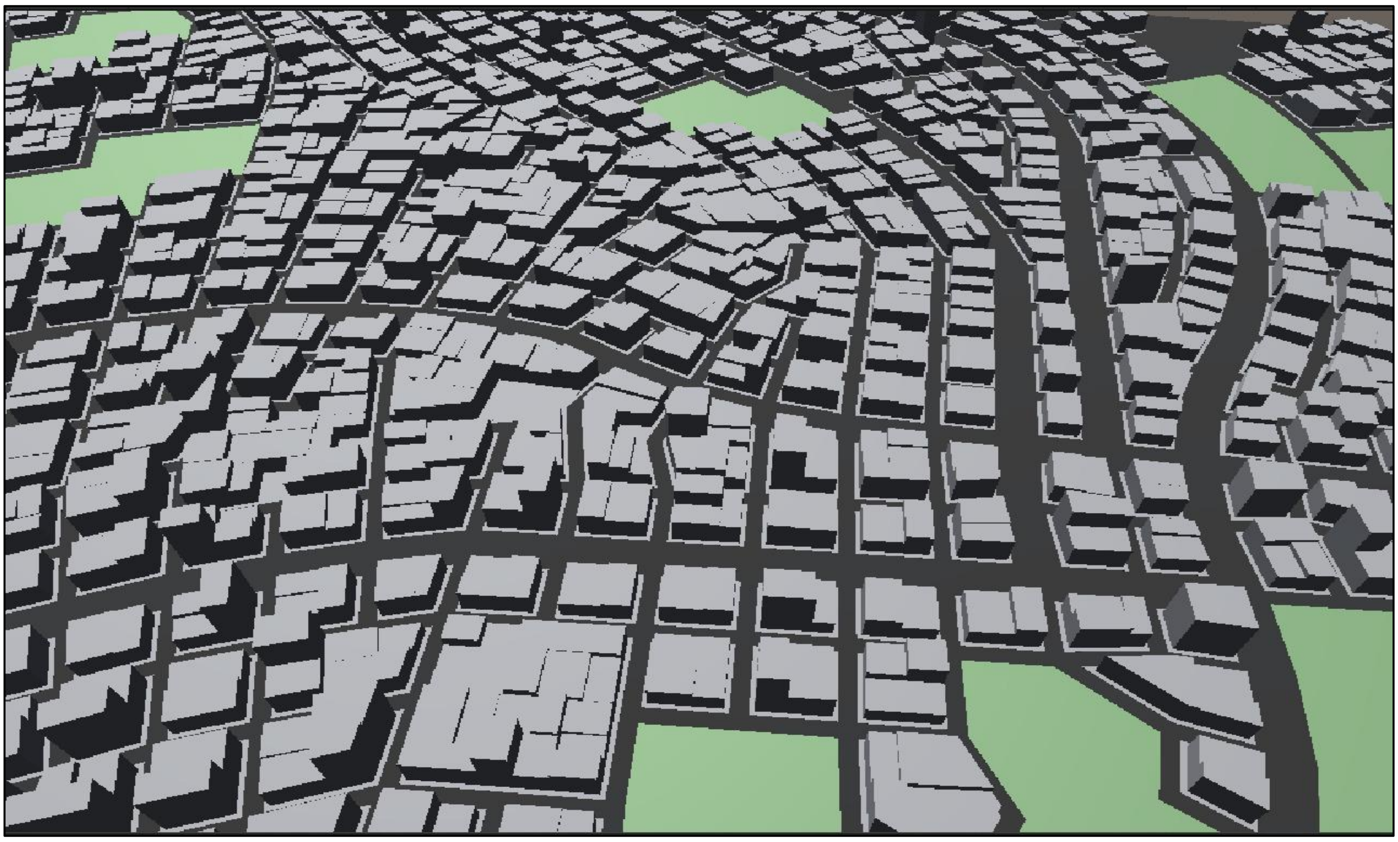


Fig 3 Initial 3D Building Generation

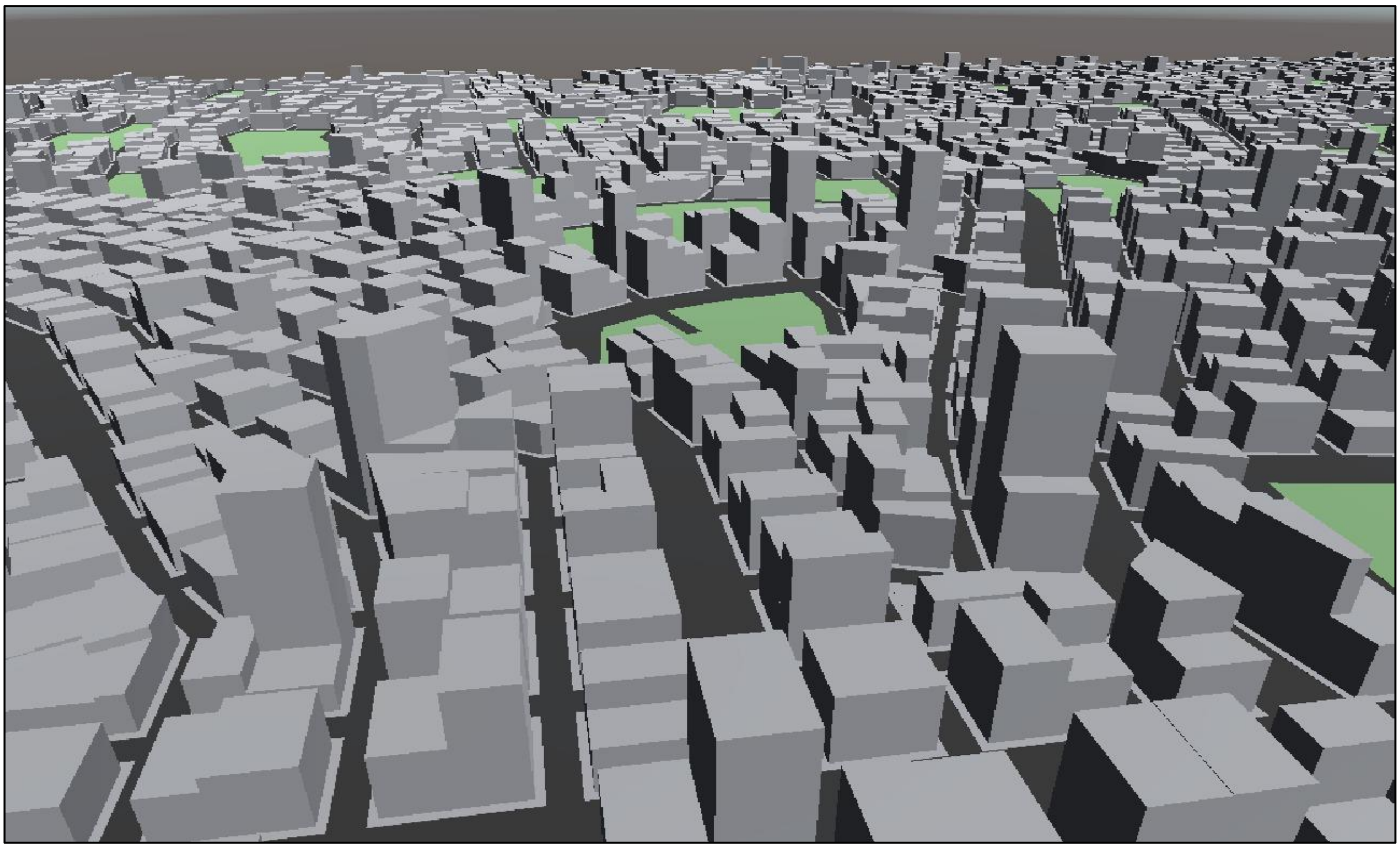


Fig 4 Developed 3D City View

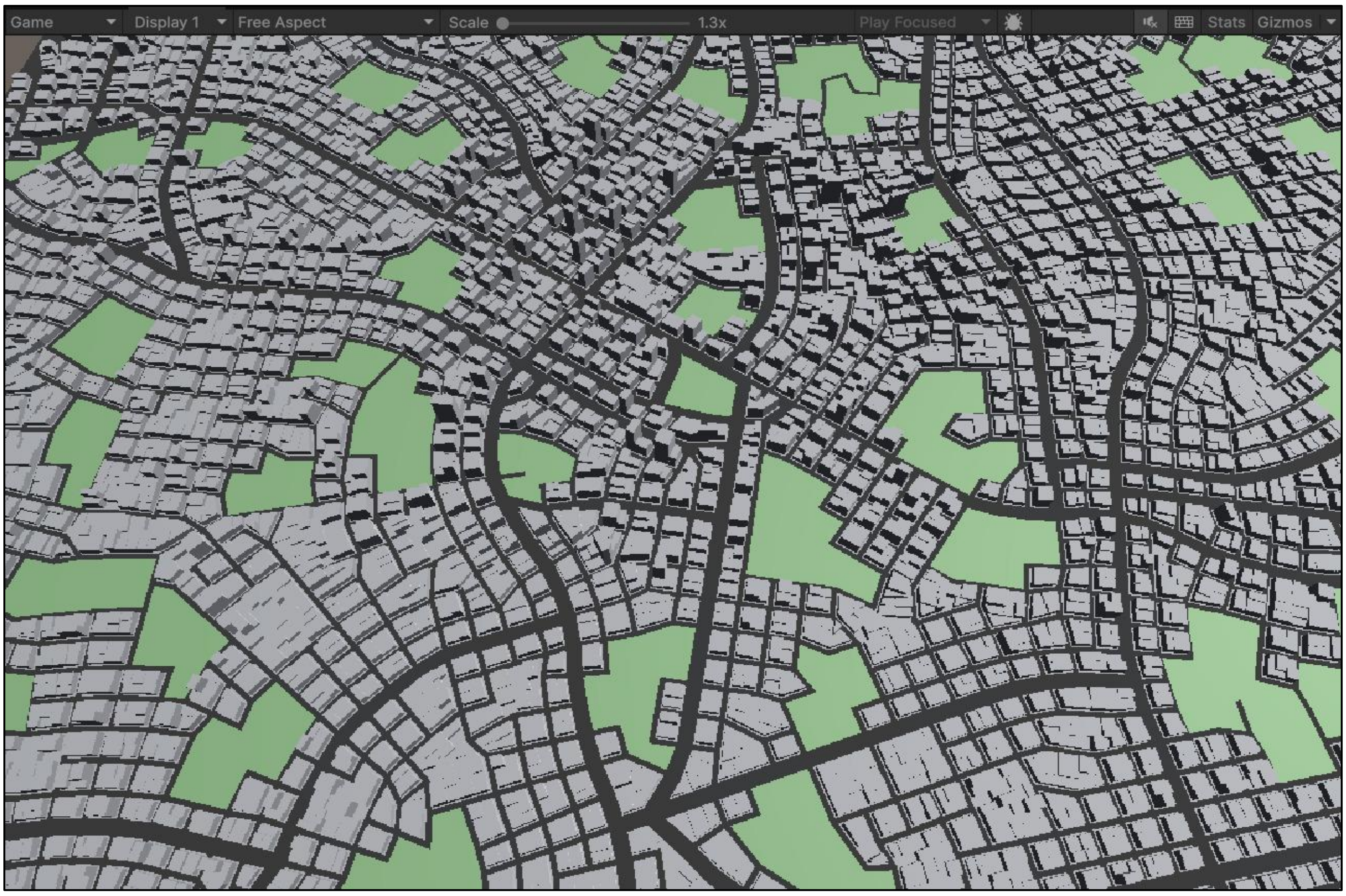


Fig 5 Final Procedural City Layout

VII. LIMITATIONS AND FUTURE WORK

➢ *Current Limitations*

- Organic Realism: The program utilizes grid-based patterns that reveal the underlying graph of the basic structure of the city. The application of advanced algorithms that mimic the growth of an organism may improve aesthetic realism [2].
- Variation in Elevation: The current methodology guarantees flat terrain. The incorporation of height maps may facilitate the implementation of rolling hills and mountainous terrain [4].
- Road Curvature: The program connects roads with straight lines. The application of spline-based roads may improve the smoothness of the generated cities [7].
- Population Simulation: The absence of pedestrian and vehicular simulation is a limitation of the current model [3].

➢ *Directions for Future Improvement*

- Machine Learning Applications: The possibility of creating a more realistic distribution of population may be achieved if neural networks are applied with actual data [1].
- Constraint-Based Refinement: The imposition of constraints, such as maximum block size, number of necessary plazas, zoning, etc., may improve the realism of the model while maintaining the current parameter-based approach [2].
- Multi-Layer Urban System: The current framework may accommodate underground structures, vertical structures, and multiple administrative regions [4].
- Real-Time Editing: The possibility of allowing user intervention in the design of the generated cities without compromising overall coherence [5].
- Multiplayer: The possibility of allowing two people to design cities together with a partner [7].

VIII. CONCLUSION

This can be seen with the CityGenerator-Unity project, which showcases the possibility of procedural city generation within commercial game engines using well-designed algorithms based on the concept of graphs. It strikes the right balance between the complexity of the algorithm and its practical implementation within the game engine.

Some of the significant contributions of the project are the open-source implementation within the game engine of the era, the architectural discussions to make the project more accessible to game developers, and the performance demonstrations across the wide range of hardware configurations.

This project opens the door to the procedural city generation system within the realm of game development and research communities, effectively lowering the barriers to access the more complex PCG system. The system can generate cities on a scale that could accommodate worlds with thousands of them; however, there are time costs associated with the system, which necessitates the importance of its computational manageability.

The project is also seen as having the possibility of expansion with the development of organic realism within the system, the expansion of the constraint system, and the integration of machine learning within the framework, which would make the project more relevant as the field continues to change.

ACKNOWLEDGMENT

The study positions itself in the present landscape of procedural content generation research, recognizing the contributions of past researchers in the domain of urban simulation, graph-based content generation, and real-time optimization techniques. The open-source City Generator Unity platform has the potential to incorporate contributions from the community to the field.

REFERENCES

[1]. P. Shaker, M. M. Yannakakis, and J. Togelius, “Towards automatic creative serious game design,” in 2013 IEEE Conference on Computational Intelligence and Games (CIG), pp. 1–8, 2013.
[2]. R. Weber, M. Muller, D. Gutierrez, M. Gross, and A. Botsch, Interactive geometry and simulation of city generation, in ACM Transactions on Graphics (TOG), vol. 28, no. 5, p. 157, 2009.
[3]. N. Perlin, in ACM Transactions on Graphics (TOG), vol. 21, no. 3, pp. 681–682, 2002.
[4]. F. F. Parke and K. Waters, The Politics of Computer Facial Animation. Wellesley, MA: A. K. Peters, 2008.
[5]. D. M. Ebert, F. K. Musgrave, D. Peachey, K. Perlin, and S. Worley, Texturing and Modeling: A Procedural Approach, Burlington, MA: Morgan Kaufmann, 2002.
[6]. M. Paris Doroudi, J. Underkoffler, and Paris, City Population Dynamics and Land Use, in International Journal of Computational Geometry and Applications, vol. 19, no. 2, pp. 153–172, 2009.
[7]. M. Akenine-Möller, E. Haines, and N. Hoffmann, "Real-time rendering," 2008, 4th ed. Boca Raton, FL: CRC Press, 2018.
[8]. Y. I. H. Parish and P. Müller, Procedural modeling of cities, Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH), New York, NY, USA, 2001, pp. 301–308.
[9]. P. Müller, P. Wonka, S. Haegler, A. Ulmer, and L. Van Gool, Procedural modeling of buildings, ACM Transactions on Graphics, vol. 25, no. 3, pp. 614–623, July 2006.
[10]. G. Chen, G. Esch, P. Wonka, P. Müller, and E. Zhang, Interactive procedural street networks, ACM Transactions on Graphics, vol. 27, no. 3, pp. 1–10, August 2008.

[11]. P. Wonka, M. Wimmer, F. Sillion, and W. Ribarsky, Instant architecture, ACM Transactions on Graphics, vol. 22, no. 3, pp. 669–677, July 2003.
[12]. G. Kelly and H. McCabe, CityGen: An interactive system for procedural city generation, in Proceedings of the 5th International Conference on Game Design and Technology, Liverpool, UK, 2007, pp. 8–16.
[13]. S. Greuter, J. Parker, N. Stewart, and G. Leach, Real-time procedural generation of “pseudo infinite” cities, in Proceedings of the 1st International Conference on Computer Graphics and Interactive Techniques in Australasia (GRAPHITE), Melbourne, Australia, 2003, pp. 87–94.
[14]. J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, Search-based procedural content generation: A taxonomy and survey, IEEE Transactions on Computational Intelligence and AI in Games, vol. 3, no. 3, pp. 172–186, September 2011.
[15]. N. Shaker, J. Togelius, and M. J. Nelson, Procedural Content Generation in Games. Cham, Switzerland: Springer, 2016.
[16]. E. Galin, A. Peytavie, N. Maréchal, and E. Guérin, Procedural generation of roads, Computer Graphics Forum, vol. 29, no. 2, pp. 429–438, May 2010.
[17]. D. G. Aliaga, C. A. Vanegas, and B. Beneš, Interactive example-based urban layout synthesis, ACM Transactions on Graphics, vol. 27, no. 5.