MINISTRY OF EDUCATION AND SCIENTIFIC RESEARCH

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA

FACULTY OF AUTOMATION AND COMPUTER SCIENCE
COMPUTER SCIENCE DEPARTMENT

GUIDED PROCEDURAL MODELING OF URBAN LANDSCAPES

LICENSE THESIS

Graduate: Diana TOFAN
Supervisor: As. dr. eng. Adrian SABOU

2017

MINISTRY OF EDUCATION AND SCIENTIFIC RESEARCH

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA

FACULTY OF AUTOMATION AND COMPUTER SCIENCE
COMPUTER SCIENCE DEPARTMENT

DEAN,
Prof. dr. eng. Liviu MICLEA

HEAD OF DEPARTMENT,
Prof. dr. eng. Rodica POTOLEA

Graduate: **Diana TOFAN**

GUIDED PROCEDURAL MODELING OF URBAN LANDSCAPES

1. **Project proposal:** *Develop an application that allows users to generate urban landscapes based on building heightmaps*
2. **Project contents:** *Introduction, Project Objectives, Bibliographic Research, Analysis and Theoretical Foundation, Detailed Design and Implementation, Testing and Validation, User's Manual, Conclusions, Bibliography*
3. **Place of documentation:** *Technical University of Cluj-Napoca, Computer Science Department*
4. **Consultants:**
5. **Date of issue of the proposal:** November 1, 2016
6. **Date of delivery:** July 14, 2017

Graduate: _____

Supervisor: _____

MINISTRY OF EDUCATION AND SCIENTIFIC RESEARCH

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA

FACULTY OF AUTOMATION AND COMPUTER SCIENCE
COMPUTER SCIENCE DEPARTMENT

Declaraţie pe proprie răspundere privind autenticitatea lucrării de licenţă

Subsemnata **Tofan Diana**, legitimată cu **C.I.** seria **NT** nr. **625588** CNP **2940210 270023**, autorul lucrării **Guided Procedural Modeling of Urban Landscapes** elaborată în vederea susţinerii examenului de finalizare a studiilor de licenţă la Facultatea de Automatică şi Calculatoare, Specializarea **Calculatoare Engleză** din cadrul Universităţii Tehnice din Cluj-Napoca, sesiunea **Iulie** a anului universitar **2016 - 2017**, declar pe proprie răspundere, că această lucrare este rezultatul propriei activităţi intelectuale, pe baza cercetărilor mele şi pe baza informaţiilor obţinute din surse care au fost citate, în textul lucrării şi în bibliografie.

Declar, că această lucrare nu conţine porţiuni plagiate, iar sursele bibliografice au fost folosite cu respectarea legislaţiei române şi a convenţiilor internaţionale privind drepturile de autor.

Declar, de asemenea, că această lucrare nu a mai fost prezentată în faţa unei alte comisii de examen de licenţă.

În cazul constatării ulterioare a unor declaraţii false, voi suporta sancţiunile administrative, respectiv, *anularea examenului de licenţă*.

Data
14 iulie 2017

Nume, Prenume
Tofan Diana

Semnătura

Contents

Chapter 1

Introduction - Project Context

Creating high-quality three-dimensional (3D) content for the media industry, including video games, movies and virtual environments is one of the most meaningful goals that Computer Generated Imagery (CGI) is trying to achieve. Along with the evolution of technology, the consumers' expectations are in a continuous growth, being essential for the software vendors to deliver realistic and detailed products to ensure the users' satisfaction.

Approaching the problem of creating digital content in the traditional way, by manually designing each 3D model from the scene, would lead to serious consequences. On one hand, hiring a high number of artists is extremely expensive and would not necessarily guarantee faster results. On the other hand, the time spent on performing such a complex and meticulous task could take years of hard work and commitment. Therefore, embarking on such a risky journey is not worth the time, money or effort, as long as more convenient alternatives do exist.

An effective solution to the problem stated above can be offered by the procedural generation algorithms. These techniques create unique 3D content at run-time based on nothing but code, and require no prior knowledge in the field - anyone can easily generate complex models only at the click of a button. In contrast to the traditional approach, several advantages can be observed. First of all, there is no need for new employees in the manufacturing companies since computers play the leading role in the design process. This results in extremely lower costs which is a huge benefit CGI provides. Secondly, the time required to generate largely-scaled models is incomparable to that of manual modeling. The desired models are ready for use only in a matter of minutes and can also be modified at request until they exceed the customer expectations.

Urban landscapes are useful for a considerable number of applications. Almost every movie or video game requires at some point a largely-scaled city scene that the characters can explore. Urban planning, that is concerned with the development and use of land, also relies on 3D visualizations in order to find the optimal solutions. The virtual reconstruction of cities is another topic of significant interest, being widely used in the entertainment industry to generate convincing cityscapes that deeply contribute to the storyline. Furthermore, urban landscapes are also useful for flight simulators, i.e.

machines that use CGI to mimic the view and experience of piloting an aircraft. Procedural modeling is powerful enough to evenly distribute hundreds of buildings on empty terrains, giving the pilots the illusion that they are flying above large cities.

Even though virtual cities gained so much popularity in the past few years, there are some features that the applications available nowadays are lacking. Little attention has been paid to the simplicity of use. As a consequence, achieving satisfactory results without having any prior knowledge in 3D modeling and programming is a difficult task. The average user would spend hours - in the best case scenario - trying to figure out how the application works, and this would be unfortunate since it is not the user's concern to understand what hides beneath the system. The sole purpose of the average user should consist only in deciding how the final outcome will look like in natural language. The workflow as a whole should be as simple as possible, without imposing any threats. This is the main concern of the currently proposed solution, which will try to find a powerful method by which one can generate large cities in the shortest amount of time. Moreover, special emphasis will be placed on the user input, so that the final results accurately reflect one's wishes.

Chapter 2

Project Objectives and Specifications

The main target of this project consists in implementing an interactive system capable of capturing the user's wishes and accurately reflect them in the generated landscapes. Thus, this paper aims to present an accessible approach by which one can fully control the geometrical aspects of a 3D city before creating it. However, in order to achieve this final goal, a subsequent step has to be performed, that being the procedural modeling of buildings. Lastly, to ensure a smooth experience, both these functionalities will be integrated in a standalone executable program. Therefore, the system is built upon three major objectives.

2.1 Guided Procedural Modeling of Buildings

The foremost objective of the paper is to generate user-guided buildings. Before creating each 3D model, the application first gathers all the data it needs and returns a result based on the given specifications. The users can express their wishes with respect to each building, being able to choose from a wide range of options. Some examples of available options include the building height and width, the number of floors of each building and the number of windows on each floor. Each building can be customized to the users' taste, as they can alter the geometry until they obtain a satisfactory result. A strong emphasis is put on offering the users full control over the system to let them accomplish their tasks individually, without being slowed down by a system that is too difficult to learn and requires prior training in order to use it properly.

2.2 Guided Procedural Modeling of Urban Landscapes

The major goal of the project consists in generating largely-scaled 3D environments, starting from the buildings implemented in the previous step. The process of modeling an urban landscape consists of several steps that need to be followed before achieving the desired result. Before the generation starts, a 2D input map provided by the user that

acts as a building footprint is required. Each map should respect a set of rules to ensure a proper behavior of the system. As a basic idea, the image to be uploaded must be grayscale and should consist of rectangular areas of different shades. In the lighter-shaded regions from the provided map, taller buildings will be placed, whereas in the darker-shaded zones the system will put the shorter buildings.

As mentioned before, great attention is paid to the degree of user control that the application allows. Therefore, the users are equipped with the freedom to choose before generating the 3D environments, adjusting their needs as they wish. Using an input image as an aerial representation of the virtual city is advantageous due to the simplicity and precision of such maps. On top of that, drawing basic rectangular shapes requires no artistic skills and is a task that can be successfully completed by anyone. Other useful attribute that can be decided by the users is the building density - a higher density means more buildings and otherwise. After obtaining an adequate result, the users can export the 3D models and make use of them.

2.3 Standalone Application

Building a standalone application that implements the Guided Procedural Modeling of Urban Landscapes (GPMUL) algorithm is essential, since it should be available for public use and not run locally on a single computer. Furthermore, the end-users are capable of testing the application at all times this way, without being limited by any temporal constraints. The application is also compatible with the most commonly-used platforms and requires no internet connection in order to benefit from the services it supplies.

Chapter 3

Bibliographic Research

Procedural modeling became a topic of great interest in the past few decades and a large amount of work has been done in the area. Not until recently, however, were man-made structures like buildings and cities recreated programatically.

This chapter will begin with an essential introduction on procedural generation and provide a short overview of the most well-known procedural techniques that are available. Next, it will discuss the related work that has been done up to this point, which will be divided in two parts: first part will describe the existing algorithms successfully used for creating architectural models, while the second part will focus on the generation of urban environments.

3.1 Procedural Generation

Procedural modeling is a technique widely used in computer graphics, as it represents a fast and effective approach by which one can easily create large amounts of 3D content automatically.

Applying specialized algorithms in order to generate realistic models brings several benefits compared to the traditional way of manually handcrafting a scene, since it can provide realistic and highly-detailed results within minutes, that would otherwise take years of meticulous work.

3.1.1 Fractals

Fractals were discovered by the mathematician Benoit Mandelbrot in 1975 and gained their name from the Latin frāctus meaning "broken" or "fractured". They were primarily designed to represent natural shapes, which could not be described by conventional geometry due to their irregular forms. Other applications include astrophysics (stars), biological sciences (chromosomes, DNA sequences) and computer graphics (textures).

The key feature of fractals consists in self-similarity. In simpler terms, this refers to the fact that no matter how much would someone zoom in, the same pattern reappears

over and over again irregardless of the scale. Fractals are based on recursive algorithms and successive recursions lead to higher levels of detail.

Figure 3.1 illustrates one of the most basic examples of fractals, called the Sierpinski triangle, where the overall shape of a triangle is subdivided recursively into smaller equilateral triangles. There is no theoretical limit concerning the number of iterations. Hence, an infinite pattern of smaller triangles can be obtained.

Fractals can be successfully used to simulate complex models in an iterative manner, starting from a set of simple equations. However, they are subject to limitations when the main intent is to create non-repetitive models with a lower degree of self-similarity. From this point of view, formal grammars or L-Systems may be considered superior choices.

Figure 3.1: The first five iterations of the Sierpinski triangle [1]

3.1.2 L-Systems

Lindenmayer systems, or L-Systems for short, are named after their founder, the biologist Aristid Lindenmayer. He introduced the concept in 1968 to visualize the growth process of plants.

L-Systems represent string rewriting systems consisting of an axiom and production rules. The axiom is the starting point from which construction begins and the production rules are composed of two parts: a predecessor on the left-hand side, where the symbol to be replaced resides and a successor on the right-hand side, where the string that will replace the predecessor is defined. To give an example, the rule `a -> b a b` transforms the string `a b c` into `b a b b c`. Starting from the initial state, the rules are applied iteratively, making it possible to recreate complex shapes, as it can be observed in Figure 3.2 that illustrates a tree obtained by applying a set of production rules.

Because a rewriting system based on letters from the alphabet could not provide the best results on plants, a graphical interpretation of L-Systems had to be introduced. That is how turtle geometry was conceived, a technique developed for modeling plant topologies of higher complexity. The turtle is actually an on-screen cursor that can draw lines given some movement commands (e.g. move right by a specified distance, turn left or right by a specified angle). Turtle graphics can be applied to L-Systems by defining a triplet (x, y, alpha), where (x, y) are the Cartesian coordinates representing the turtle's position and

[1] Source: https://upload.wikimedia.org/wikipedia/commons/thumb/0/05/Sierpinski_triangle_evolution.svg/2000px-Sierpinski_triangle_evolution.svg.png

angle alpha represents the direction in which the turtle is facing. Given a step size d and the angle increment alpha, the turtle can respond to movement commands. [1]

Although L-Systems are a powerful tool capable of generating rich landscapes with sophisticated plants of different species, they cannot be applied in other contexts, being suitable only for modeling vegetation and elements occurring in nature [2]. Nevertheless, they offer a solid starting point when one wants to generate different shapes from real world described by regular geometry.

Figure 3.2: Trees with L-Systems [2]

3.1.3 Shape grammars

Another essential procedural technique that can be extended further to cover the case of buildings and cities is represented by shape grammars.

Shape grammars were first described in [3] and consist in a set of recursively applied rules that rely on the same principles as L-Systems, the major difference between them being that the shape grammars use 2D shapes instead of string representations within each rule. As a result, they are able to generate designs that can be used in arts and architecture. Figure 3.3 is a basic example of how a complex shape can be created by using this grammar.

Shape grammars represent one of the fundamental aspects on which the current paper is based, extending their capabilities such that they can be applied for modeling 3D buildings.

[2] Source: http://www.digitalpoiesis.org/GDesign02c.html

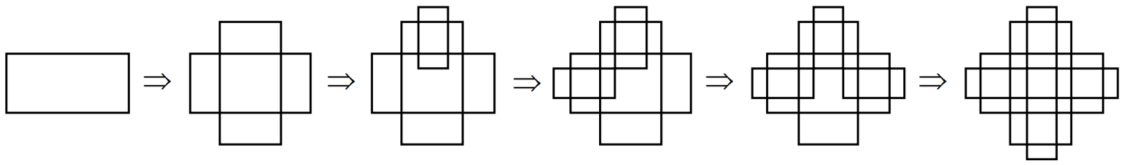

Figure 3.3: Shape grammar [3]

3.2 Procedural Generation of Buildings

In terms of buildings, procedural techniques proved to be very successful over time, as they offered a quick way by which one could create highly-detailed models made up from billions of polygons only in a matter of minutes. As a result, this particular topic became extremely popular, being the primary focus of many researchers who came up with innovative solutions that drastically improved the overall aspect of virtual buildings.

3.2.1 CGA Grammars

Although L-Systems were primarily designed to model the growth of biological systems, their capabilities have been extended enormously over time, turning them into a viable alternative for constructing 3D buildings. The earliest studies addressing this particular matter were introduced in [4], where CGA (Computer Generated Architecture) grammars first came into shape. As L-Systems provided accurate results only in the context of plants, they had to be modified so that they would also comprise the case of regular geometry.

CGA grammars represent a unique kind of shape grammars based on production rules that describe the building geometry. They are applied in an iterative manner until they generate the final result. Each rule from the grammar has a similar format:

```
id: predecessor : condition -> successor : probability
```

where *id* is a unique identifier for each rule, *predecessor* specifies which shape will be replaced with the *successor*, *condition* is a Boolean expression that needs to be fulfilled before the rule can be applied and *probability* is a value between 0 and 1 that measures the extent to which the rule will occur.

Additionally, there are multiple operations that can be specified within the body of each rule including: translate, rotate, scale, split along one axis, tile a specific element (repeat) and decompose a shape into smaller components.

An obvious advantage to the proposed method in [4] is represented by the user input itself - people who use the application can express their need better this way and

[3] Source: http://proceduralgeneration.blogspot.ro/2014/10/procedural-modelling-techniques-in.html/

their wishes will be granted. Furthermore, the degree of accuracy of the generated content is worth mentioning as well. However, defining rules that accurately describe the desired output is not a task that everyone can achieve, as the grammar language in which they are written may seem unfamiliar to the inexperienced user with no background on programming. This is one of the issues that the currently proposed system will try to solve.

3.2.2 Pseudo-Random Number Generators

As the name states, pseudo-random-number generators (PRNG henceforward) are not completely random - although they can produce long sequences of random results, these results are actually dependent on an initial seed value. If the seed value is known, the entire sequence of numbers can be reproduced.

PRNGs played an important role in [5], where they were used for the generation of buildings. Each building is built in reverse order, from top to bottom. The algorithm starts with a 2D floor plan consisting of randomly selected and merged shapes and then these floor plans are vertically extruded until they reach a randomly-generated height. The top and bottom of the final building have the same shape of the corresponding floor plan, as it can be seen in Figure 3.4.

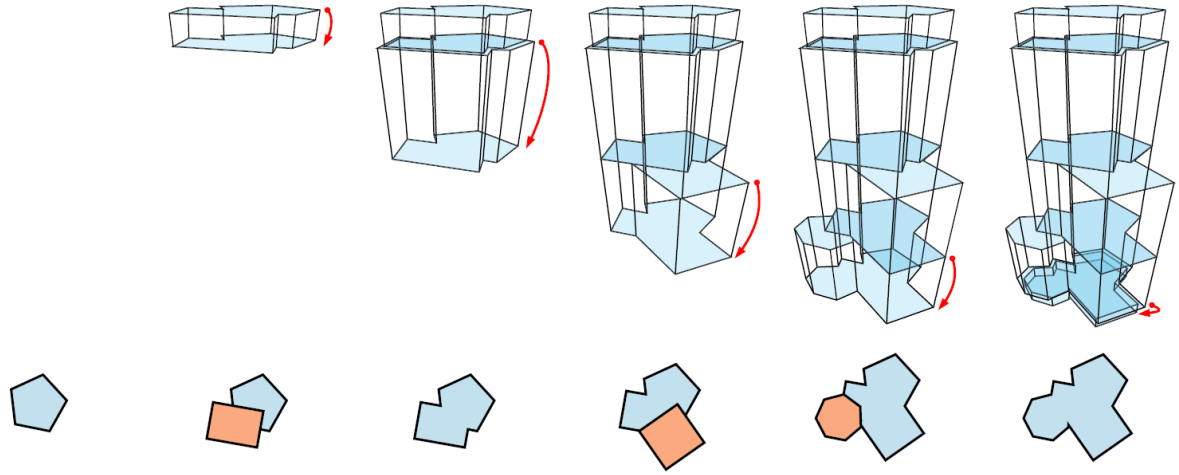

Figure 3.4: A building created by using PRNGs [5]

The approach described in [5] is powerful enough to generate thousands of unique buildings using only an initial seed value as an input, but lacks an important feature. Since the users do not have full control over the output, it is highly unlikely to obtain a specific type of building, especially in a short amount of time, as the algorithm presented in this paper massively relies on randomness. That is another key aspect on which the current paper will focus, trying to prioritize the users' needs by letting them decide exactly what they are looking for through input parameters.

3.2.3 Façade Images

Another major contribution to the procedural generation of buildings was brought in [6], where a comprehensive algorithm designed for urban reconstruction is illustrated. Given a single image of a building façade, the system is capable of reproducing a 3D geometric model with an indistinguishable resemblance to the input image.

The proposed solution consists of four stages: façade structure detection, tile refinement, element recognition and shape grammar rule extraction. In the first step, the input façade is split into separate floors and tiles, as illustrated in the first two façades from Figure 3.5. This can be achieved by detecting similar image regions using Mutual Information (MI) and then creating a data structure called Irreducible Façade (IF) that contains information about floors and tiles symmetries. Next, the IF is thoroughly analyzed by the system for deciding upon the most optimal subdivision. In the second step of the process, the tiles that were detected in the previous step are further subdivided until the doors and windows from the input image are found (Figure 3.5, third façade). Moving further, the third stage matches the architectural elements with 3D objects from a library, for ensuring high-quality results and providing semantic information. In the last step of the algorithm, the computed subdivisions are encoded as CGA shape grammar rules inspired by those defined in [4] (Figure 3.5, last façade).

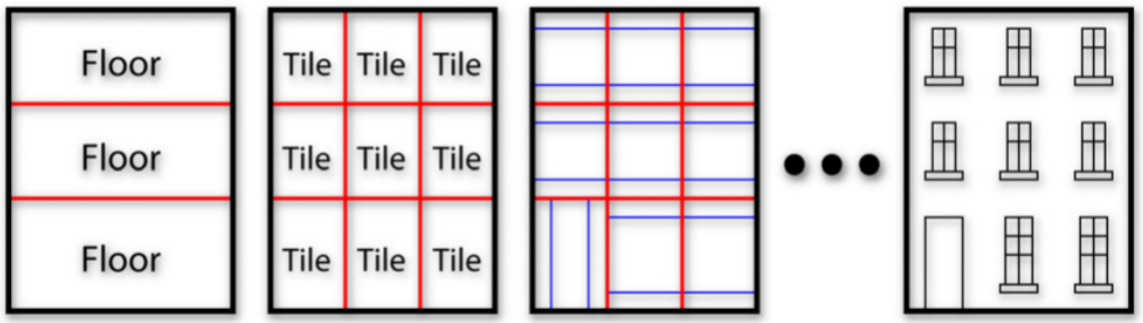

Figure 3.5: Hierarchical subdivision of the input facade into floors and tiles [6]

The image-based procedural modeling is among the best choices when one wants to replicate a building from the real world. However, on a larger scale, when the final goal is to generate an urban environment, the image-based technique may prove to be slightly inappropriate compared to the other approaches, since there are a lot of unnecessary computations involved which might drastically increase the response time. As a conclusion, this method can indeed provide remarkable results, but only if used in the right context and on a much smaller scale.

3.3 Procedural Generation of Cities

As stated in [7], the key element when generating a procedural city stands in the street network, i.e. a 2D representation of the interconnecting roads. After analyzing what repeating patterns exist in real-world maps, researchers came to the conclusion that street networks can be either grid-based, concentric or irregular.

Grid-based networks consist of perpendicular roads distributed in a regular checkerboard pattern and can be found in many modern US cities. In concentric networks, there exists one central point from which the streets radiate towards the city outskirts. As for the irregular networks, one can imagine their structure similar to that of an infinite tree with lots of branches going in random directions. The streets from the large European cities are usually arranged into radial or irregular patterns.

3.3.1 Extended L-Systems

CityEngine is the most widely used system for urban modeling nowadays and was first introduced in [8]. The application is very comprehensive, creating largely-scaled cities from scratch with complex buildings that are uniformly distributed on street networks.

In order to achieve the performance of generating entire cities in real-time, the L-Systems used in [4] for modeling individual buildings were further extended for urban environments. The CityEngine system follows a series of important steps until it reaches the desired output. First, it receives as an input a 2D map which is needed for generating the road network. Next, the areas between the roads are divided into building lots - regions that determine where each building will be placed. After the road network and building lots are set up, the buildings are generated in the same manner as in [4]. In the final step of the process, a parser interprets all the results for displaying them correctly on screen.

The procedural generation of cities based on L-Systems stands out from other existing approaches due to the high degree of realism it provides. Distinctive buildings can be obtained and the street networks are very similar to those from the real world. There is also implemented an Immediate Feedback system, where the user can see the results of each change in real-time. Figure 3.6 is an example of a scene that can be altered within the CityEngine interface. In the upper left side, there can be observed many sliders, each one representing an input parameter that can be interactively adjusted (e.g. the size of the city, the number of floors of each building, the width of a window).

Unfortunately, even though CityEngine has gained a lot of attention, it is not free of charge and might not be worth the money for those who need a simplistic representation of an urban landscape and nothing more. In addition, there is a problematic area which could be improved. Although the application requires a single 2D image for constructing sophisticated cities, it can be overwhelming trying to figure out the right patterns that should be included in the input data, since the resulting road network is an altered version of the provided map.

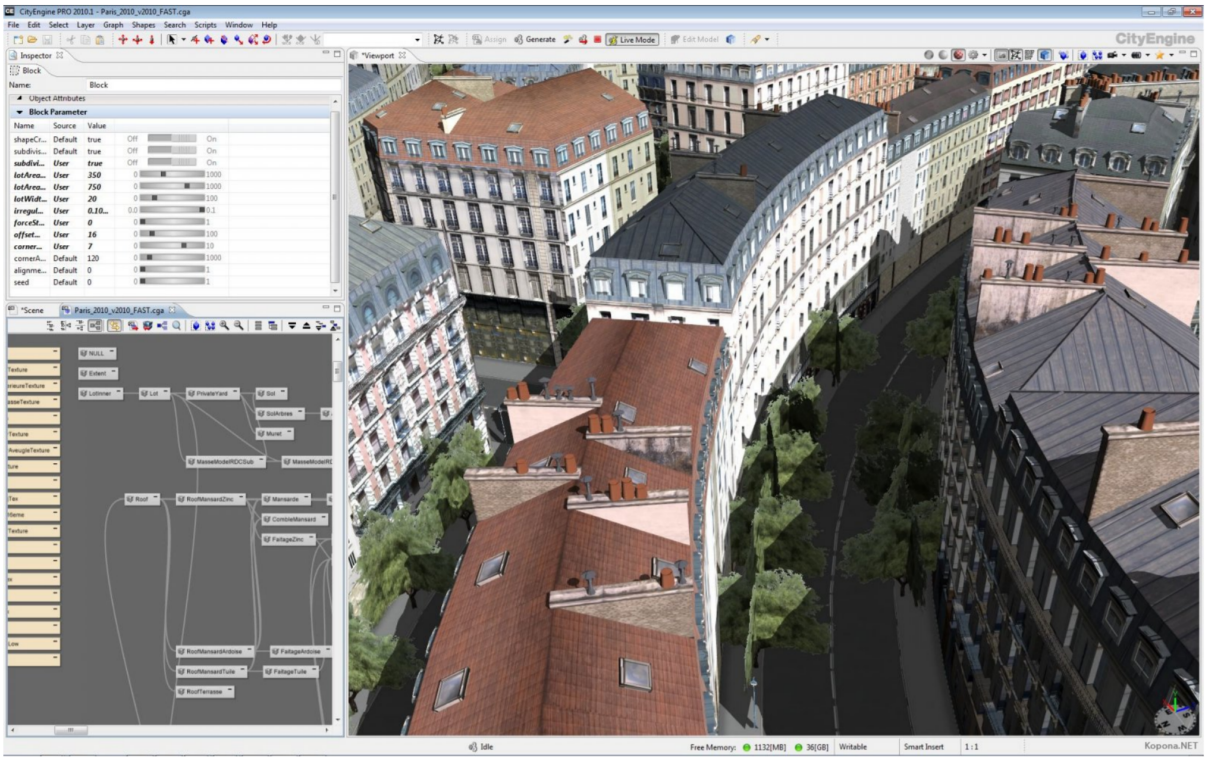

Figure 3.6: CityEngine interface [4]

3.3.2 Grid Layout

Paper [5] describes an efficient method capable of generating an urban landscape at run-time. The application creates a grid-based road network by dividing the terrain into square cells on a 2D surface. In each resulting cell, a procedural building is placed. The size of each block is constant, but can be adjusted globally.

An important aspect worth mentioning about the system implemented in [5] is that it comes with an excellent performance and extremely low response times due to an essential constraint that the application adopts: only the content that is inside the viewing frustum is generated and nothing else (Figure 3.7). This allows for massive cities to be rendered, since the amount of memory required to store the scene and the graphical processing power decrease enormously.

The application also presents a building cache, where buildings generated beforehand are stored. The building cache is based on the least recently used (LRU) principle, which keeps in cache only the recently-accessed buildings and replaces the old ones. Using buildings that are already stored in the memory instead of generating them from scratch

[4] Image source: http://web.ist.utl.pt/antonio.menezes.leitao/Rosetta/FinalReport/reports/ArturAlkaim-Report.pdf

saves a lot of memory usage and considerably improves the overall performance (up to 8 times).

Besides the advantages mentioned above, one area that can be further improved is concerned with the degree of realism that the application provides. Because of the grid-based road network and lack of diversity in terms of building architecture, the output city may seem too simplistic and unconvincing for some. This is another major focus of the current application, which will try to emphasize multiple architectural styles at once.

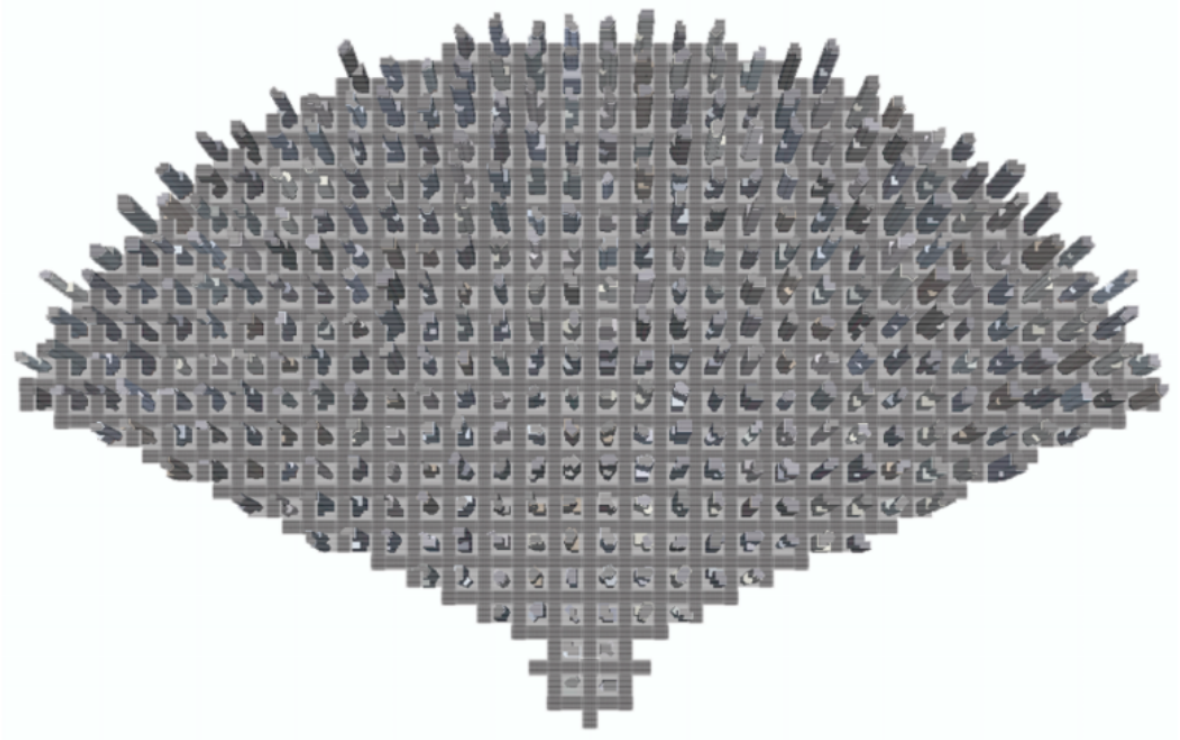

Figure 3.7: Only buildings visible within viewing frustum are rendered [5]

3.4 Technological Considerations

3.4.1 Unity

Unity is a cross-platform game engine developed by Unity Technologies, primarily used for video games and simulations. The engine offers support for creating 2D and 3D applications, being able to manipulate the objects and attach various components to them.

The programming languages that are compatible with Unity are C#, UnityScript, i.e. a scripting language for Unity with the syntax of Javascript and Boo, which has a

Python-like syntax. However, the most popular choice among consumers is undoubtedly C#.

The applications developed with Unity can be exported with one-click on multiple platforms: mobile (Android, iOS, Window Phone), desktop (Windows, Mac OS X, Linux), web (WebGL), console (PS4, Xbox One, PlayStation Mobile, Playstation Vita, Wii U) and TV (tvOS, Android TV, Samsung Smart TV). Unity is also the primary choice for VR (Virtual Reality) development, supporting some well-known VR headsets: Oculus Rift, Gear VR and Playstation VR.

A feature-rich and highly flexible editor is integrated in the Unity engine, speeding up the developers' workflow. As it can be seen in Figure 3.8, the Unity editor is divided into five sub-windows: Project Browser on bottom, Inspector on right, Game View in the middle, Scene View also in the middle and Hierarchy on left. One can switch between the Scene View and the Game View by selecting either the Scene or Game tab placed in the middle of the screen on top.

Figure 3.8: Unity Editor [5]

The Project Browser comprises all the assets that have been imported into the project and are available for use. The Inspector shows information about each GameObject, allowing the user to modify the values. In addition, this is the place where scripts can

[5] Source: https://blenderartists.org/forum/showthread.php?375039-Unity-editor-is-heading-to-linux

be attached or changed. The GameView provides a preview of how will the final application look. In the SceneView, the application is constructed, allowing users to drag and drop objects around the scene. Ultimately, Hierarchy displays a list of all the GameObjects that are currently placed in the scene.

The greatest motivation behind using Unity as a primary development tool is represented by its built-in rendering pipeline. Benefiting from this advantage minimized the time spent on the underlying logic and allowed me to fully concentrate on the project objectives.

Chapter 4

Analysis and Theoretical Foundation

This chapter will consist of two sections that establish the theoretical framework of the system. First section will elaborate on the algorithms that were used in implementing the current application and will explain in detail each step that needs to be carried out in order to replicate the final results. Although the used algorithms were briefly described in the previous section, a thorough explanation will be provided next, so that the readers gain a full understanding of the operating principles that hide beneath the system. As for the second section, it will focus on the use cases of the system that specify the sequence of actions that needs to be performed in order to accomplish each task.

4.1 Methods

The first and primordial condition that needs to be fulfilled before generating urban environments consists in having individual building models for use. Therefore, the initial major concern is represented by the procedural modeling of buildings itself. Only after the latter step is completed successfully one can proceed to creating virtual cities.

4.1.1 Procedural Modeling of Buildings

Each building was created in a layer-based approach, similar to the iterative design described in [4]. One important difference to mention though is that the buildings implemented in the GPMUL algorithm do not rely on grammar rules provided by the user, drastically simplifying the process, thus making it more suitable for the creation of virtual cities at a later stage. The algorithm starts from a basic 3D shape such as a plain cube (the first layer) and adds new layers within each iteration. The order in which the layers are disposed is the following:

- First layer - A simple cube (known as the building scope) as shown in Figure 4.1 - left. The scope is the top-level parent in the hierarchy, containing all the constituent

parts of the building. The length and width of the cube is a random value between the minimum and maximum widths specified by the user at run-time.

- Second layer - The building façades: front, back, left, right, top and bottom. Each building façade was obtained by cloning the initial scope, rescaling it accordingly and moving the object in the correct position afterwards. Figure 4.1 - right illustrates a building with the precomputed façades.

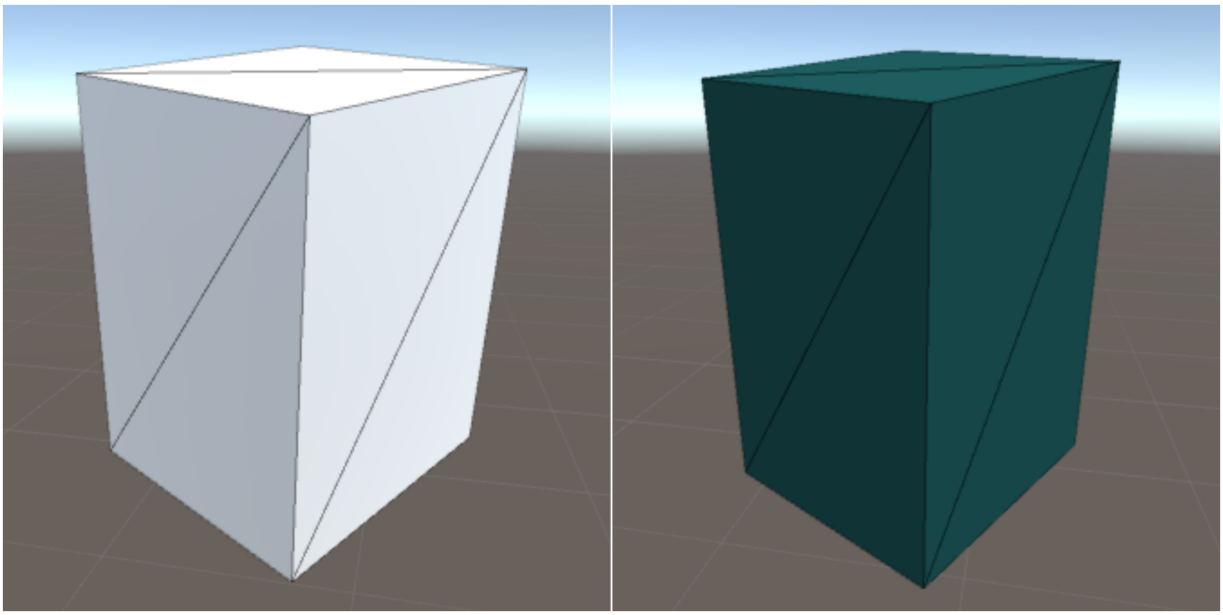

Figure 4.1: The building scope (left). The building façades (right)

- Third layer - The building floors. In other words, the front, back, left and right façades were subdivided into separate floors. Subdividing each façade means splitting it vertically, on the Y axis, in a predefined number of regions, as shown in Figure 4.2 - left, which illustrates a building subdivided into six floors. The number of floors was obtained by dividing the building height by a constant value introduced at run-time. A higher number indicated fewer building floors and vice-versa. After finding out the exact number of floors, the Y-scale of each floor was randomly yet carefully chosen so that it would fit the building height.

- Fourth layer - The window areas of each floor, as well as the door areas. More precisely, each particular floor from each façade is split horizontally, on the X axis, either into window or door areas. Figure 4.2 - right depicts such an example of a building, which is divided not only into separate floors, but also into window/door areas. The number of window or door areas was computed in a similar manner as above - the size of each building floor was divided by a constant value defined before

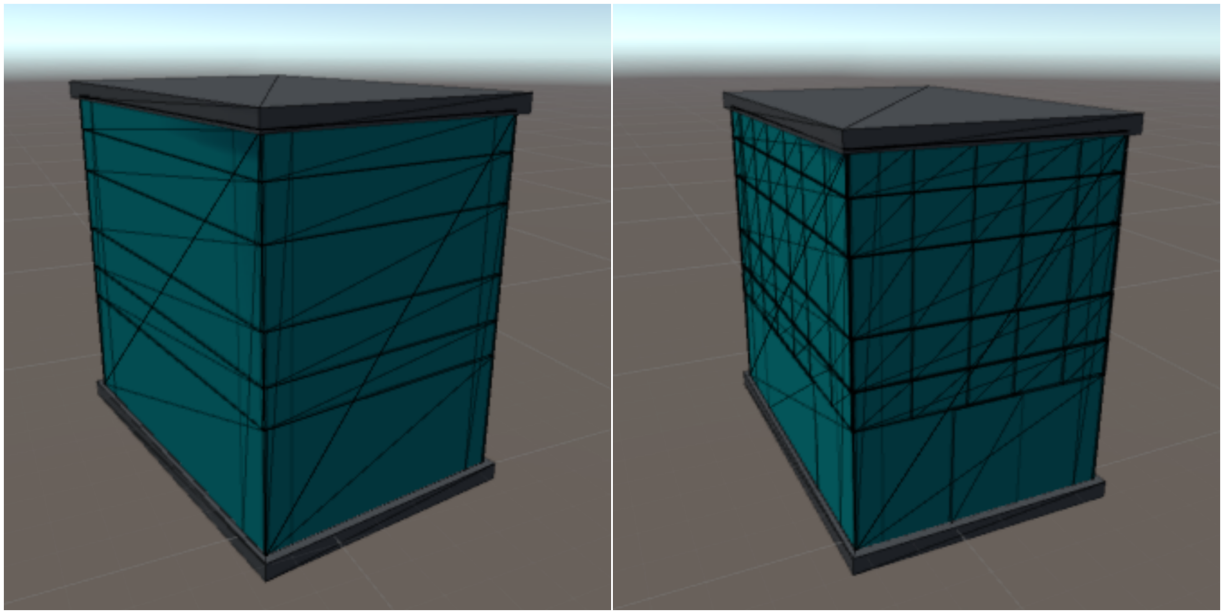

Figure 4.2: The building floors (left). Window/door areas (right)

generating the building. The X-scale and Z-scale values of each window or door areas were also calculated likewise.

- Fifth layer - The last layer that creates window and door objects placed in the window and door areas. The top side of the building was also customized, attaching a roof to the short buildings that have less than 2 floors. In Figure 4.3, the wireframe of a building in the final stage containing all the 5 layers is shown.

As it can be observed, the construction of buildings is heavily based on overlapping geometry, which comes with a significant drawback, namely the z-fighting problem. Fortunately though, this issue can be tackled by assigning different shaders to each particular layer.

Z-fighting problem is a phenomenon in 3D rendering that always occurs when two or more polygons have identical values in the depth-buffer. Since the building layers are coplanar and have the same z-positions with neither one in front, this results in a flickering effect, meaning that each layer will ”fight” to color the pixels of the exterior walls.

Shaders are an effective solution that can solve the z-fighting problem. A shader represents a computer program that controls the lighting, shading and special effects within an image. In other words, it tells the computer how to render each pixel. Shaders run on the graphics card or Graphical Processing Unit (GPU) to manipulate a 3D scene during the rendering pipeline before the image is drawn to screen. They are written in special shading languages. For instance, in this particular application, the shaders were written in ShaderLab, a special language provided by Unity.

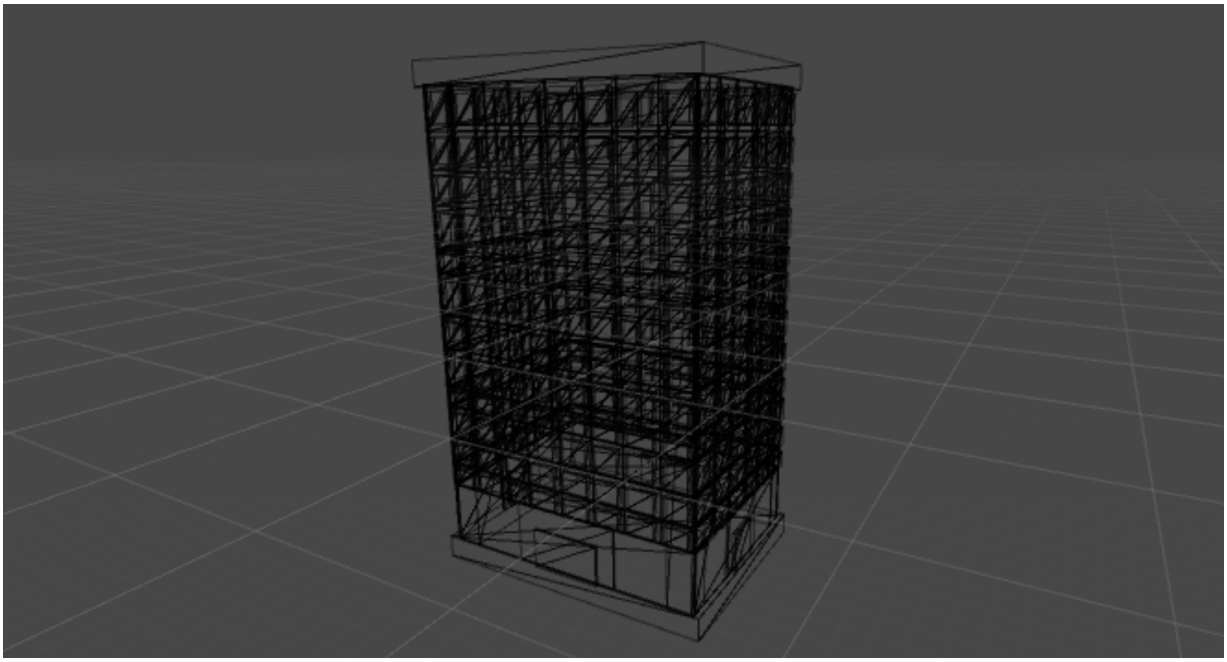

Figure 4.3: The building wireframe

Figure 4.4 illustrates the graphics pipeline, a conceptual model introduced in computer graphics to visualize the sequence of steps performed in order to render a 3D scene to a 2D screen. The pipeline starts with a 3D model and ends with a flat image on screen. The first stage of the rendering pipeline consists in defining an array of vertices containing attributes such as their position, normal vectors or texture coordinates in 3D space, which will be sent as an input to the vertex shader. Vertex shaders calculate the projected positions of the vertices in screen space, but can also modify the vertex attributes. The output produced by the vertex shader is sent to the primitives generation stage, where the triangles are assembled. This is achieved by taking the input vertices in order and grouping them in groups of three. The resulting triangles are sent to the rasterization stage, which takes each triangle and converts it into pixels. The fragment shader takes as arguments the rasterized triangles and returns the color and depth values that will be drawn into the framebuffer. Even though the most common operations performed in the fragment shader include lighting and texture mapping, they can also produce sophisticated special effects. Moving on to the testing and blending phases, this is the place where the fragments resulted from the fragment shader are processed. Depth testing is used to discard the fragments that are located behind other objects. Stencil testing discards fragments as well by testing the stencil value of each fragment against the value in a stencil buffer. If the test fails, then the fragment is culled. The fragments that remain after performing these two tests have their color value alpha blended with the color value they are overwriting. The final step implies drawing the final color, depth and stencil values into the framebuffer.

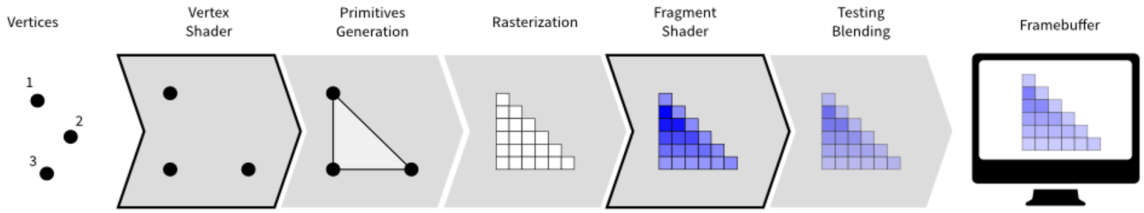

Figure 4.4: The graphics pipeline [1]

Going back to the generation of buildings, the issue of z-fighting can be fixed by assigning different shaders to each particular layer of the building. The key element stands in using render queues to determine the order in which the objects are drawn on screen. Although there are only four predefined queues, one can use a render queue with values in between. According to [9], the four commonly-known render queues are as follows:

- Background - This render queue is rendered before any others and has the numerical value of 1000. It is mostly used for rendering skyboxes.
- Geometry - This is the default render queue that is suitable for opaque geometry. It is rendered after the Background queue and has the value of 2000. The majority of objects use this render queue.
- AlphaTest - The AlphaTest queue is rendered after the Background and Geometry queues. Alpha tested objects use this queue and it has the value of 2450.
- Transparent - The Transparent Queue is rendered after the Background, Geometry and AlphaTest queues and has the value of 3000. It is useful for accurately representing glass objects or particle effects.
- Overlay - Anything rendered last goes in the Overlay queue. It has the value of 4000.

In the case of procedural building, the Geometry render queue was used, because the buildings are characterized by opaque geometry. The next shaders were assigned to the composite layers:

- First layer - Geometry queue (2000)
- Second layer - Geometry+1 queue (2001)
- Third layer - Geometry+2 queue (2002)
- Fourth layer - Geometry+3 queue (2003)
- Fifth layer - Geometry+4 queue (2004)

[1] Source: https://www.cs.duke.edu/courses/compsci344/current/classwork/15_shading/

Hence, by using render queues, a clear distinction can be made between what will be visible and what will be hidden in the end. Moreover, it is easier to create a logical hierarchy between the composite objects, as in the case of procedural buildings which are decomposed into basic components before being rendered.

Figure 4.5 illustrates a rendered building, after all the previous steps have been successfully accomplished.

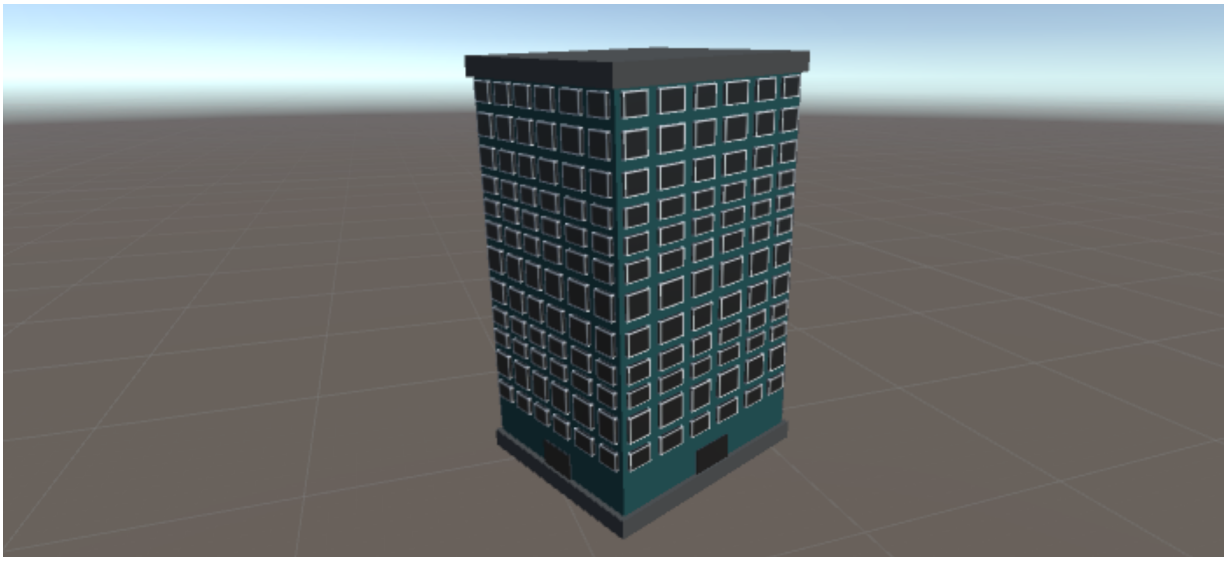

Figure 4.5: The rendered building

Lastly, Figure 4.6 shows the object hierarchy resulted after generating a building with 2 floors, 1 door on the first floor (one door per façade - 4 doors in total) and 3 windows on the second floor (3 windows per floor - 12 windows in total). One may notice that each new layer also adds a new level of nodes to the hierarchical tree as follows:

- First layer - Building node
- Second layer - Bottom, Front, Back, Left, Right, Top nodes
- Third layer - Floor1 (Front), Floor2 (Front), Floor1 (Back), Floor2 (Back), Floor1 (Left), Floor2 (Left), Floor1 (Right), Floor2 (Right)
- Fourth layer - DoorArea1, DoorArea2, DoorArea3, WindowsArea1, WindowsArea2, WindowsArea3 (on Front, Back, Left, Right façades), RoofArea (on Top façade)
- Fifth layer - Door, Window, Window, Window (on Front, Back, Left, Right façades), Roof (on Top façade)

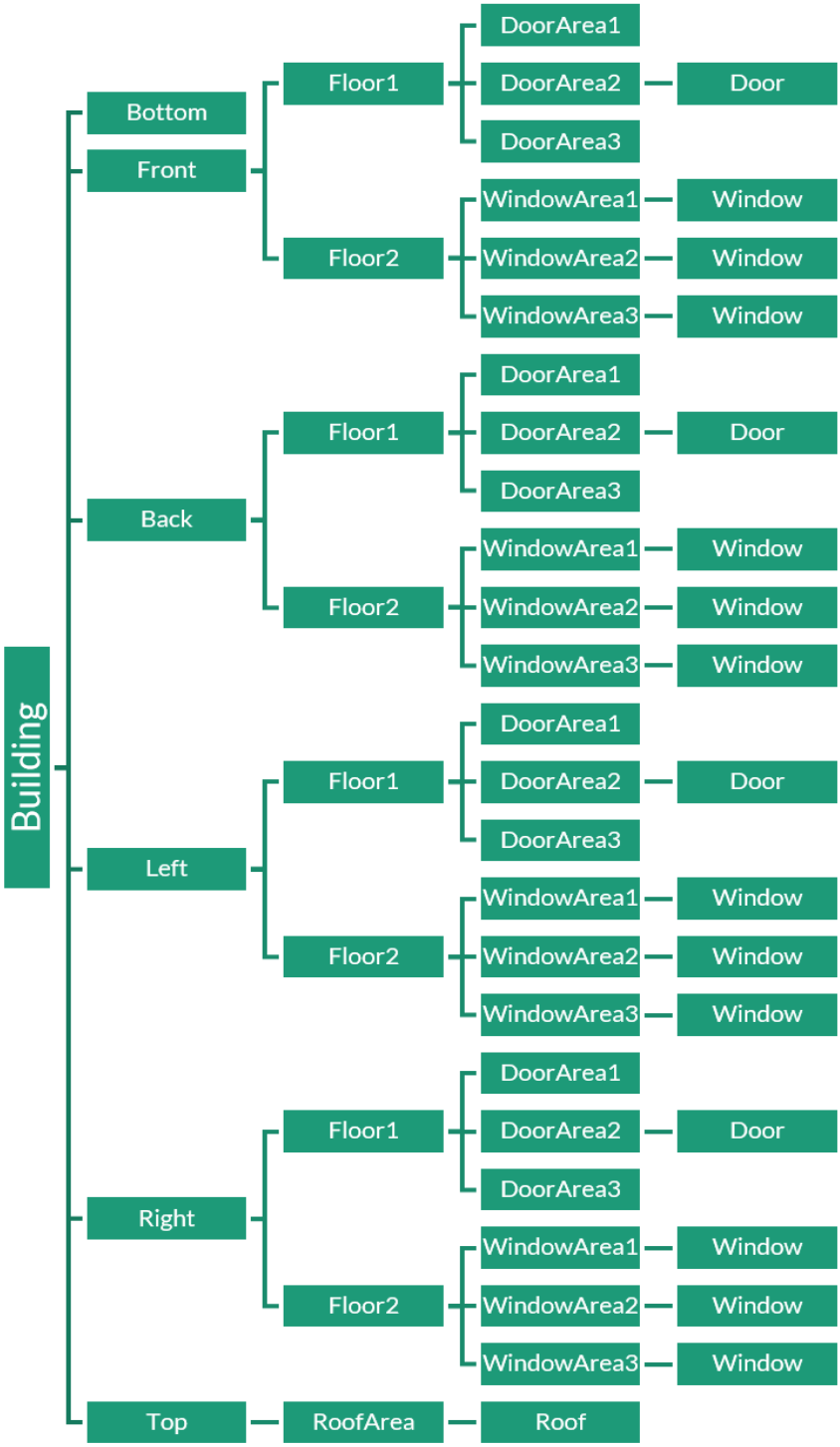

Figure 4.6: Building hierarchy

4.1.2 Procedural Modeling of Cities

After the task of procedurally modeling 3D buildings was successfully accomplished, generating urban environments became a straightforward process which consisted in simply spawning a large number of buildings across a plane. It is however important to mention that the procedural generation of landscapes is completely guided by the user, who has full control over the result. There are two basic steps that need to be performed before generating a 3D city. First and foremost, a grayscale image representing the city map has to be provided and secondly, the building parameters have to be properly adjusted for obtaining an accurate output.

To begin with, the input image that is required by the application is actually a building heightmap. A building heightmap represents an aerial view of the city, where all the buildings can be clearly seen. A distinguishing feature that characterizes the building heightmaps is the easiness by which one can determine the height of each building. Only by analyzing the colors' brightness from the map, one can clearly tell if a building is taller or shorter than another one. Therefore, the key element when creating a building heightmap stands in the shades that are used. The darker the shade, the taller the building, and otherwise. In the context of the current application, only grayscale heightmaps are allowed, since they provide a simplistic yet highly effective way to represent the buildings distribution across a city.

Each input map has a similar structure, consisting of grayscale rectangles. As for each rectangle, it defines a specific region from the city, where several buildings will be placed. The height of each building is solely determined by the area to which it belongs. Lighter shaded rectangles will be populated with taller buildings, whereas the darker shaded rectangles will produce shorter buildings. Thus, the tallest buildings will reside in the white rectangles, whereas the shortest buildings will belong to the rectangle that has the darkest shade of gray. The black zones from the map will not be filled with any buildings.

Figure 4.7 shows a working input map that was used in the application for testing purposes. Each rectangle will comprise several buildings of varying heights. The tallest buildings will be found in the first rectangle from the first row, while the shortest buildings will be placed into the first rectangle from the third row. In a similar fashion as stated above, the rest of the buildings will be generated. The lighter a rectangle, the taller a building within its region and the other way round.

The next step that needs to be followed after successfully uploading the map consists in adjusting a set of parameters before generating the actual content. These parameters are represented by floating-point values and have different purposes. The input parameters that can be modified are the following:

- Density - This parameter defines the buildings density and refers to the number of buildings that can be found within the city. In other words, the higher the density value, the higher the number of buildings that will be distributed across the city. A low density value indicates a city that is sparsely populated with buildings.

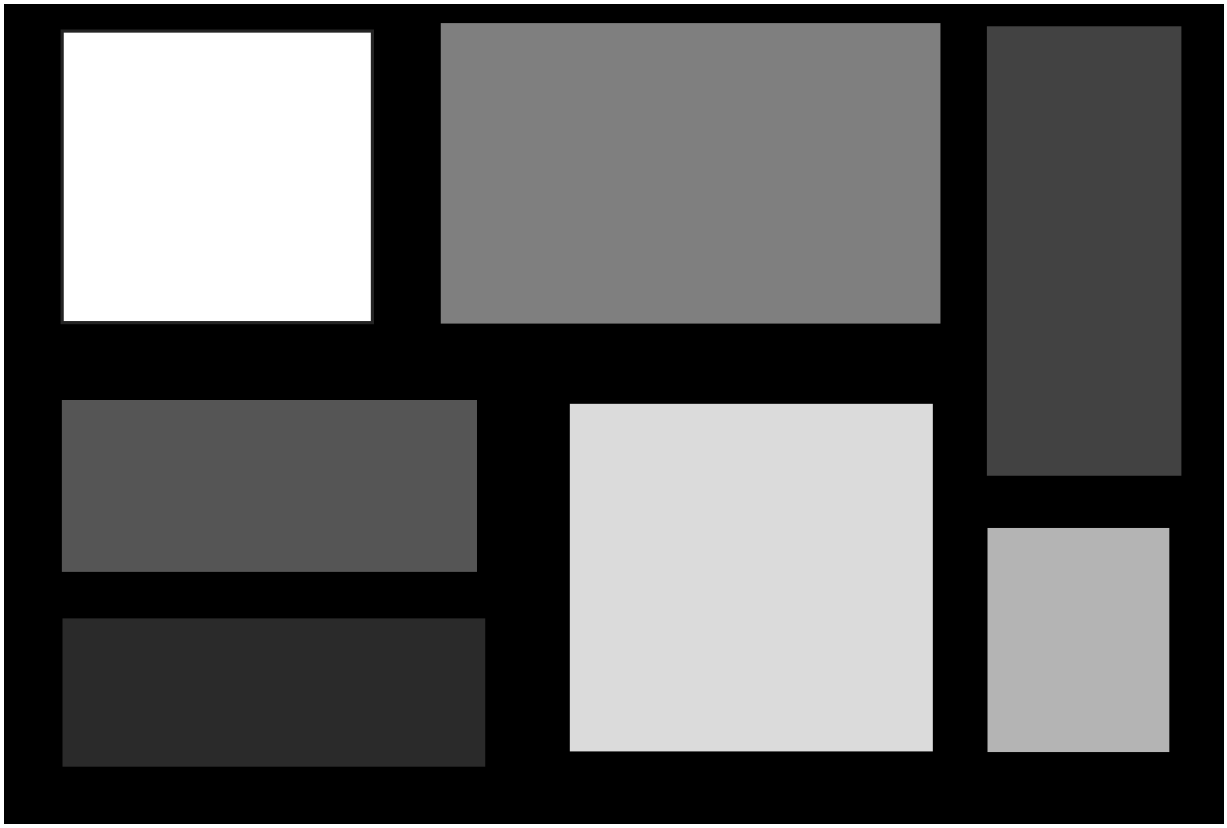

Figure 4.7: Building heightmap

- Height - While the gray shades from the input map dictate the buildings heights, the user can also control how visible the difference of heights will be. If a lower value is introduced, the building heights as a whole when inspecting the city from above will not be very noticeable. On the other hand, a higher value will produce heights whose values vary greatly between each other, leading to a non-uniform distribution.
- Minimum width - Although the width and length of each building are randomly chosen, the values have to fit in a certain range. This ensures that the buildings are not ridiculously narrow or extremely wide, increasing the degree of realism. In this case, the minimum width represents the lowest range that is allowed for the widths and lengths. More exactly, all the buildings will have the width and length greater or equal than this minimum width.
- Maximum width - The maximum width represents the highest range that is permitted for the width and length of each building. To put it differently, all the buildings will have the width and length less or equal than this maximum width.
- Floor constant - As explained in the previous section, in order to find the number of floors of each building the total height has to be divided by a constant value. The

floor constant defines that value. As a basic idea, a low floor constant will produce more floors and a high floor constant will result in fewer floors.

- Window constant - When constructing a building, the number of windows on each floor is determined by dividing the total width by a constant value. The window constant represents that value. In the same way as above, a low window constant will generate more windows, whereas a high window constant means fewer windows.

After the two basic steps are finished, the system gathers all the data it needs and the procedural generation begins. However, for ensuring a uniform distribution of the buildings across the city and avoid placing them in straight lines which would look rather inaesthetic, several sampling points from the map are chosen. These points are random points uniformly distributed on the surface of the input image. Each building will then be placed on the position of such a point. A higher number of sampling points will lead to a higher density of buildings.

The algorithm by which the sampling points are chosen follows a simple idea. The input image is divided into a grid of equal rows and columns and then, in each newly formed sub-region, an equal number of sampling points is distributed. Scattering sampling points over a surface is equivalent to generating random x and y coordinates within that surface and ensuring that they do not overlap with other points. Figure 4.8 shows a set of sampling points generated given the previous input map. Each sampling point will be replaced by a building in the exact same position. Since the x and y coordinates of each point on the map are preserved, it will not be very difficult to identify the grayscale value of the points and render the buildings. Though, as an exception to the rule, the points that are spawned across black areas will not be replaced by any buildings, because black represents the ground level.

By using this approach, some cases which involve overlapping buildings might occur. However, this issue can be easily solved by checking before generating the actual building whether it would intersect with the already existing buildings or not. In case it does, the new building is no longer created to not interfere with the current ones.

4.2 Use Case

Use cases encapsulate the functional requirements of the system, i.e. what operations the system must be able to perform. Therefore, by relying on such an approachable method right from the inception stages of the application, one can identify the key actions which need to be prioritized in order to deliver a high-quality product. Use cases provide many advantages because they are able to capture the exact sequence of steps required in order to perform specific tasks. Besides, their simplistic representation makes them more accessible as they can be analyzed and interpreted effortlessly by anyone without prior knowledge in the software development field.

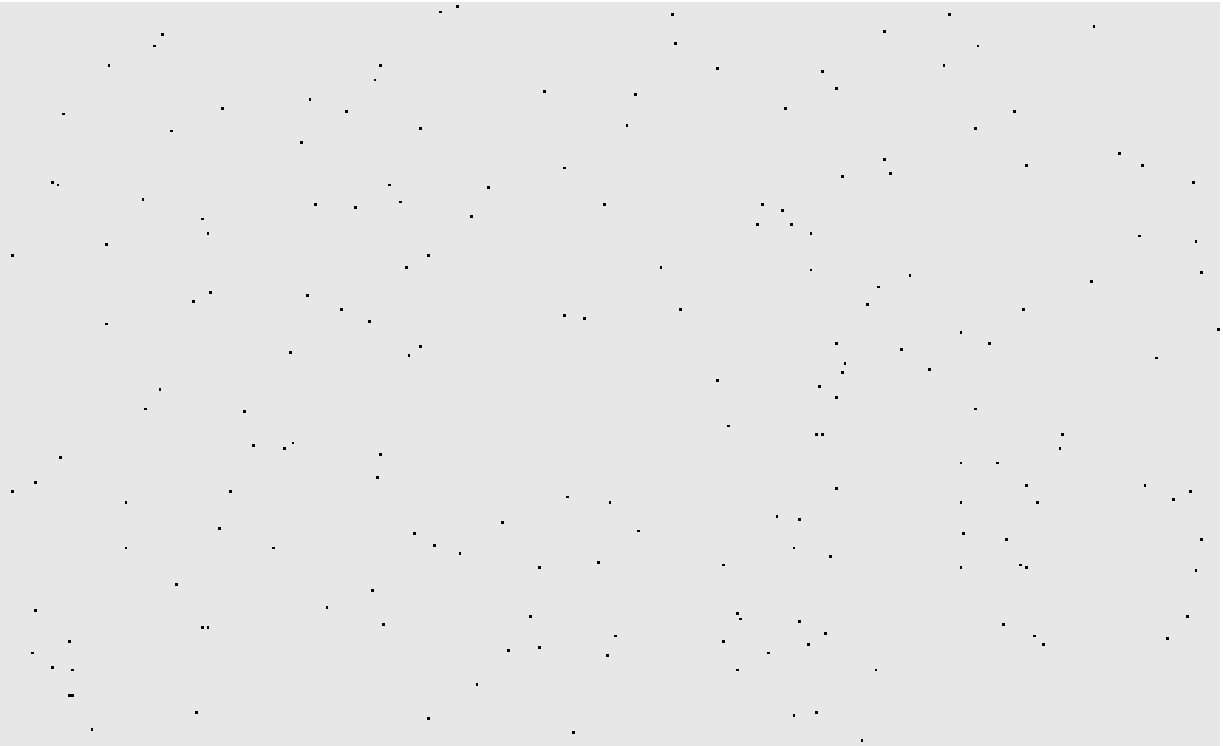

Figure 4.8: Random uniform points across the map

4.2.1 Use Case Diagram

To gain a deeper understanding of how the application works, a use case model will be presented below. Four important use cases have been identified. Each one contributes by some extent to achieving the final goal of the project, i.e. exporting the 3D model and using it for the formerly intended purposes. The four use cases that will be depicted in Figure 4.9 are the following:

- Load building heightmap
- Adjust parameters
- Generate 3D city
- Export model

4.2.2 Use Case Description

1. Load building heightmap
   - Primary actor: The user

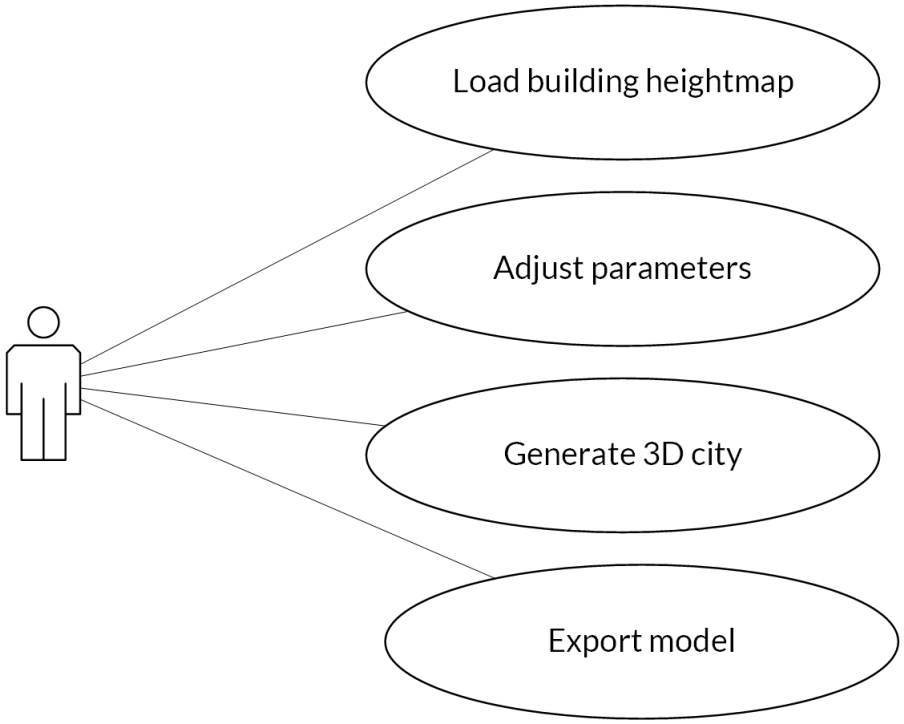

Figure 4.9: Use Case

- Purpose: The user wants to load a 2D image into the application
- Preconditions:
  1. The application is running
  2. The image is grayscale
- Postconditions:
  1. The image is displayed in the application
  2. The system will generate a city based on the input map
- Main flow:
  1. The user clicks on the ”Upload” button
  2. The user selects an image from the file browser
  3. The image is successfully loaded into memory
  4. The system displays the image and its name
- Extensions (Alternate flow):

  2.a. The user selects another file instead of an image
    1. The system displays an error message
    2. The user selects a new file having the correct extension (.jpg, .png)

  2.b. The user selects an image that does not conform to the rules

1. The system displays an error message
2. The user selects a valid image

- Special requirements: None

2. Adjust parameters

- Primary actor: The user
- Purpose: The user wants to adjust the input parameters before generating the city
- Preconditions:
  1. The application is running
- Postconditions:
  1. The city parameters are altered
- Main flow:
  1. The user modifies the input parameters through sliders
  2. The user presses the ”Generate” button
- Extensions (Alternate flow):

2.a. The image is not loaded

1. The system displays an error message
2. The user selects an image from the file browser

- Special requirements: None

3. Generate 3D city

- Primary actor: The user
- Purpose: The user wants to generate the city
- Preconditions:
  1. The application is running
  2. The image is loaded
  3. The parameters are properly adjusted
- Postconditions:
  1. The system will generate a city based on the user's description
- Main flow:
  1. The user loads the map image
  2. The user adjusts the input parameters
  3. The user click on ”Generate” button

- Extensions (Alternate flow):

  1.a. The user selects another file instead of an image
    1. The system displays an error message
    2. The user selects a new file having the correct extension (.jpg, .png)

  1.b. The user selects an image that does not conform to the rules
    1. The system displays an error message
    2. The user selects a valid image

- Special requirements: The system graphics card must support 3D rendering

4. Export model

- Primary actor: The user
- Purpose: The user wants to export the 3D city that was generated by the system
- Preconditions:
  1. The application is running
  2. The city is displayed on screen
- Postconditions:
  1. An .obj file can be found in the file system
- Main flow:
  1. The user clicks on the ”Export” button
  2. The user selects from the file browser the path where the file will be saved
  3. The system saves the 3D model in the specified location
- Extensions (Alternate flow):

  3.a. The save operation fails because the system does not have write permissions

- Special requirements: The system must have write permissions

Chapter 5

Detailed Design and Implementation

This chapter will be the most comprehensive one, presenting in depth the design of the system along with the implementation details. The first section will be dedicated to the conceptual architecture of the application, which will help visualize the ideas more clearly and identify the major components at a superficial yet meaningful level. After offering a high-level visualization of the system, the chapter will move forward to the second section, which will focus on the class diagram and will explain the role and importance of each particular module within the proposed scheme. Finally, the last section will describe the User Interface (UI) of the application and will discuss its efficiency from an usability point of view.

5.1 Conceptual Architecture

Conceptual architecture is concerned with identifying the major components of the system and the way in which they interact. It extends the functional requirements identified in the previous use cases and determines how the system should be decomposed in order to obtain proper results. Furthermore, highlighting the key elements without delving into too much details provides the big picture of the system and determines its strong points, i.e. where it will excel.

5.1.1 Conceptual Architecture Diagram

The major downside of other procedural modeling tools consists in the lack of control and usability. Since they require prior knowledge in the field of computer graphics and geometry, the average user would spend hours - in the best case scenario - trying to learn how to use the system instead of achieving the formerly intended goal. This consequence would not bring benefits to either side, because the application might deliver poor-quality results that do not meet the user's expectations and are therefore impractical. Consequently, unsatisfied users will not reuse the same application all over again due to its ineffectiveness. Another problem that needs to be pointed out is that the algorithms

do not take into consideration the user's needs and produce random and unpredictable results. Even though this might be convenient for others, there are also users who need a customized representation of a virtual city. The current application will try to solve the existing weaknesses by prioritizing the usability.

The application comes with an intuitive user interface accessible to all the newcomers. One does not need to spend additional time to figure out how the system works, since it does not contain any abstract terms or extra options hidden from the public. Instead, it only displays the input parameters required by the algorithm in a simplistic manner, without overcomplicating the interface. As a whole, the user interface consists of buttons, sliders and a panel which encompasses the 3D urban landscape that is generated. This chosen design encourages the users to take full control over the results and guide the system towards producing the desired output.

Moving on to interacting with the newly-created scene, a first-person controller was implemented to allow the users explore the city from a one-point perspective and decide whether it suits their needs or not. In case everything comes out as expected, they can export the city and use it to their hearts' content. On the contrary, if the result is still lacking some features, they can re-generate the city by altering the input parameters or change the building heightmap. The conceptual diagram from Figure 5.1 illustrates the most significant components of the system and the relationships among them.

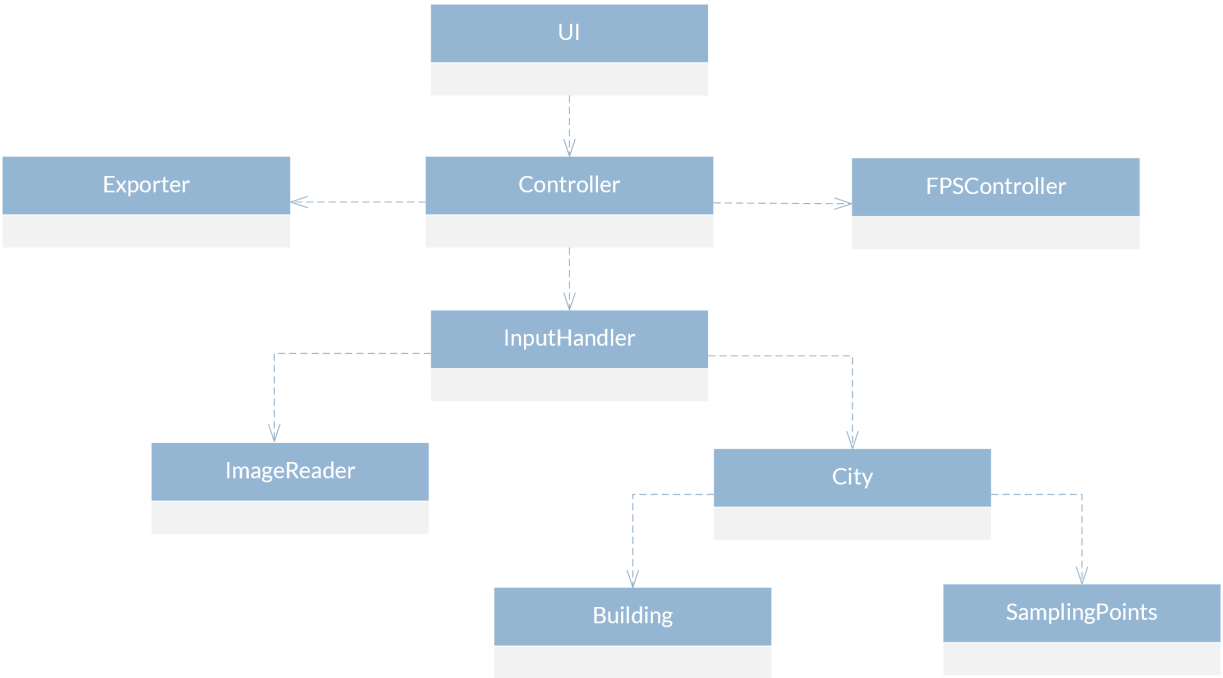

Figure 5.1: Conceptual Architecture

5.1.2 Components Overview

As it can be observed above, the architectural components are structured in a convenient logic inspired by the Model-View-Controller (MVC) pattern. The key elements on which the current application is based are the following:

- UI - The primary purpose of the User Interface is to determine what will be rendered on screen. Moreover, it receives user inputs and sends them to the Controller for further handling.

- Controller - The Controller is the central point within the application and deals with the overall implementation logic. It links the UI elements to their corresponding event listeners and updates the data displayed on screen when the user manipulates the view.

- Exporter - The Exporter concentrates on creating a basic .obj file from the previously generated urban landscape. The latter file can be saved on the local disk in a specified location, by selecting an appropriate path from the file browser.

- FPSController - The First-Person Controller makes it possible for users to navigate through the environment in the first-person mode. Experiencing a 3D scene in such a realistic manner provides many advantages compared to simply exploring it by zoom or rotation and this is the primary motivation for implementing this component.

- InputHandler - The InputHandler receives from the Controller the input parameters and uses them to generate urban landscapes. For performing this operation, it sends the gathered data forward to the ImageReader and City components.

- ImageReader - As the name states, the ImageReader reads all the bytes from the image given its path and returns it to the InputHandler.

- City - The City component is responsible with generating urban landscapes based on the input parameters received from the InputHandler. First, it uses the SamplingPoints component to distribute several random points across the surface of the map, and then, it spawns buildings in the positions of these points through the Building component.

- Building - The Building component is used for creating unique buildings based on the user's description. The generated buildings are sent to the City component, which distributes the buildings over an urban landscape.

- SamplingPoints - The SamplingPoints class scatters a predefined number of random uniform points across the 2D surface of the input map. Each sampling point will be replaced by a building in the exact same position.

5.2 Class Diagram

The greatest motivation behind a class diagram stays in the level of detail it offers. Instead of illustrating only the fundamental components on which the system is based, it provides a comprehensive description of the overall implementation. Thus, a class diagram extends the conceptual architecture to a much wider extent, elaborating on the entire development of the system. Figure 5.2 shows the class diagram of this application. It contains all its composite classes, as well as the most relevant attributes and methods within each class. What needs to be pointed out though is that the diagram still lacks some trivial attributes and repetitive methods. They were skipped for avoiding any ambiguities and simplifying the scheme, making it more intelligible.

As it can be noticed from the figure on the next page, half of the classes are derived from MonoBehaviour. MonoBehaviour represents the base class of all the Unity scripts. If a class extends the MonoBehaviour class, then it means that is uses the services provided by Unity in order to build a 3D graphical environment. Another thing worth mentioning is that the City class plays the primary role within the application, creating urban landscapes with the data gathered from the Controller, InputHandler and ImageReader. As for the generation of large landscapes, the City class uses the Building class to produce unique buildings and the SamplingPoints class to scatter random points across the surface of the map.

The following sections will examine each individual class from the figure, providing an in-depth analysis of all its constituents.

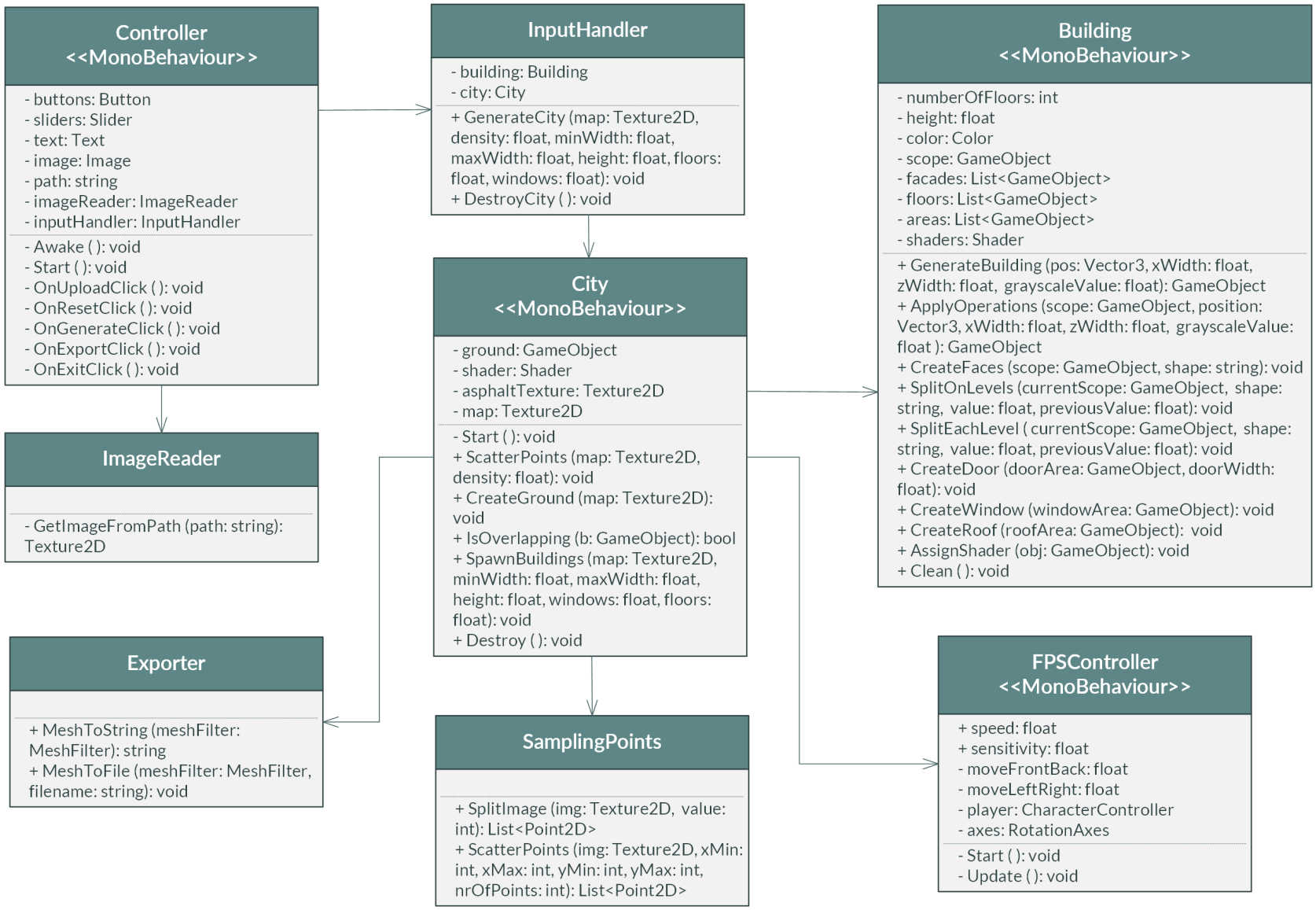

Figure 5.2: Class Diagram

5.2.1 Controller

The Controller handles the user interaction, acting as a middle layer between the UI elements and the functional components. Whenever someone presses a button or modifies the value within a slider, an event is triggered, informing the system of what actions to perform. There are three types of UI elements that this application uses, namely buttons, text areas and sliders. They are further detailed in the following section (Section 5.3) that conducts a complete analysis of the UI.

The buttons displayed on screen are: Upload, Reset, Generate, Export and Exit. They have onClick listeners attached to them which call functions that pass the gathered data forward to the InputHandler and ImageReader. In the case of the sliders, they make it possible for the users to alter the input parameters of the algorithm, such that the results correspond to their needs. For ensuring that the urban landscapes will turn out realistic, the sliders are characterized by minimum and maximum values. Values within these predefined ranges can be adjusted by the users. While interacting with the sliders, the Controller calls methods that update the values of the text fields in the UI. Each slider has an associated text field that displays the current value of the sliders in real-time as the users modify it.

When uploading an image from the local disk, the Controller saves its path into the path variable. Having the path field defined allows then for displaying the input image on the UI in a small rectangle, along with its name and extension. Moreover, the ImageReader receives the file path as a parameter and converts the image to a 2D texture that can be further manipulated by the InputHandler.

5.2.2 ImageReader

The ImageReader class, as the name suggests, has a single responsibility - reading images. It achieves this task through the method called GetImageFromPath which encloses a two-step process. First, the system reads the content of the input image given its path and saves it into a byte array. Secondly, it transforms the image into a 2D Texture by loading the byte array into an empty texture. This very approach of splitting the application logic into multiple classes ensures low coupling and high cohesion.

5.2.3 InputHandler

The InputHandler can be visualized as a helper class between the UI and the City component. Every time an event occurs, the Controller captures it and attaches a callback function that tells the system what to do next. However, the Controller cannot take full responsibility over everything and has to transfer a part of the workload to another specialized classes. This is where the InputHandler becomes useful. The Controller collects all the needed data and forwards it to the InputHandler, whereas the InputHandler uses the received data for passing it to the city generator itself.

There are three methods residing in the InputHandler class: GenerateCity, DestroyCity and ExportModel. GenerateCity is the most meaningful function, interacting directly with the city generator for producing urban landscapes. Before producing an urban landscape, several sampling points from the loaded image are chosen and the buildings are uniformly distributed in these positions. The GenerateCity method takes as parameters the input map provided by the user and the additional variables that describe the output: density - which defines the number of buildings within the city, minWidth - representing the lower bound range of the buildings' widths, maxWidth - representing the higher bound range, height - establishing the height values, floors - indicating the number of floors and windows - responsible for the number of windows. Moving on, the DestroyCity function is called when someone presses the Reset button and has the sole purpose of removing the 3D content that is displayed on screen. ExportModel is the opposite method that can save the 3D content shown on screen as an .obj file. The saveLocation and filename parameters indicate the place where the landscape should be saved and under which name.

5.2.4 City

The City class is the central component within the system responsible for the procedural generation of urban landscapes. There are several methods implemented here, each one contributing to reaching the final goal of creating 3D scenes:

- Start - Start is a lifecycle method derived from the MonoBehaviour class that is called once in order to initialize the script. The City component uses this function for loading the customized shaders into the application and for setting up the camera in a convenient position to ensure that the users visualize the 3D scene from a bird's eye view.

- CreateGround - This method creates an asphalt surface on which the buildings will be later placed. The asphalt surface consists of a 2D plane having the dimensions proportional to the input map. For making sure that the results will not exceed the vertex limit of 65,534 vertices, the input map was rescaled to 10% of the original size. This approach minimizes the response times and allows the users to upload any image irregardless of its resolution.

- SpawnBuildings - This method is responsible for the distribution of buildings on the formerly generated ground. The x and y coordinates of the random points that were uniformly distributed on the 2D surface of the map are saved in a list such that they can be accessible in the subsequent phases. In the following step, the algorithm iterates through the list of sampling points and checks whether the pixel color is black or not. The colors of the pixels can be easily determined given the x and y coordinates of the points. A black pixel color indicates a ground-level height and therefore no buildings will be placed in that position. On the contrary, if the pixel color has another grayscale value different from black, then it will be replaced by a

building whose height depends on the color brightness in the exact same position. To solve the issue of overlapping buildings that might occur, an auxiliary method named CheckIfItIntersects was implemented, which verifies before placing each building in the scene if it is overlapping with the already existing ones. In case it does, the building is destroyed for not compromising the results.

- ScatterPoints - This function uses the SamplingPoints class in order to distribute random uniform points across the surface of the provided map. The number of scattered points is determined by the density parameter whose value is specified at run-time before generating the buildings landscape.

- CheckIfItIntersects - The primary purpose of this method is to prevent spawning in the scene buildings that overlap. Before placing a building in the cityscape, this function verifies if it intersects with the already existing buildings. This is done by comparing the bounding box of the new building to the bounding boxes of the other buildings. If an intersection occurs, then the new building is destroyed and not added to the list.

- Destroy - Destroy is the operation performed when someone clicks on the Reset button and wants to re-generate a new city. It wipes out the scenery being displayed on screen, including the concrete ground and each particular building. After the scene is removed, the users will be able to upload another map and start the process all over again.

5.2.5 Building

Another important class on which the city generator depends is represented by the Building module. This class is the most comprehensive one and is concerned with the generation of each individual building, embodying the following methods:

- GenerateBuilding - This method creates a simple cube primitive as the starting block from which the building will be modeled, passing it forward to the ApplyOperations function that constructs the final form of the building given its parameters. The parameters are used to specify the building's position in the scene (x, y, z coordinates), its dimensions and the grayscale value which dictates the height. The GenerateBuilding method is called by the City component every time it creates a new building that does not intersect with the other ones.

- ApplyOperations - This function takes full responsibility for modeling the 3D buildings. It comprises all the action steps that need to be followed, relying on auxiliary functions defined within the same class to accomplish the desired results. In the initial step, the height of the building is calculated according to the grayscale value received as a parameter. The height formula can be altered though, adjusting the

value of the height slider from the UI. Increasing the height value will produce buildings whose heights are more discernible, being easier to observe from the distance. On the other hand, selecting a lower value for the height will result in buildings whose varying heights are harder to distinguish if one does not have a closer look upon the scene. After the building height is identified, the formerly created cube is rescaled accordingly based on the input parameters provided by the user in the UI. The latter cube is then decomposed into façades by calling the CreateFaces method that returns the exterior sides of the cube, namely the front, back, left, right, top and bottom areas. The number of floors within the building is established by dividing the previous height to a constant value specified by the user at run-time. A higher value will result in fewer floors, while a lower value will produce a higher number of floors. The next phase has the purpose of splitting the front, back, left and right façades into separate floors. In other words, each façade from those enumerated above will be split on the Y axis and subdivided into several regions. Having the number of floors computed, the algorithm continues by generating random heights for each floor and afterwards, each floor is rendered based on its size. A similar technique is applied for generating window and door areas, the only difference being that they are obtained by splitting the building floors on the X axis instead. After the floors, windows and doors areas are rendered, the building is sent back to the City component that uniformly distributes it into the scene.

- CreateFaces - The CreateFaces method receives as parameters the building scope representing the initial cube from which the algorithm starts, and a shape string which tells the system what façade to generate (either ”front”, ”back”, ”left”, ”right”, ”top” or ”bottom”). Each exterior side of the building is created in a similar manner. The scope is cloned and the duplicate object is parented by the initial cube, to form a logical hierarchy. Next, the clone is rescaled until it resembles a 2D plane and then it is repositioned depending on the target location given by the shape string. Because the buildings are constructed in a layer-based approach, the z-fighting problem is prevented by assigning shaders to each different layer of the building. The building façades are characterized by the same shader, that basically tells the system to render the initial cube first and only after that its exterior sides.

- SplitOnLevels - This method renders the floors of a specified façade. The façade is specified by the shape string parameter.

- SplitEachLevel - Similarly to the previous method, SplitEachLevel is used for rendering the window and door areas of each floor. The shape string identifies the façade on which the floor resides, whereas the currentScope parameter defines the building floor which will be split horizontally.

- CreateDoor - This function creates a door mesh parented by the doorArea object. The door width is given by the doorWidth parameter.

- CreateWindow - This function creates a window mesh parented by the windowArea object.
- CreateRoof - This function creates a roof mesh parented by the top façade.
- AssignShader - The AssignShader method was created with the Don't Repeat Yourself (DRY) principle in mind, trying to eliminate any duplicate code. It attaches appropriate shaders to the objects. Each shader offers valuable information about the render queues, solving the depth buffer inconsistencies that might occur.

5.2.6 SamplingPoints

The SamplingPoints class contains the algorithm that scatters random uniform points across the surface of the 2D map. Two methods are implemented here: SplitImage and ScatterPoints. To begin with, the SplitImage method divides the input image into an equal number of rows and columns, creating a grid. The grid dimensions are given by the value parameter. In each small square within the grid, the ScatterPoints function is called, distributing an equal number of points in that region. ScatterPoints receives as parameters the input image, the minimum value and the maximum values for the x and y coordinates and the number of points that will be generated in each grid square. Separating the image into smaller parts ensures that the sampling points are distributed more evenly on the bitmap.

5.2.7 FPSController

The FPSController component provides a means of navigation through the scene and handles the collisions. A first-person perspective simulates the field of view that one perceives in real life, contributing to a higher degree of realism. The FPSController class includes one method that solves two problems at once.

The Update method is derived from the MonoBehaviour class and is called once per frame. This is the place where all the first person controller logic resides. The player field is used to identify the main character that will be controlled by the users when they press W, A, S or D and rotate the mouse. Besides, it supports collision detection, imposing constraints to the users when they try to enter a building. The first person camera is enabled only after an urban landscape is generated and shown on screen. It can be paused and reloaded by pressing the Space bar. The speed attribute controls the character's speed while moving forward, backward, to the left or to the right. As for the sensitivity field, it defines the rotation angle by which the camera will rotate when the users move the mouse.

5.2.8 Exporter

The Exporter component allows the users to save the 3D urban landscape in memory and use it as they wish at a later stage. It consists of two methods, namely MeshToString

and MeshToFile. The MeshToString method provides a string representation of the provided GameObject, including information about each vertex, normal, uv and triangle. The MeshToFile function processes the generated string, saving it into an .obj file where it can be directly visualized without the use of any graphical tools. The MeshFilter parameter belongs to the city model that gathers the ground and all the buildings in a hierarchy, whereas the filename string is the absolute path where the file will be saved.

5.3 User Interface

Designing high-quality interfaces is a task overlooked by many. Unfortunately, this very problem often leads to the failure of numerous software systems, due to their steep learning curves. Even though an application might deliver exactly what it promises, it becomes worthless if the users spend too much time learning how to use it instead of benefiting from its results. Therefore it is extremely important to set the UI development as a top priority, taking into consideration that the users interact directly with it.

5.3.1 User Interface Overview

As a whole, the UI is split in two major sections: the left-hand side, that is solely dedicated to the input parameters and the right-hand side, consisting of a rectangular panel where the urban landscapes are displayed. Figure 5.3 illustrates the application's UI in the initial state, when no map is uploaded and no parameters are adjusted. One may observe that the right panel is empty. This indicates that no 3D content has been generated yet.

The left-hand side of the application was designed in a simplistic manner to ensure that the novice users encounter no difficulties when generating virtual environments. As shown in Figure 5.3, it can be noticed that the interface follows a logical workflow and encourages the users to start the process from top to bottom, by uploading the map first and adjusting the values subsequently. However, one can perform the order of operations as desired as long as all the requirements are met, not influencing the final results in any way. After the map is successfully loaded, a rescaled version of it will be displayed in the small white rectangle. Visualizing the input image is helpful when the 3D content is shown on the right-hand side, helping the users check more efficiently whether the output meets their expectations or not. Another thing that needs to be outlined is that the users cannot export the models before they are generated. That is the reason why the Export button is disabled. The following section will elaborate on each UI element, describing its purpose and offering examples of usage.

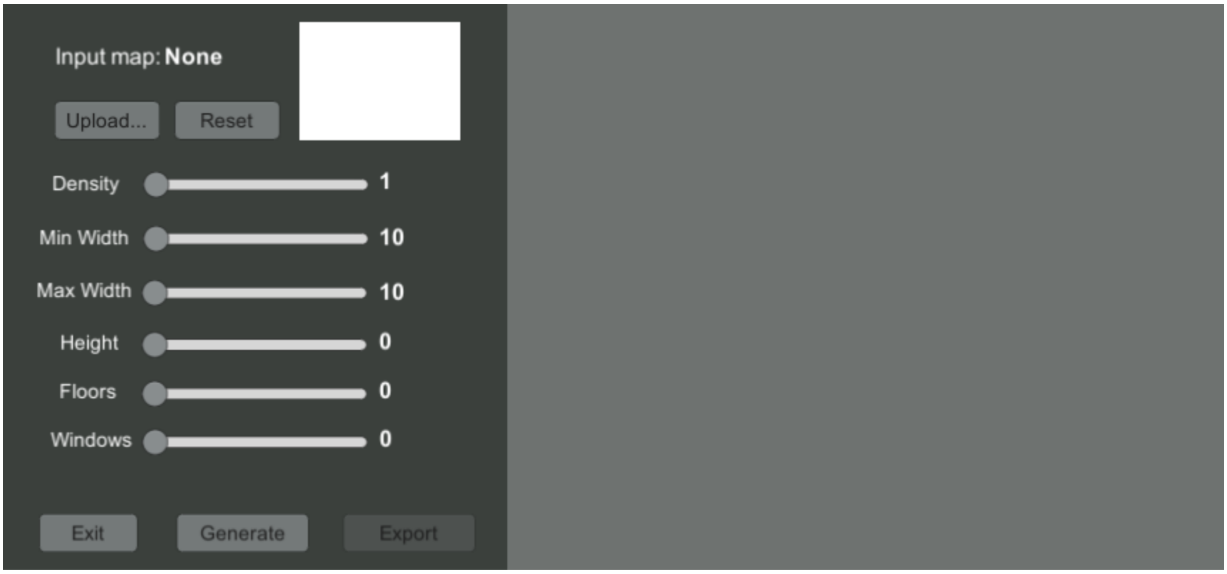

Figure 5.3: User interface

5.3.2 User Interface Elements

As previously mentioned, the UI is divided in two significant parts: the input section on the left-hand side and the results panel on the right-hand side.

The primary purpose of the input section is to offer the users full control over the output. They can guide the algorithm into producing accurate landscapes that match the descriptions they provide. The input section contains three major components:

- Image upload - This component is located on top and contains one text area, two buttons (Upload and Reset) and a small rectangular panel. The text area indicates the name and extension of the image that is currently loaded. If no image is loaded, the text "None" is displayed. When the user clicks on the Upload button a file browser like the one in Figure 5.4 appears, where a grayscale image can be selected from the computer and loaded into the application. Provided that the input image is valid, it will be rescaled accordingly and shown in the white rectangular panel as illustrated in Figure 5.5. In case the user presses the Reset button and there is no content generated yet, the image will be erased from memory, the rectangle's color will change to white and the application will return to its initial state. If the Reset button is pressed while a 3D scene is displayed on the right-hand side of the screen, the system will delete that scene with no possibility of recovery, as well as the input map.

- Parameters configuration - This component is located in the middle and consists of six sliders. These sliders are depicted in Figure 5.6. Each slider represents an input parameter that is needed by the algorithm before generating the 3D environment.

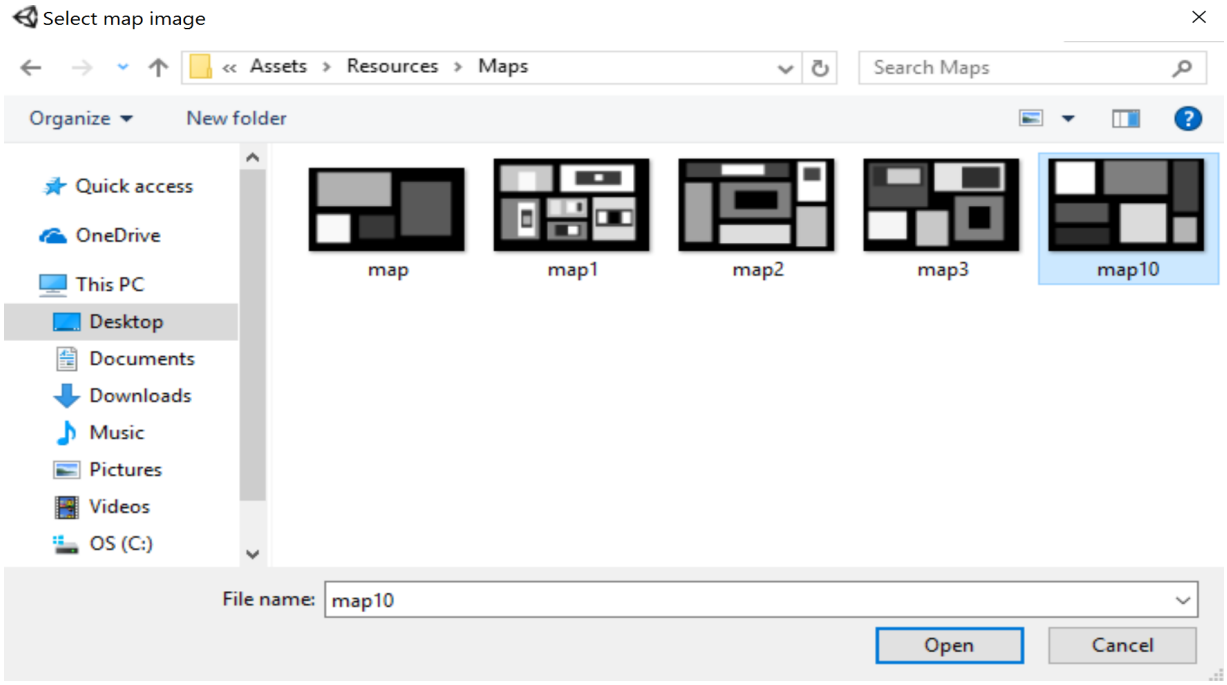

Figure 5.4: File browser

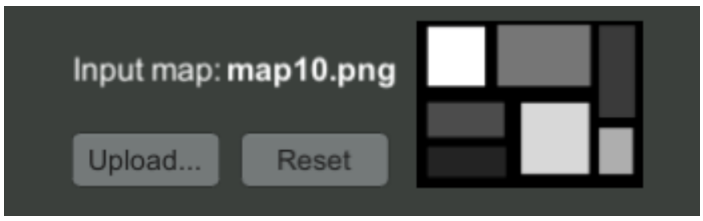

Figure 5.5: Image successfully uploaded

The first attribute is called Density and refers to the urban density within the city. A higher density value produces a larger number of buildings. As a rough estimate, the approximate number of buildings that will be generated is the square of the value specified in the Density slider. However, this is not an exact estimate due to the overlapping buildings that are removed in the process. The next parameters on the list are represented by Min Width and Max Width. They determine the dimensions of the buildings that will be spawned across the city. When a building is created, its width and length values are randomly generated within a specified range. Min Width and Max Width represent the lower and upper bounds in that range. Consequently, the Min Width should be smaller than the Max Width, otherwise an error will occur. Continuing with the Height parameter, it controls the level of variation between the buildings' heights. Last but not least, the Floors and Windows parameters determine the number of floors and windows from each building. A lower value for the Floors

and Windows will result in a higher number of floors and windows and vice-versa.

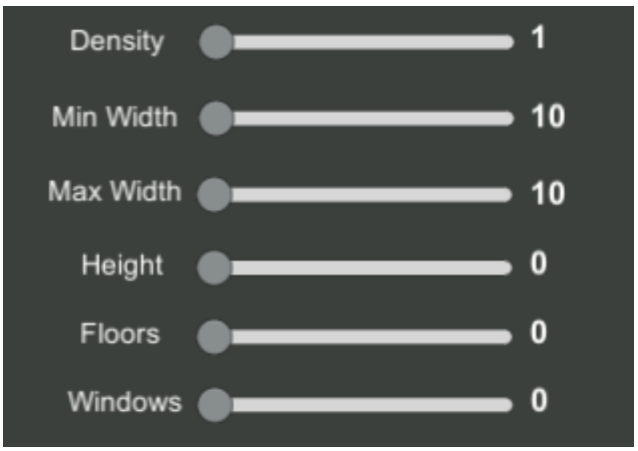

Figure 5.6: Adjustable parameters

- Buttons - There are three buttons located on the bottom-side: Exit, Generate and Export, as pictured in Figure 5.7. The Exit button is used to close the application. The Generate button is pressed only after all the necessary data has been provided. If the input image is missing or the sliders' values are invalid, the system will throw an error. Otherwise, if the parameters are correctly configured, the application will construct the urban landscape based on them, displaying the final result on the right panel. The Export button is disabled when no 3D model is shown on screen, becoming active when an urban landscape is generated. It allows the users to save the 3D scene in .obj format, in the path selected in the file manager.

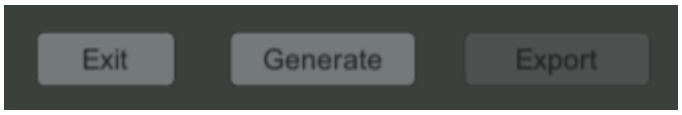

Figure 5.7: Bottom-side buttons

The results panel on the right-hand side is empty when no 3D content was generated beforehand or the user pressed the Reset button. This container is responsible for showing the users the generated 3D content. The first person camera allows them to navigate easily through the urban landscape and examine the scene in depth. On top of that, the system gives permission to pause and reload the character movement at all times by simply pressing the Space bar. Figure 5.8 presents an urban landscape that was generated based on the input parameters from the left. One may notice that the buildings' heights correspond to the grayscale values from the input map and the Export button is now enabled.

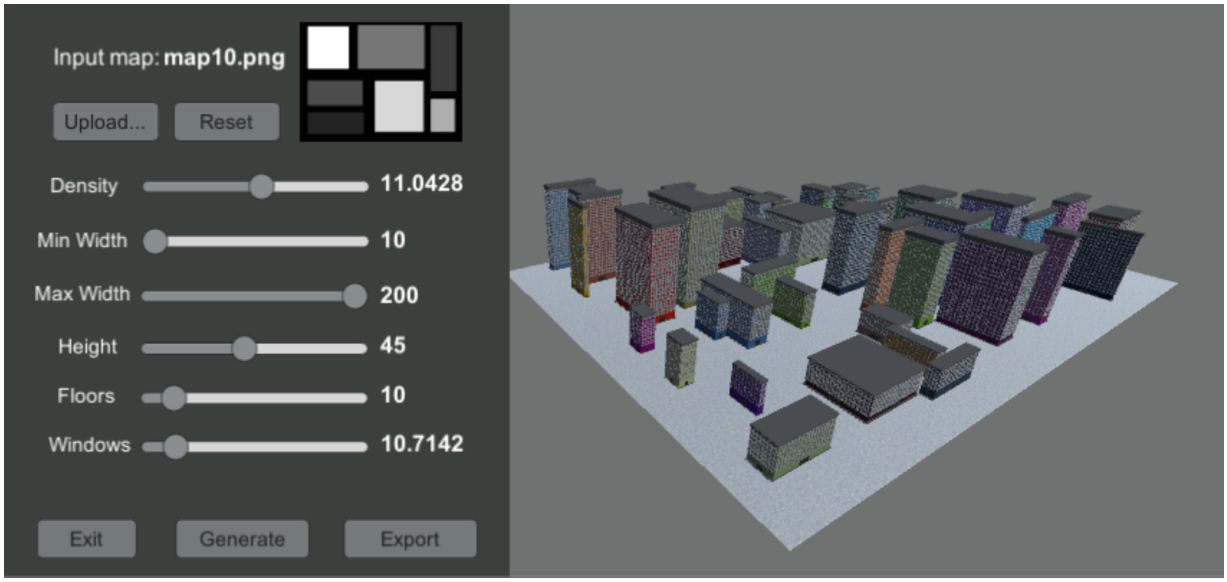

Figure 5.8: Urban landscape

Chapter 6

Testing and Validation

Software testing is one of the key elements that guarantees a system behaves correctly and delivers high-quality results. Testing is a strong technique by which one can identify in time the functionality flaws and fix them before they lead to serious consequences. This application was tested in small increments while being developed. More precisely, each component was addressed separately to verify if it meets the demands. The components were integrated in the system only after making sure they ran smoothly and did not threaten the application performance-wise.

The purpose of this chapter is to highlight the two fundamental testing strategies that need to be taken into consideration when delivering a software product and discuss them in the context of the current application. The first section will deal with the functional testing techniques and will include several examples of usage to verify the accuracy of the output. In the second section, nonfunctional testing will be presented to examine the quality of the system from a user's perspective.

6.1 Functional Testing

Functional testing is concerned with testing the components within the system to ensure they function properly. Carrying out such tests is a safe and sure approach that detects the major weaknesses for further improvements. A great emphasis is put on the system's requirements, therefore one of the major goals of functional testing stays in identifying whether the system performs all the operations that were planned in the early stages of development. The following sub-chapter will describe the manner in which the most significant modules of the application were tested before being integrated.

6.1.1 Tested Components

- Building - The Building class is the first and most complex module that was developed. Since all the buildings are constructed iteratively, several tests were carried out within each iteration before proceeding to the next one. As one may recall,

each building comprises five overlapping layers. Thus, the tests focused on checking whether the z-fighting phenomenon occurs when adding a new layer over the others. This very problem was solved by configuring custom shaders and assigning them to their corresponding layers. Another test that was performed verified the randomness degree to which a building was generated. Each building turned out to be unique. However, several boundaries were established, to avoid generating highly-unrealistic buildings. There were also tests concentrated on the level of details of each building to determine if the doors and windows were correctly disposed, without any geometrical gaps in between. That is how an optimal formula for placing each building component in the right position was conceived and adapted on the long run.

- City - The City component is determined by external factors, namely the distribution of sampling points on the provided map, alongside with the generation of each individual building. Only after assuring these components operated smoothly was the City class tested. The City class was tested directly from the UI, when the application was in the final phase. The technique used relied heavily on generating as many urban landscapes as possible with different input parameters and examine their level of accuracy. The accuracy degree was established by answering two essential questions: "Are the grayscale values from the input map reflected in the urban landscape?" and "Did the adjusted parameters provide what they stated?". Several examples of usage will be included in the future section of this chapter that will elaborate on this matter.

- Controller - The Controller was manually tested by constantly interacting with the application when it was under development. Testing the Controller is a straightforward approach that was done directly through the UI. Because each UI element has an event listener attached, a normal behavior that the Controller should expose is to call the corresponding handler function when an element is being manipulated by the user. This test was successfully passed, due to the proper behavior of the buttons and sliders that guide the procedural algorithm. They performed the correct operations, without any mismatches.

- ImageReader - The ImageReader class encloses a single method responsible for converting an image into a texture that will be further used by the application. This functionality was tested by debugging the application for checking three aspects. The first question that came up was whether the path existed and the file was found. Secondly, the content of the byte array was verified for ensuring that the image's pixels were successfully read. A last check was performed on the content of the texture after loading the byte array into it.

- InputHandler - Since the InputHandler is a central component that connects the UI elements to the functional operations within the application, testing it was relatively

easy. The technique used consisted in altering the input parameters and generating multiple landscapes, while paying attention if the correct methods are being called.

- SamplingPoints - The SamplingPoints component was individually tested before being integrated into the system. Black points were initially scattered across the surface of a white image and then the image was saved in memory, to visualize the results and check the correctness of the algorithm. The parameters were also varied to cover all the test cases. For example, splitting the image into more rows and columns resulted in a higher density of points that were uniformly distributed over the image. After the desired level of granularity was reached and no overlapping points were detected, the component was fully integrated into the application with slight changes. These changes were necessary due to the incompatibility of the Unity engine with the basic graphics functionality provided by C#.

- FPSController - The FPSController was tested at run-time, while using the application. After a 3D environment was generated and shown on screen, the FPSController allowed to examine the scene from a first-person perspective, by using the keyboard and the mouse. Significant effort was put into ensuring that the users would benefit from a smooth experience. More specifically, the speed of movement was adjusted so that the character would not move too fast or too slow, but somewhere in between. Besides, the rotation angles were adjusted in the same manner, to accurately resemble the human field of view.

6.1.2 Tested Workflow

This section will provide a step-by-step example of how the city generation algorithm works. What will be omitted though is the procedural modeling of individual buildings, concentrating instead on how the buildings are spawned in a largely-scaled scene.

In Figure 6.1, a complex map was provided, along with the requested input parameters. After random points were distributed across the surface of the image, they were saved in a list that was sent back to the City component. The urban landscape was constructed in an iterative manner, going through the list of sampling points and replacing each point with a building given that its pixel value was not black. Figure 6.1 shows the cityscape obtained after the first 25 iterations. One may observe that one iteration does not necessarily produce a building. This is because of the overlapping geometry that might occur during the process. Therefore, a sampling point is not guaranteed to yield a building. Even though all the points are considered, the buildings will be placed carefully in the scene so that they do not intersect.

Figure 6.2 shows how the urban landscape turned out after 50 iterations (left), 75 iterations (middle) and 100 iterations (right). A conclusion that may be drawn is that the number of buildings within the scene increases along with the number of iterations that are performed.

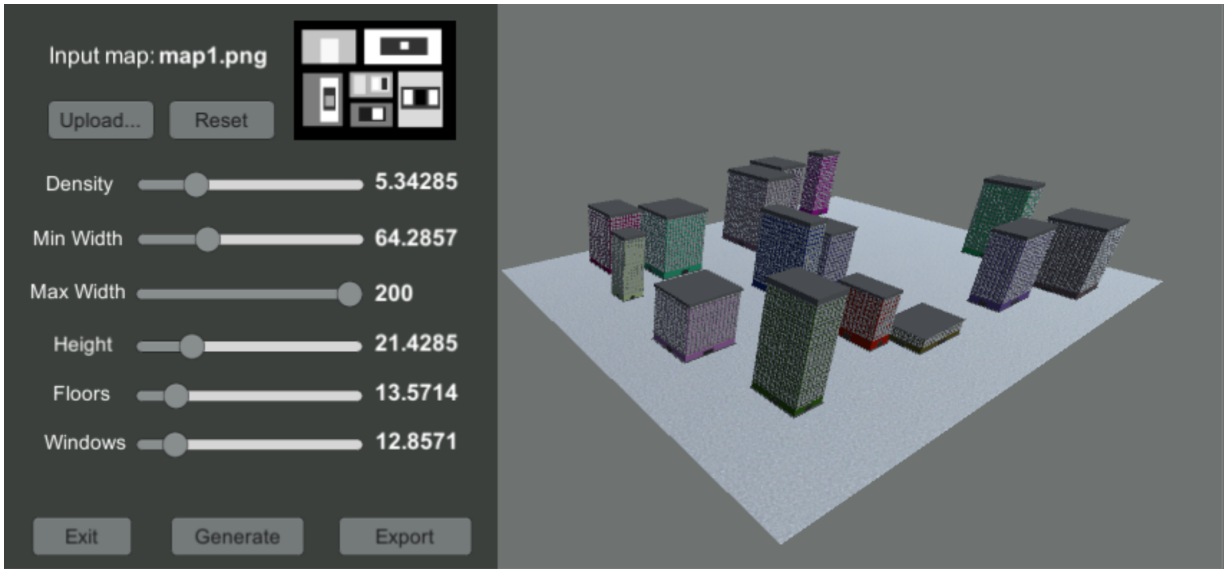

Figure 6.1: Urban landscape

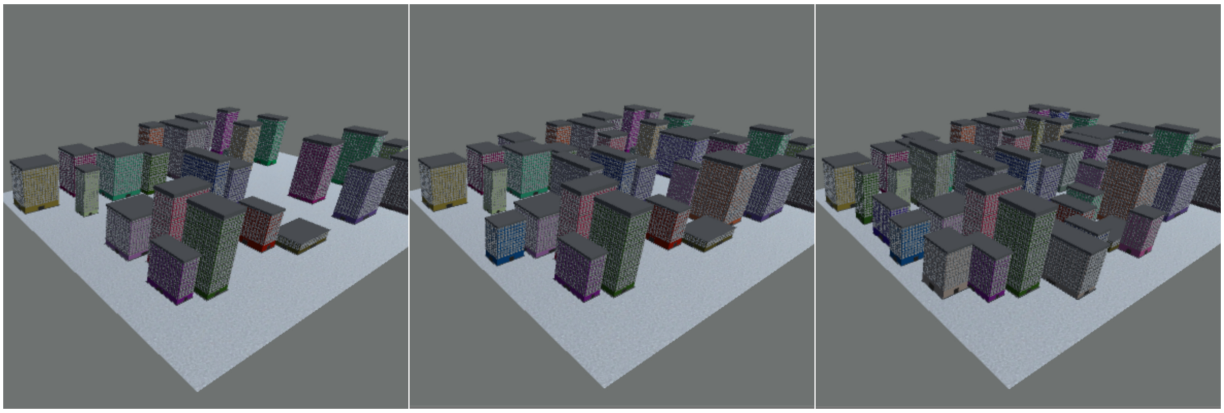

Figure 6.2: Iterations

6.1.3 Building Heightmap Accuracy

Alongside the tested workflow, a set of different input maps were used to generate urban areas in order to verify the correctness of the output in terms of building heights.

In Figure 6.3 - left, an image with light-shaded rectangles was provided. Translated in simpler terms, the system was informed to place the tallest buildings in the upper right area, the second tallest buildings in the upper left area, the third tallest buildings in the bottom left area and lastly the shortest building in the bottom right area. As can be seen from Figure 6.3 - right, the urban landscapes resulted in the expected behavior.

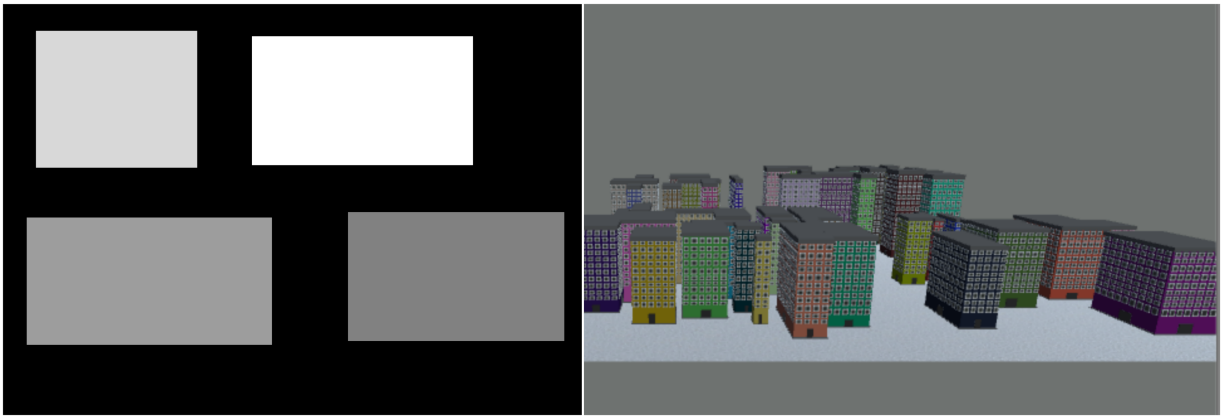

Figure 6.3: Tall buildings

Figure 6.4 - left deals with the opposite case, that being the generation of short buildings. Converted to natural language, the system was informed this time to place the shortest buildings in the bottom left area, the second shortest buildings in the right area, the third shortest building in the bottom middle area and ultimately the tallest buildings in the upper middle area. As shown in Figure 6.4 - right, the initial predictions proved to be true. One may remark that some of the buildings have a roof on top. That is due to the rule that adds a roof to any building having two or fewer levels.

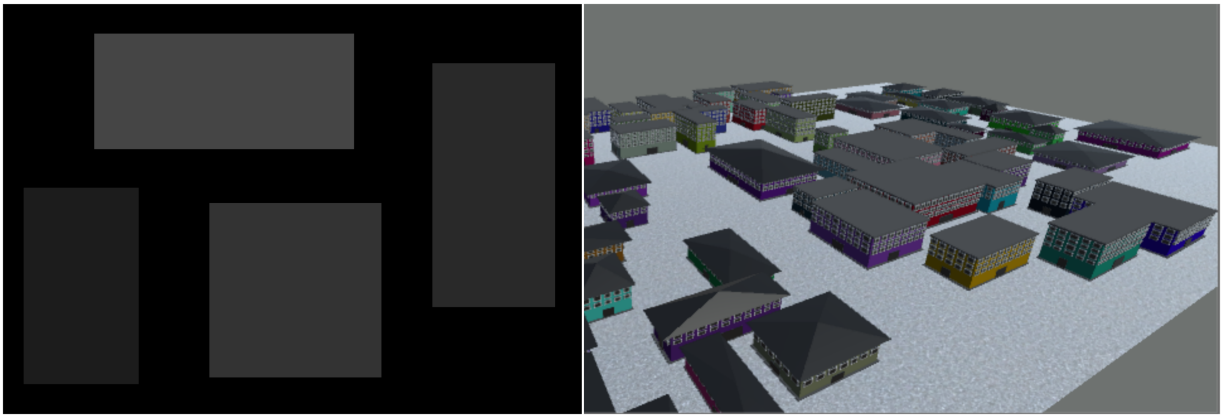

Figure 6.4: Short buildings

Another test that was carried out focused on buildings with mixed heights. In order to perform this test, the map from Figure 6.5 - left was uploaded. To explain the input image more clearly, it consists of four rectangular zones having the same color. That means that all the buildings residing within those zones should have similar heights. In each of these four rectangles, a smaller rectangle was placed, having a contrasting color. The expected behavior is as follows: the tallest buildings should be placed in the upper right area, the second tallest buildings should be placed in the upper left area, the third tallest

buildings should be placed in the bottom right area and no building should be placed in the bottom left area. All the building groups previously mentioned should be surrounded by medium-sized buildings. As depicted in Figure 6.5 - right, the system performs accordingly.

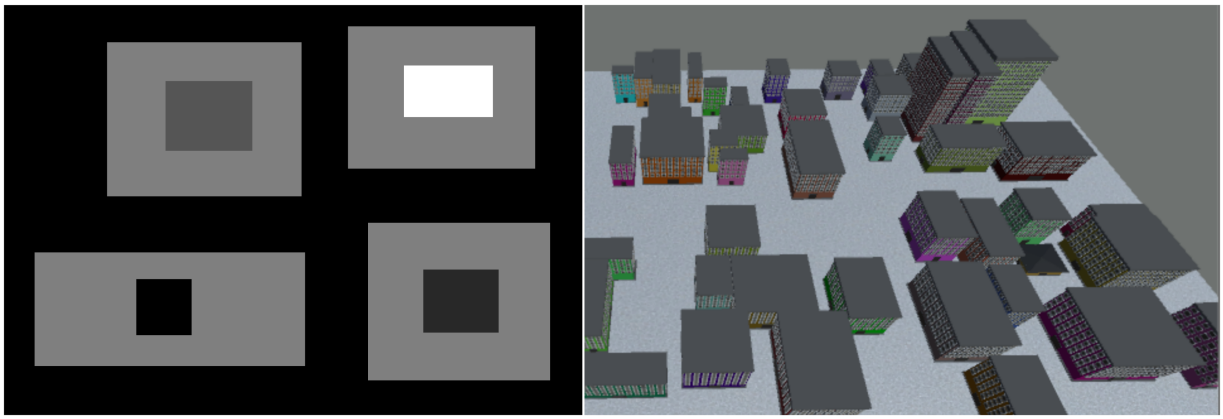

Figure 6.5: Mixed buildings

In Figure 6.6 - left, a slightly different kind of heightmap was provided. Although it also focused on scenes with buildings of mixed heights, the black building-free zones were significantly reduced. Figure 6.6 - right highlights the results. It becomes obvious that the building distribution across the city is more concentrated compared to the formerly mentioned cases, creating narrower roads. As for the building heights, they were computed in a similar fashion as above. The gray shades from the map are accurately reflected in the landscape, with the tallest building in the bottom middle area and the shortest buildings in the bottom left area.

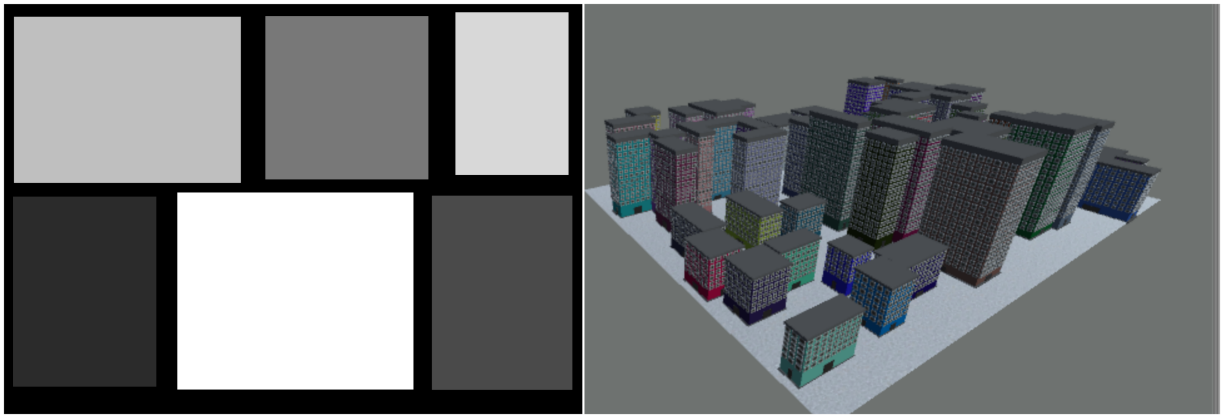

Figure 6.6: Mixed buildings

6.2 Nonfunctional Testing

In contrast to functional testing, nonfunctional testing does not focus on the system requirements, but rather on the quality the system provides. Nonfunctional testing is a fundamental technique when developing an application, determining how it performs under normal circumstances. This type of testing is applied from a user's perspective and is strongly tied to the customer satisfaction, because it examines the system from a performance point of view responsible for assuring a smooth experience.

In order to measure the system's performance at run-time, the profiling tool integrated within the Unity editor has been used. The primary purpose of a profiling tool is to identify how the system behaves in the background. Besides, it provides an easy way by which one can discover the exact source of the performance issues that might occur.

Figure 6.7 illustrates the Central Processing Unit (CPU) usage for a landscape comprising 100 buildings, i.e. how long it took the CPU to complete each task. Each operation is represented by a different color. As it can be seen from the image, the CPU spends the majority of time on rendering operations (the green part). This is a logical and justified behavior, since the current application relies entirely on procedural generation techniques, creating all the data algorithmically.

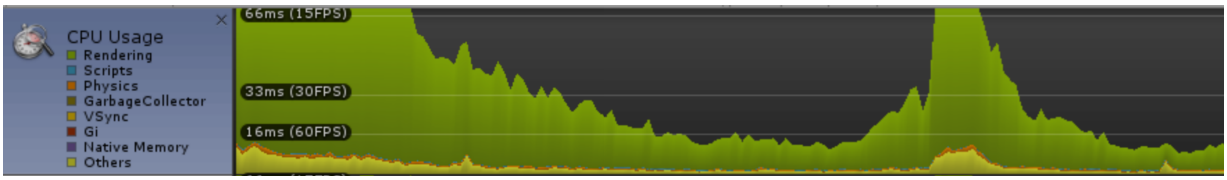

Figure 6.7: CPU usage

Another thing worth noticing is that the Frames Per Second (FPS) increase to 30 FPS - 60 FPS when the users zoom in the scene and navigate through the landscape. All the applications exposing this frame rate run smoothly, without any graphics lag. However, when the users get far away from the generated city and are able to see all the buildings from above, the FPS will decrease to 15 FPS. This might result in an unpleasant experience at times and needs further optimizations. Since the major focus was put on creating an application with a high degree of usability addressed to all types of users irrelevant of their backgrounds, the performance factors came as a second priority. It has to be mentioned though that the CPU usage depicted in Figure 6.7 is not applicable to all cases, being highly dependent on the level of detail of the buildings and ultimately the overall landscape.

Chapter 7

User's Manual

This chapter will provide a helpful guide for the newcomers with all the necessary information one needs to know in order to install the application and run it locally. The first part will elaborate on the hardware and software resources required by the system in order to function properly. Next, a set of step-by-step instructions will be included that explain how to install the application on the local machine. In the last section, a short tutorial on how to use the application from the perspective of an inexperienced user will be described.

7.1 System Requirements

According to [9], the minimum requirements that need to be fulfilled before running a Unity application on Desktop are the following:

- OS: Windows XP SP2+, Mac OS X 10.8+, Ubuntu 12.04+, SteamOS+
- Graphics card: DX9 (shader model 3.0) or DX11 with feature level 9.3 capabilities
- CPU: SSE2 instruction set support

7.2 Installation Steps

The system installation is extremely simple and can be easily achieved by anyone. After the DVD was inserted in the computer, an archive will appear and the next steps must be performed:

1. Extract the files from the .zip archive.
2. Drag and drop the UrbanLandscapes.exe file in the desired location.

As it can be seen, this is a portable application that can be accessed at all times and does not require Internet connection.

7.3 Usage Instructions

Opening this application is as simple as opening any other executable program, by double-clicking it. After opening the UrbanLandscape.exe program, the UI will appear, asking for user input in order to create an urban landscape. Because the purpose of each UI element was described earlier in Chapter 5 - Section 5.4.2, a brief overview will be offered next to sum up the previous explanations.

The recommended workflow for generating an urban landscape is as follows:

1. Open any image editing tool (e.g. Microsoft Paint, Paint.NET).

2. Draw a building heightmap that fulfills the system requirements. The image should be graycale, consist of multiple rectangles and provide an aerial view of the city. Darker-shaded rectangles will produce regions with shorter buildings, whereas lighter-shaded rectangles will result in areas populated by taller buildings. An example of a valid input map is illustrated in Figure 7.1.

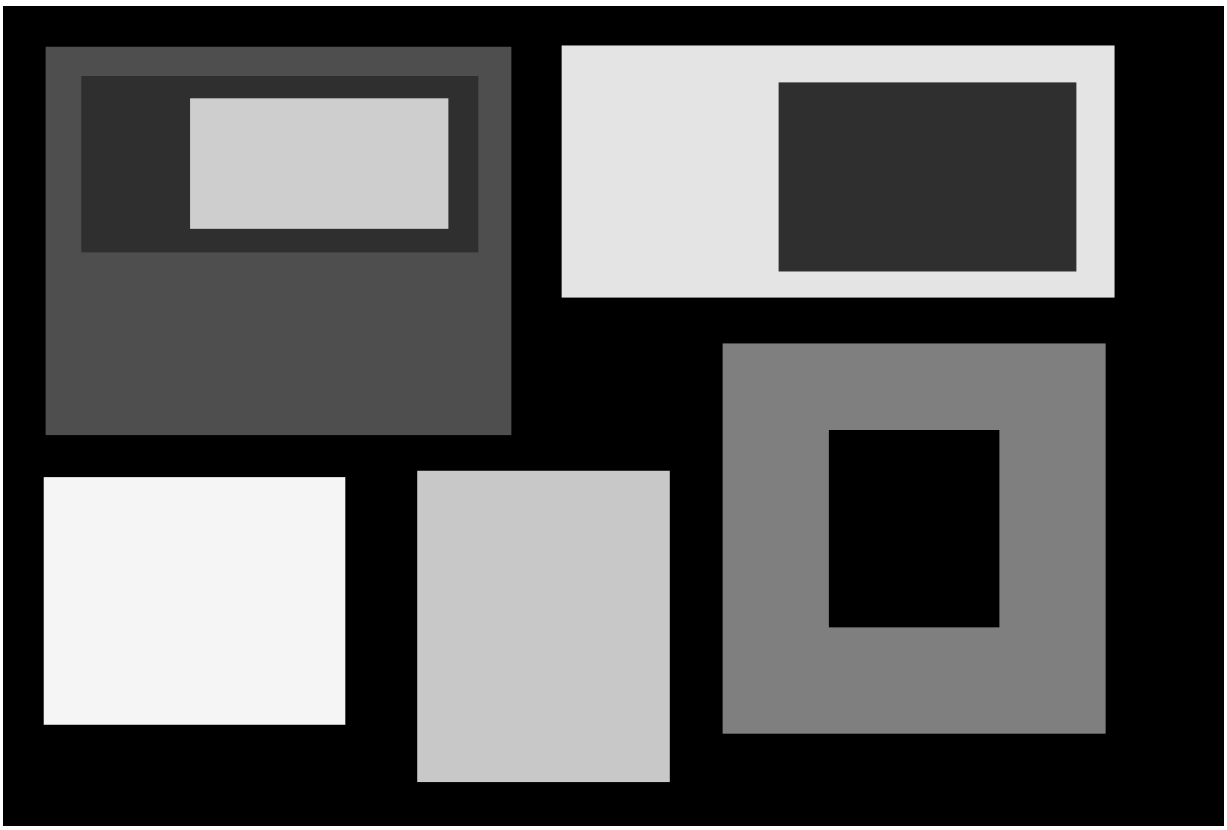

Figure 7.1: Input map

3. Double-click on the UrbanLandscape.exe to open the application. An empty UI similar to the one depicted in Figure 7.2 will appear.

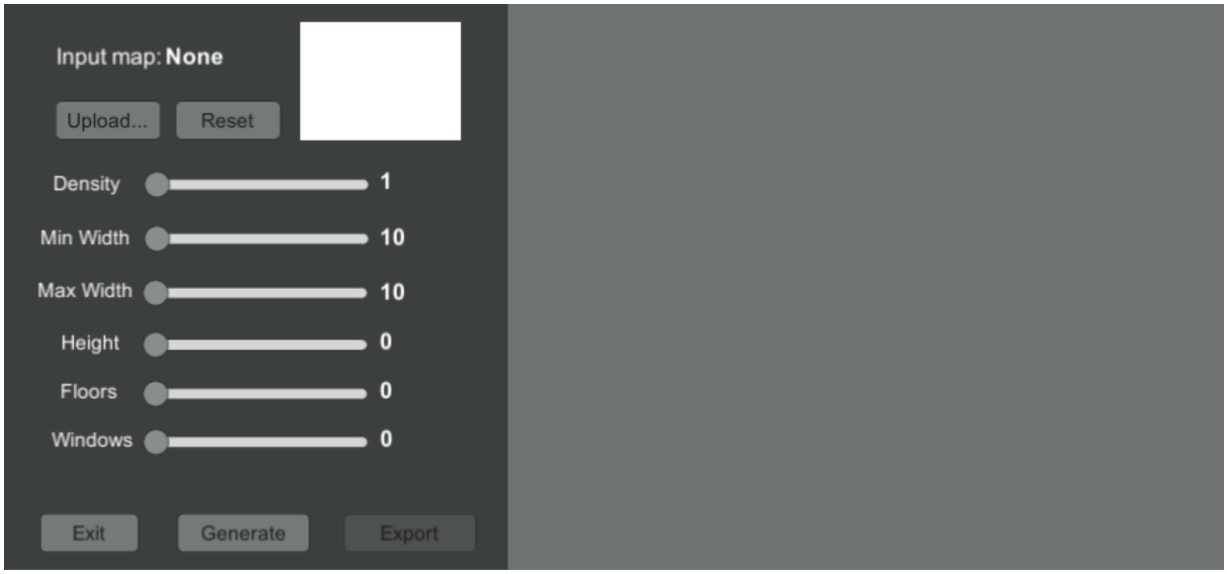

Figure 7.2: Empty UI

4. Click on the ”Upload” button, navigate to the file's location and select the previously drawn image. Figure 7.3 shows the updated UI after the image was successfully loaded into the application.

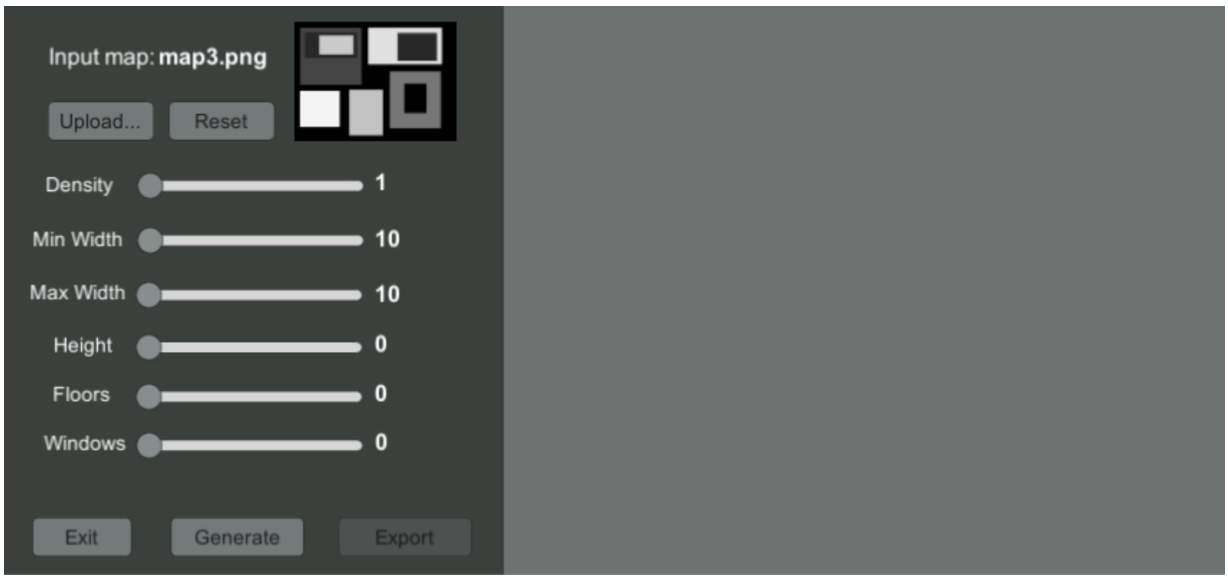

Figure 7.3: Updated UI after uploading an image

5. Adjust the input parameters, taking into consideration their meaning:
   - Density - The number of buildings within the city

- Min Width - The minimum widths of each building
- Max Width - The maximum widths of each building
- Height - The height differences between all buildings
- Floors - The number of floors of each building (lower - more floors, higher - fewer floors)
- Windows - The number of windows on each floor (lower - more windows, higher - fewer windows)

Figure 7.4 presents an example of how can one alter the parameters. As a guideline, the number of buildings to be generated will be close to the square of the density value. Thus, in the below figure, if the density value is 10, this means that maximum 100 buildings can be spawned across the scene. Moving on, Min Width and Max Width are adjusted for generating wider buildings and exclude the narrower ones. Ultimately, the Floors and Windows values indicate a higher number of floors and windows evenly distributed on each building, due to the almost equal values between the two.

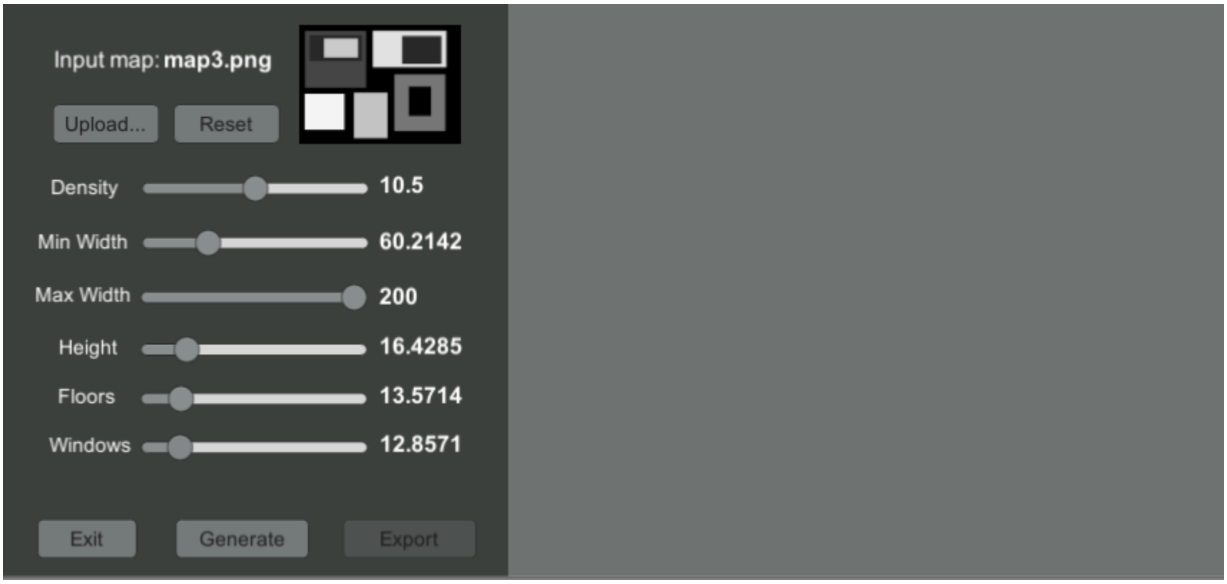

Figure 7.4: Altered parameters

6. Press the ”Generate” button. This will call the procedural generation methods that were implemented and produce a 3D environment based on the provided attributes. Figure 7.5 shows an example of urban landscape that is displayed on screen after the system finishes performing all the required operations. As an additional remark, the first-person camera is initially paused and can be enabled by pressing the Space bar.

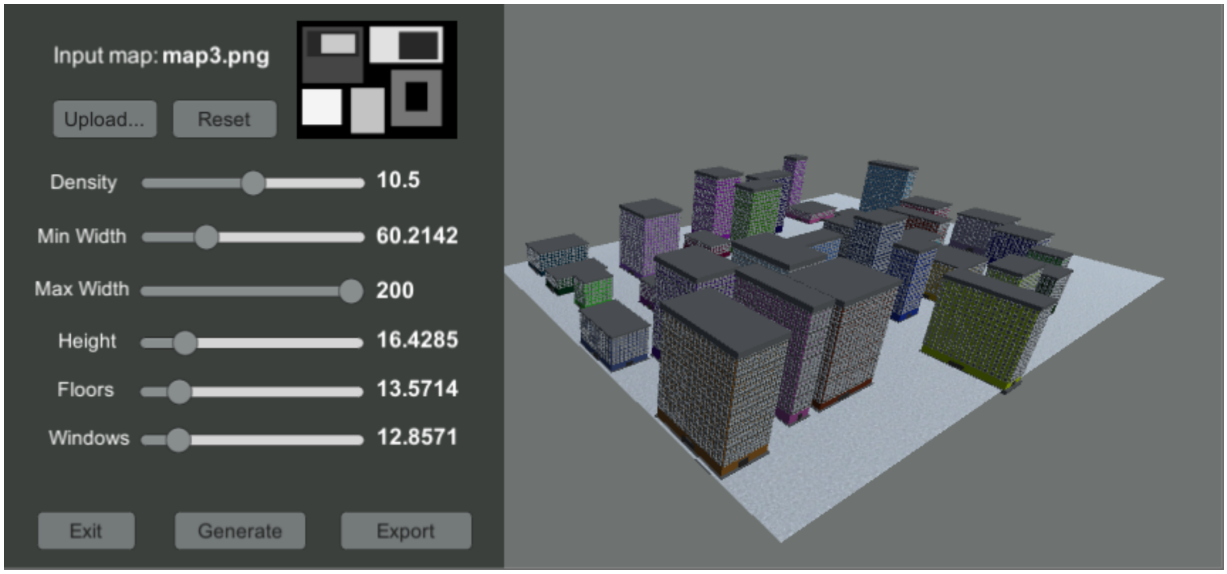

Figure 7.5: Output

7. Explore the scene from various angles by using the W, A, S or D keys for directions and mouse movements for rotating around. Figures 7.6 and 7.7 show some examples of visualizing the resulted scene. As can be observed, the urban landscape is accessible from all angles and the users can examine it in detail irregardless of their distance from the scene.

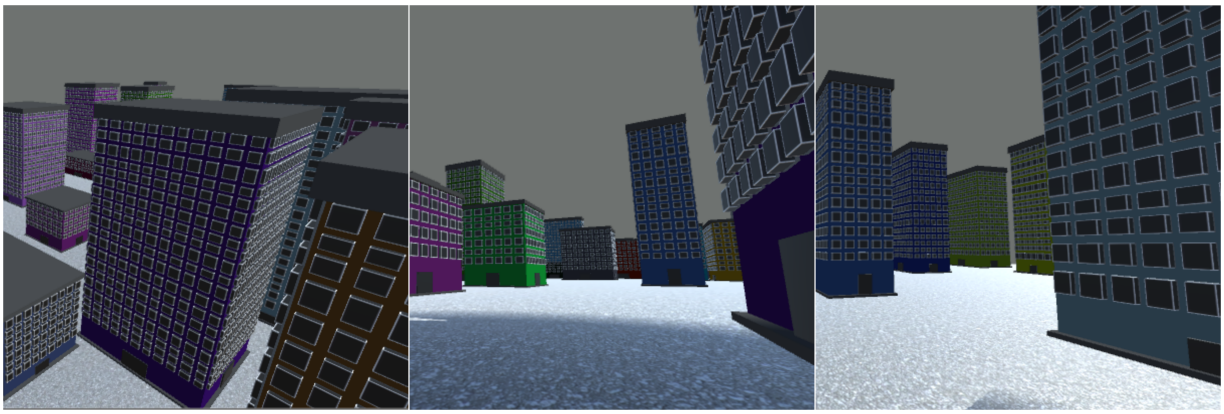

Figure 7.6: First-person perspective (I)

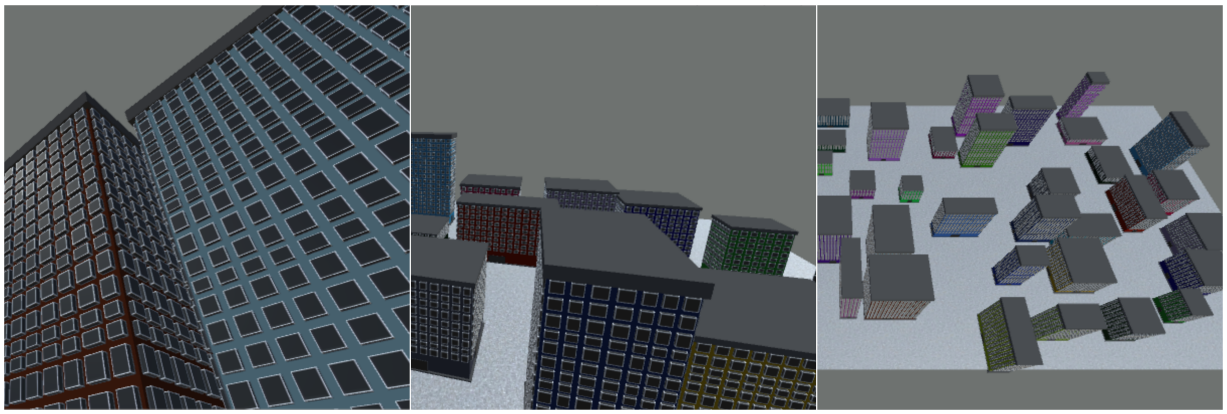

Figure 7.7: First-person perspective (II)

8. After generating procedural content, there are three options available:
   - Export the 3D model in .obj format by pressing the Export button, selecting the path and introducing the file name in the text box.
   - Re-generate a new scene. For removing the current scene displayed on screen, one needs to click on the Reset button. The application will return to its initial state depicted in Figure 7.2.
   - Close the application by pressing the Exit button.

Chapter 8

Conclusions

8.1 Contributions and Achievements

By developing this application, an accessible approach for generating user-guided landscapes was introduced. Compared to the other existing techniques, the system is aimed to a larger community and does not require any prior knowledge in order to use it. The algorithm that was implemented prioritizes the users' wishes and offers them full control over the results, unlike other applications that rely on randomness and do not allow the full customization of 3D content before being created.

8.2 Obtained Results

On the whole, the three main objectives of this paper were accomplished.

The most difficult task involved the implementation of the first objective, namely the procedural modeling of buildings. Although the topic has been extensively studied over the past few years, the majority of the techniques concentrated on individual buildings and did not consider how they could be applied in the context of large or medium-sized cities. This project started from this very assumption and as a result, a minimalistic algorithm was conceived that produces unique lightweight models without affecting the rendering performance. The key feature that this algorithm exposes is that it does not depend on complex inputs which are difficult to provide by the average user. Instead of obliging the newcomers to learn the syntax of a completely unknown language, they are allowed to describe the building geometry in natural language directly from the UI. By employing this method, the ambiguities that might occur before generating 3D content are completely removed. Besides, in the context of largely-scaled environments which do not focus on the level of detail, generating individual buildings should be as fast as possible.

The second objective was represented by the procedural generation of urban landscapes, which is also the primary goal within this paper. The landscapes were successfully populated with multiple buildings uniformly distributed at random positions across the

scene and the buildings' heights were computed based on the grayscale values depicted in the map. The most challenging part during this stage consisted in solving the overlapping problem. Fortunately though, a solution based on bounding boxes was found, solving this particular issue. Moreover, the city generation algorithm was constantly optimized to improve the response times.

As a last objective, a Desktop application incorporating the above functionalities was built. An UI was added to the application, useful for guiding the algorithm towards the landscape generation, along with a controller responsible for handling the user input.

8.3 Further Development

The system can benefit from further improvements such as:

- Increased scalability - The algorithm used can be extended to a much wider extent to produce infinitely-sized cities. However, generating huge scenes is an extremely demanding process and possible workarounds have to be considered. For instance, instead of creating hundreds of unique buildings at run-time, building models could be saved in memory beforehand and accessed during the city generation, to avoid prolonged waiting times. Another alternative would consist in generating fewer buildings and cloning them repeatedly until they fill up the entire landscape.

- Street network - Another useful upgrade would be adding a street network in the virtual city, as drawn by the user in the input map.

- Complex buildings - The buildings' geometry can be improved through addition and subtraction of shapes.

Bibliography

[1] P. Prusinkiewicz and A. Lindenmayer, *The algorithmic beauty of plants*. New York: Springer-Verlag, 1990.

[2] P. Wonka, M. Wimmer, F. Sillion, and W. Ribarsky, "Instant architecture," *ACM Transaction on Graphics*, vol. 22, no. 3, pp. 669–677, Jul. 2003, proceedings ACM SIGGRAPH 2003. [Online]. Available: https://www.cg.tuwien.ac.at/research/publications/2003/Wonka-2003-Ins/

[3] G. Stiny, "Introduction to shape and shape grammars," *Environment and planning B*, vol. 7, no. 3, pp. 343–351, 1980.

[4] P. Müller, P. Wonka, S. Haegler, A. Ulmer, and L. J. V. Gool, "Procedural modeling of buildings." *ACM Trans. Graph.*, vol. 25, no. 3, pp. 614–623, 2006. [Online]. Available: http://dblp.uni-trier.de/db/journals/tog/tog25.html#MullerWHUG06

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

[6] P. Müller, G. Zeng, P. Wonka, and L. J. V. Gool, "Image-based procedural modeling of facades." *ACM Trans. Graph.*, vol. 26, no. 3, p. 85, 2007. [Online]. Available: http://dblp.uni-trier.de/db/journals/tog/tog26.html#MullerZWG07

[7] H. M. George Kelly, "A survey of procedural techniques for city generation," *ITB Journal*, 2006.

[8] P. Müllller, "Procedural modeling of cities." in *SIGGRAPH Courses*, J. W. Finnegan and D. Shreiner, Eds. ACM, 2006, pp. 139–184. [Online]. Available: http://dblp.uni-trier.de/db/conf/siggraph/siggraph2006courses.html#Muller06

[9] Unity user manual. [Online]. Available: https://docs.unity3d.com/Manual/index.html