Bachelor Project

Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

Direct rendering of procedural models

Alexander Temnyakov

Supervisor(s): Jiri Bittner, Ph.D.; Bedrich Benes, Ph.D
May 2021

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Temnyakov** Jméno: **Alexander** Osobní číslo: **483736**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačové grafiky a interakce**

Studijní program: **Otevřená informatika**

Specializace: **Počítačové hry a grafika**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Přímé zobrazování procedurálních modelů

Název bakalářské práce anglicky:

Direct Rendering of Procedural Models

Pokyny pro vypracování:

Zmapujte metody přímého zobrazování procedurálních modelů pomocí metody sledování paprsků a případné dostupné implementace. Implementujte přímé zobrazování procedurálně generovaného modelu města v reálném čase. Zaměřte se na možnost zpracování nejen procedurálně generované geometrie, ale i procedurálně generovaných světelných zdrojů a textur. Procedurální generování města bude řízeno daty reálných modelů měst, která jsou dostupná v rámci projektu Open Street Map. Implementaci realizujte v C++ s využitím rozhraní OptiX pro vrhání paprsků. Výslednou implementaci důkladně otestujte z hlediska rychlosti zobrazování pro různé velikosti generovaných scén a typu osvětlení (denní vs noční scéna, počet světelných zdrojů). Vyhodnoťte kvalitu zobrazování denních a nočních měst a porovnejte výsledné obrázky s fotografiemi.

Seznam doporučené literatury:

[1] Smelik, Ruben M., et al. 'A survey on procedural modelling for virtual worlds.' Computer Graphics Forum. Vol. 33. No. 6. 2014.
[2] Steinberger, Markus, et al. 'On-the-fly generation and rendering of infinite cities on the GPU.' Computer graphics forum. Vol. 33. No. 2. 2014.
[3] Schwarz, Michael, and Pascal Müller. 'Advanced procedural modeling of architecture.' ACM Transactions on Graphics (TOG) 34.4 (2015): 1-12.
[4] Nishida, Gen, et al. 'Interactive sketching of urban procedural models.' ACM Transactions on Graphics (TOG) 35.4 (2016): 1-11.
[5] Merrell, Paul, and Dinesh Manocha. 'Model synthesis: A general procedural modeling algorithm.' IEEE transactions on visualization and computer graphics 17.6 (2010): 715-728.
[6] Wald, Ingo, et al. 'RTX Beyond Ray Tracing: Exploring the Use of Hardware Ray Tracing Cores for Tet-Mesh Point Location.' High Performance Graphics (Short Papers). 2019.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

doc. Ing. Jiří Bittner, Ph.D., Katedra počítačové grafiky a interakce

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **11.02.2021** Termín odevzdání bakalářské práce: _____

Platnost zadání bakalářské práce: **30.09.2022**

doc. Ing. Jiří Bittner, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

Datum převzetí zadání

Podpis studenta

Acknowledgements

I am deeply grateful to Jiri Bittner, Ph.D. and Bedrich Benes, Ph.D. for their guidance, valuable advice, and feedback throughout this project. At the beginning, it seemed to be impossible to develop. But thanks to them, I did it.

Declaration

I declare that this work is all my own work and I have cited in the bibliography all sources I have used.

Prague, May 21, 2021

Prohlašuji, že jsem předloženou práci vypracoval samostatně, a že jsem uvedl veškerou použitou literaturu a zdroje. V

Praze, 21. května 2021

Abstract

Projects in the field of 3D graphics may be focused on rendering large scenes with a big number of objects. A human is not able to create all objects manually in a reasonable amount of time. To avoid such a problem, the objects may be created by a program instead of the human.

In 3D applications, the polygons of 3D models usually are stored in the memory. This approach consumes it much. To reduce memory consumption, the polygons may be created procedurally at run time on demand instead of storing them in the memory.

When using the path tracing algorithm, light sources may be a problem. They can be small, so the rays almost never hit them. They can be big, so they have a redundant influence on the environment.

The target of this work is to develop an algorithm of rendering with a nontraditional geometry representation in which the polygons are not stored in the memory; to develop an algorithm of procedural object creation, to implement a rendering application that uses this geometry representation and also focuses on light sources and to evaluate the algorithm and the implementation by comparison with the traditional geometry representation.

Keywords: computer graphics, ray tracing, path tracing, OptiX, procedural model generation, procedural cities, OpenStreetMap, light sources, lighting

Supervisor(s): Jiri Bittner, Ph.D.; Bedrich Benes, Ph.D

Abstrakt

Projekty ve oboru 3D grafiky mohou být zaměřeny na renderování obrovských scén s velkým počtem objektů. Člověk není schopen je vytvořit manuálně za krátkou dobu. Pro řešení tohoto problému objekty mohou být vytvořeny programem místo člověka.

V 3D aplikacích polygony 3D modelů obvykle jsou uloženy v paměti. Takový přístup ji spotřebuje ve velkém množství. Pro zmenšení spotřeby paměti polygony mohou být vytvořeny programem za běhu na požádání místo jejich uložení do paměti.

Při použití metody sledování cest světelné zdroje mohou být problémem. Mohou být malé, proto je paprsky skoro nikdy nezasahují. Mohou být velké, proto mají zbytečný vliv na okolí.

Cílem této práce je vymyslet algoritmus renderování s netradiční reprezentací geometrie, při které se polygony neukládají do paměti; vymyslet algoritmus procedurálního vytvoření 3D modelů, naimplementovat aplikaci, která používá takovou reprezentaci geometrie a také klade důraz na světelné zdroje; a vyhodnotit algoritmus a implementaci porovnáním s tradičním způsobem reprezentace geometrie.

Klíčová slova: počítačová grafika, metoda sledování paprsků, metoda sledování cest, OptiX, procedurální generování modelů, procedurální města, OpenStreetMaps, světelné zdroje, osvětlení

Překlad názvu: Přímé zobrazování procedurálních modelů

Contents

Figures

Tables

Chapter 1

Introduction

Sometimes scenes in graphics applications consist of a very large number of objects. That means it is very difficult to create them manually. This problem may be solved by procedural object generation. That is, the program creates the objects instead of a human. It may create any number of objects fast in comparison to the human. This approach may save much time of work, even years. If a scene has many objects, they require much memory and a big number of data movements to the graphics card. With such an approach to object creation, the generated geometry does not have to be stored in the memory. It may be created on demand[1] for each frame[2]. It is a very efficient way to reduce the amount of used memory. There is a question: how to determine if an object's geometry has to be created. The ray tracing algorithm may answer this. However, it is likely that the process of geometry creation on demand for each frame requires much time. Thus, it significantly increases the rendering time of the frame. One of the most important targets of this project is to measure the difference in memory consumption and the rendering time between procedural object creation on demand and storing objects, which were created procedurally, in the memory.

For such an approach to object representation, I developed a rendering algorithm and a rendering application for it. The rendering algorithm is a variant of path tracing[3]. It is able to simulate various types of materials and it uses special techniques for light sources in the scene to make the image more realistic and attractive. The primary functionality - rendering using ray tracing - is implemented by using OptiX Ray Tracing Engine by Nvidia[4]. The rest is written in C++ using various libraries. Also, it uses some utilization functions and things provided with that API.

Because the geometry can not be created out of thin air, the objects require some data which is used as the cornerstone for their geometry. For this purpose, I developed a program. It was chosen that the scene would be

[1] If an object influences what the user sees on the screen, its geometry is created. Otherwise, the geometry is not created.

[2] When the program calls the function which renders a frame, the program creates the geometry for the objects and it is not stored in the memory.

[3] Path tracing is a type of ray tracing.

[4] See [27].

a city. For this reason, the program uses OpenStreetMaps[5] to prepare the required data. The program is written in C#, it uses OsmSharp library[6] to process the OpenStreetMap data and some other libraries for other purposes, such as mathematical operations. Also, it uses Unity[7] as a graphical user interface (GUI).

This report describes how the developed rendering algorithm works, how the OpenStreetMap data looks like, how to use it to create a model of a city, how to use the rendering algorithm with the created objects to get beautiful and realistic pictures. It provides measurements of the procedural geometry representation, compares it with the traditional one, and shows visuals of the project. As a conclusion, it discusses if object creation on demand for each frame is useful, if it actually solves the problems, and what own problems it has.

[5] See [4].
[6] See [5].
[7] See [7].

Chapter 2

Rendering

This chapter provides an introduction to ray tracing and path tracing and describes how the developed rendering algorithm works, what materials it has, and how these materials are simulated.

2.1 Ray Tracing

Ray tracing is an algorithm of rendering which produces realistic effects: reflections, refractions, and shadows. Rendered images using ray tracing are more natural than images rendered using rasterization.[1] A visual comparison is shown in figure 2.1. The essence of it is rays that are cast from the camera into the scene. This technique is called *ray casting*. They find the closest object in the scene to the point from which they are cast. In the beginning, a ray, which is called the *primary ray*, is cast from the camera into the scene. The algorithm finds the closest object on the way of this ray. Therefore, the camera sees the hit point on the object. The pixel should be influenced by the color of this point. To check if the hit point on the object is occluded[2] by another object in the scene, another ray, which is called the *shadow ray*, is cast from this point towards a light source. If it hits an object on its way, this point is occluded. If it does not, this point is not occluded. As you can see, the algorithm is simple for such things. The primary and the shadow rays are visualized in figure 2.2. The article *Ray Tracing*[3] by Nvidia provides a brief introduction to ray tracing and its basics.

Ray tracing may be combined with a shading model used for rasterization. For example, with the Phong shading model. In this case, the same calculations are made for each intersection point of a ray and an object as for each vertex or fragment when using OpenGL shaders. However, ray tracing allows an easier way to render shadows: the use of the shadow ray is simpler and more precise than the creation of the shadow map.[4]

[1] See [29].
[2] That is, it is in a shadow.
[3] See [28].
[4] See [13, Shadow Mapping].

Modern personal computers are enough powerful to make all calculations for this algorithm in a short time and ensure the use of such programs in real time.

There is a great online book named *Scratchapixel - Learn Computer Graphics From Scratch!*[5] that explains the ray tracing algorithm. Also, there is a bachelor project related to this topic *Real-Time Ray Tracing in Unreal Engine*[6] by Vojtěch Vavera which discusses the implementation of the ray tracing algorithm.

(a) : Screenshot 1.

(b) : Screenshot 2.

Figure 2.1: Figure showing a visual comparison of ray tracing and rasterization. In both the screenshots, the left part demonstrates an image with ray tracing, the right part shows an image with raterization.[7]

[5] See [6].
[6] See [46].
[7] Taken from [40].

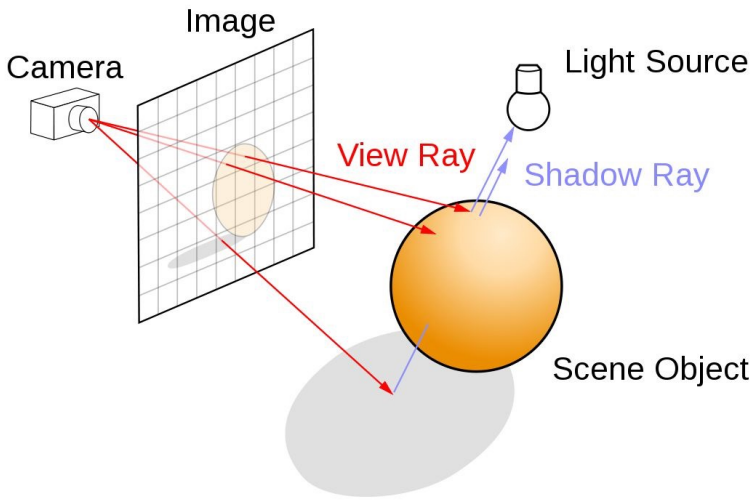


Figure 2.2: Radiance and shadow rays.[8]

2.2 Path Tracing

Path tracing is a type of ray tracing. It simulates more effects than simple ray tracing and produces a more realistic image. In this algorithm, a ray is reflected from object surfaces and flies in the scene until some conditions are met. The algorithm is based on the determination of how much of the illuminance will go towards the camera. In simple words, the ray flies and takes the colors of hit objects, adds and multiplies them. If it hits the skydome in the scene, it also takes its color.

There is an equation called the *rendering equation* which is the essence of path tracing. James Kajiya describes it in the article *The Rendering Equation*.[9] Also, Victor Li briefly and understandably explains it in the article *Raytracing - Rendering Equation Insight*.[10]

$$L_o(\omega_o) = L_e(\omega_o) + \int_\Omega f(\omega_i, \omega_o) L_i(\omega_i) (\omega_i \cdot n) \, \mathrm{d}\omega_i.$$

L_o is the outgoing light. It is the sum of the emitted light L_e and the reflected light L_r. The reflected light L_r is the sum of the incoming light L_i multiplied by a function that describes the physical properties of the material at the current point.

There are many conditions for how to decide if the ray has to be terminated. It may be terminated if the depth - the number of how many times a ray was reflected - equals to a selected value, it may be terminated with some probability or if it flies out of the scene.

[8] Taken from [28].
[9] See [16].
[10] See [23].

The direction of ray reflection from the object surfaces may be random.[11] As a consequence, the algorithm produces noisy images. There are two easy ways to prevent that. First, an implementation may cast a big number of rays from each pixel and take the average result of them. The bigger this number is, the more likely that the average values converge to the same one. This solution requires powerful hardware. Second, the implementation may save each frame in a buffer and when it renders a new frame, it interpolates the color of each pixel between the color from the previous frame stored in the buffer and the color of the current frame.[12] This solution works if nothing in the scene changes, but it is fast and it doesn't increase the rendering time.

A simple implementation of path tracing does not require many lines of code. It can fit in 99 lines[13]. The book *Scratchapixel - Learn Computer Graphics From Scratch!* describes path tracing[14]. Also, the book *Physically Based Rendering: From Theory to Implementation*[15] is a great resource to study the path tracing algorithm.

2.3 Nvidia OptiX Ray Tracing Engine

Nvidia OptiX Ray Tracing Engine[16] is a modern high-level ray tracing API. It allows the programmer to implement a small part of ray tracing instead of the implementation of the whole algorithm and structures for it. By the small part, it is meant the programmer has to store the primitives of objects in the scene or to store data required to procedurally create them, to store material data for the objects, to implement a function named *raygen program* which casts rays from the camera, a function named *closest program* which is called when a ray hits an object, a function *miss program* which is called when the ray hits the skydome. They do not have to implement acceleration structures and to detect ray-primitive intersections if the polygons are stored in the memory. They have to detect intersections if the objects are created procedurally. In this case, each of them is bounded by an axis-aligned bounding box. If the ray hits a bounding box, the API calls a function named *intersection program* which creates the geometry of the object and detects an intersection. Nvidia provides an article named *OptiX: A General Purpose Ray Tracing Engine* on this API, which helps to understand it better.

The API has been still being developed. The version used in this project is 7.2.0. The previous versions 6.x.x and less completely differ from 7.2.0. If a project is written in one of the previous versions, it can not be easily upgraded to use version 7.x.x or greater. It is required to fully rewrite the

[11] For example, it is random for a diffuse material such as rubber, but it is not random for mirror.
[12] It is explained in section 2.4.5.
[13] See [9].
[14] See [6, Global Illumination and Path Tracing].
[15] See [35, ch. 14, p. 5].
[16] See [27].
[16] See [32].

project. For this reason, there is a lack of open-source projects which may be used to study the API. Besides the sample projects by Nvidia, Ingo Wald provides a great tutorial on this API.[17]

Nvidia provides a denoiser[18]. It is AI-accelerated. So, it is an efficient and fast way to eliminate noise in comparison with other ways of denoising[19].

There is various software using the API. The most popular of them are Blender[20], Adobe After Effects CC[21] and OctaneRender.[22]

2.4 Developed Rendering Algorithm

The rendering algorithm in this project is a variant of path tracing. Rays are cast from the camera and they reflect from the object surfaces in the scene until some conditions, which will be described later, are met. The objects have their diffuse and emissive colors. The pseudocode shown in algorithm 1 describes how the calculation of the color for a pixel works. In this project, the color of the pixel is normalized. That is, it is in form $[x, y, z]$, where $x, y, z \in [0, 1]$.

Algorithm 1 The function to compute the radiance along a ray.

Input: the ray's origin O and its direction D.

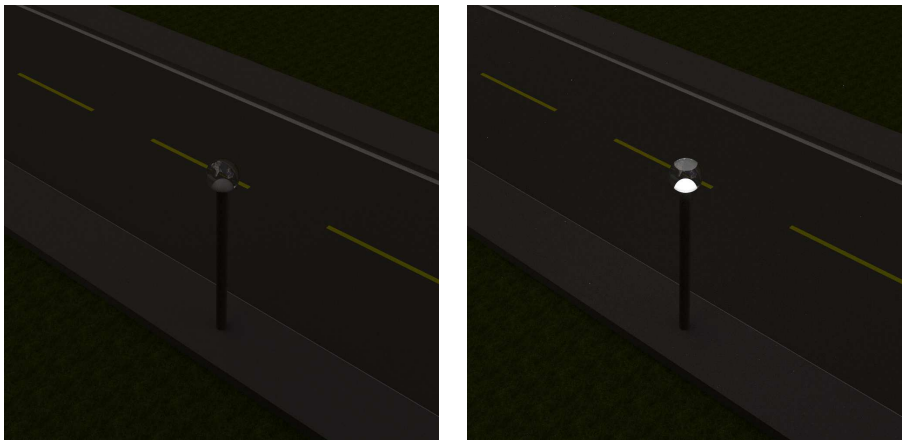
```
1:  function RADIANCE(O, D)
2:      if CLOSEST_HIT(O, D) then
3:          P = GET_INTERSECTION_POINT()
4:          terminate, c = RUSSIAN_ROULETTE()
5:          if terminate then
6:              return emissive_color * c
7:          new_D = CALCULATE_DIRECTION()
8:          D = new_D
9:          return emissive_color + diffuse_color * RADIANCE(P, D)
10:     else
11:         return DETERMINE_SKYDOME_COLOR()
```

The parts of this pseudocode will be explained later. Because OptiX is not friendly with recursive functions and to use such functions is undesirable, the program mainly uses a non-recursive variant of this algorithm. However, in some situations, recursion is used.[23]

Let us explain what is the emissive color and the diffuse color. The emissive color is responsible for the light properties of the material. Light

[17] See [47].
[18] See [26].
[19] Further, subsection 2.4.5 discusses some ways of denoising.
[20] See [2].
[21] `https://www.adobe.com/products/aftereffects.html`
[22] See [30].
[23] Recursion is used to reflect two rays from the glass material. It is more convenient than the non-recursive method.

sources have this color greater than zero, other objects have this color equal to zero. Figure 2.3 shows what is the emissive color. To explain what is the diffuse color of an object, let us look at an example. When you are talking about the color of something, often you are talking about the diffuse color. For example, if an apple is green, its diffuse color is green. However, we cannot determine what diffuse color metals have, because they reflect everything around.



(a) : The emissive color is equal to 0.

(b) : The emissive color is greater than 0.

Figure 2.3: Figure visualizing an object with a different emissive color.

It is necessary to introduce 2 terms. **The ray has a direct influence on a pixel**

- if it was cast from the camera;
- if it was reflected from glass or mirror[24] and the previous ray was cast from the camera or it was also reflected from glass or mirror.

In the other case, that is, **the ray has an indirect influence on a pixel**

- if it was reflected from an object which has a material other than glass or mirror.

2.4.1 Skydome

Path tracing uses the skydome to light the scene. The skydome is considered a light source and there is no directional light. The skydome has its emissive and diffuse colors like the other scene objects. As you learned above, the emissive color of a light source is a nonzero vector.

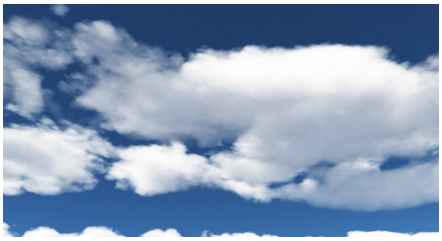
[24] The materials are discussed further in subsection 2.4.3

However, in this implementation, the emissive color of the skydome is not used in a situation. If the ray has a direct influence on a pixel, only the skydome's diffuse color is used. If the emissive color of the skydome was also used, we would see an incorrect color on the screen, because according to the rendering algorithm, the diffuse color has to be added to the emissive color. If the emissive color is $[1, 1, 1]$ and it is summed with any diffuse color, it results in the situation when the skydome has a uniform white color.[25] [26] If the ray has an indirect influence on a pixel, both the diffuse color and the emissive are used.

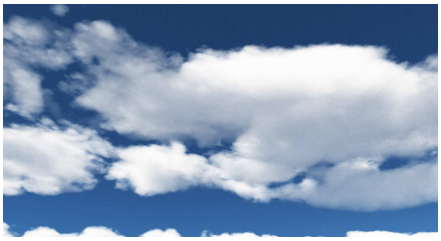
Algorithm 2 The function to determine the color of the skydome.

```
1: function DETERMINE_SKYDOME_COLOR
2:     if ray_has_direct_influence_on_pixel_color then
3:         return diffuse_color
4:     else
5:         return emissive_color + diffuse_color
```

Figure 2.4 shows how the sydome would look if its emissive color was used always and there was not the *if-else* statement. According to algorithm 2, if it always returned *emissive_color + diffuse_color*, the returned value would be close to $[1, 1, 1]$ or even it would exceed this value.



(a) : Emissive color $[0, 0, 0]$.



(b) : Emissive color $[0.5, 0.5, 0.5]$.

(c) : Emissive color $[0.7, 0.7, 0.7]$.

(d) : Emissive color $[1, 1, 1]$.

Figure 2.4: Figure demonstrating the use of the skydome color when the ray has a direct influence.

[25] $[1, 1, 1]$ is white. Each value is clamped between $[0, 0, 0]$ and $[1, 1, 1]$.
[26] $[1, 1, 1] + [x, y, z] \geq [1, 1, 1]$. The result is always white.

Figure 2.5 shows visuals of algorithm 2. The skydome has the same color. However, the scene is more or less bright, because the used emissive colors are different.

(a) : Emissive color $[0, 0, 0]$.

(b) : Emissive color $[0.5, 0.5, 0.5]$.

(c) : Emissive color $[0.7, 0.7, 0.7]$.

(d) : Emissive color $[1.0, 1.0, 1.0]$.

Figure 2.5: Figure demonstrating the algorithm of the skydome color use.

2.4.2 Ray Reflection

If the ray hits an object, it is reflected in some direction. This direction is calculated by one of the following methods.

Specular Sampling

The ray is reflected by the rule of specular reflection[27].

Diffuse Sampling

It is also called cosine weighted hemisphere sampling. The main idea of this sampling is that the calculated direction is close to the normal vector of a point. Calculation of this vector d is the following[28]:

[27] See [45] or `https://en.wikipedia.org/wiki/Specular_reflection`.

[28] In this work, [1, 0, 0] is the left direction, [0, 1, 0] is the up and [0, 0, 1] the forward.

$$\theta = \arccos(\sqrt{u_1}), \tag{2.1}$$

$$\phi = 2 \cdot \pi \cdot u_2, \tag{2.2}$$

$$d = [\cos(\phi) \cdot \sin(\theta),\ \cos(\theta), \sin(\phi) \cdot \cos(\theta)], \tag{2.3}$$

$$u_1 \in [0, 1],\quad u_2 \in [0, 1]. \tag{2.4}$$

u_1 and u_2 are random numbers from the specified interval. The line 2.3is the transformation from the spherical coordinate system.

Glossy Sampling

The idea is taken from the article *Importance Sampling of the Phong Reflectance Model*[29]. Calculation of the direction is the following:

$$\theta = \arccos(u_1^{\frac{1}{1+n}}), \tag{2.5}$$

$$\phi = 2 \cdot \pi \cdot u_2, \tag{2.6}$$

$$d = [\cos(\phi) \cdot \sin(\theta),\ \cos(\theta),\ \sin(\phi) \cdot \cos(\theta)], \tag{2.7}$$

$$u_1 \in [0, 1],\quad u_2 \in [0, 1]. \tag{2.8}$$

Again, u_1 and u_2 are random numbers from the specified interval. Line 2.7 is the transformation from the spherical coordinate system. n is the specular exponent. The higher the specular exponent is, the sharper the reflection is.

2.4.3 Materials

The material type determines what sampling is used for ray reflection. There are 4 types of materials in this project: diffuse, glossy, specular, and glass.

Diffuse Material

Diffuse sampling is used to determine the direction of the reflection.

Glossy Material

This type of material is similar to the diffuse material, but it is slightly specular. That is, it reflects the silhouettes and colors of objects around. The principle of ray reflection for this material is taken from the article *Importance Sampling of the Phong Reflectance Model*[30]. Shortly, there are two parameters: k_d and k_s. They meet the condition $k_d + k_d \leq 1$. A random number $r \in [0, 1]$ is generated. If r is less than k_d, the direction is calculated by using diffuse sampling. If r is less than $k_d + k_s$, the direction is calculated by using glossy sampling. Otherwise, the ray is not reflected, it is terminated.

[29] See [21].
[30] See [21].

The diffuse material may be treated as a variant of this material with $k_s = 0$ and $k_d = 1$.

Algorithm 3 Pseudocode for the glossy material.

```
1: r = RANDOM_FLOAT.RANGE(0, 1)
2: new_direction
3: if k_d < r then
4:     new_direction = DIFFUSE_SAMPLING()
5: else if r < k_d + k_s then
6:     new_direction = GLOSSY_SAMPLING()
7: else
8:     TERMINATE_RAY()
```

Specular Material

In other words, it is mirror. Specular sampling is used to determine the direction of the reflection.

Glass Material

When we look at glass, we see in it objects which are behind and in front. Glass reflects and refracts all incoming rays. Therefore, if a ray hits glass, two rays are cast from the hit point: a ray in the direction defined by refraction and a ray in the direction defined by ideal reflection. Then, the colors from these two rays[31] have to be summed with some weights:

$$color = r \cdot refraction\_ray\_color + (1 - r) \cdot reflection\_ray\_color.$$

Parameter r is defined by Schlick's approximation. The articles *An Inexpensive BRDF Model for Physically-based Rendering*[32] and *Background: Physics and Math of Shading*[33] explain this formula.

In terms of efficiency, it is not good to cast these two rays always. It is more efficient to cast either the refraction ray or the reflection ray in a situation. The rendering application randomly chooses which of the two rays to cast. The probability is calculated as[34]

$$probability\_of\_reflection = 0.25 + 0.5 \cdot r.$$

If a ray that hits glass has a direct influence on a pixel's color[35], both the rays have to be cast to get a more precise and realistic result and to reduce error. Then, the two rays which are cast from the hit point on glass also have a direct influence on the pixel's color. If the ray does not have a direct influence or the recursion depth is too big, the error is negligible. Therefore, in these cases, it is enough to cast a single ray from the hit point.

[31] That is, the results of the path tracing algorithm started at this point with these directions.
[32] See [37].
[33] See [15].
[34] Taken from [9].
[35] Look at 2.4 to remember what it means.

Algorithm 4 Pseudocode for the glass material.

```
1:  r = 1
2:  P = intersection_point
3:  reflection_dir = REFLECT()
4:  refracted = false
5:  if refracted then
6:      r = FRESNEL_SCHLINK()
7:      refraction_dir = REFRACT()
8:  if ray_has_dir_influence and recursion_depth < g_depth then
9:      refraction_color = glass_diffuse_color
10:     reflection_color = glass_diffuse_color
11:     if refracted then
12:         D = refraction_dir
13:         refraction_color = 1 − r * RADIANCE(P, D)
14:     D = reflection_dir
15:     reflection_color = r * RADIANCE(P, D)
16:     color = refraction_color + reflection_color
17: else
18:     reflection_proba = 0.25 + 0.5 ∗ r
19:     refraction_proba = 1 − reflection_proba
20:     rnd = RANDOM_FLOAT.RANGE(0, 1)
21:     if refracted AND rnd < refraction_proba then
22:         D = refraction_dir
23:         color = 1 − r * RADIANCE(P, D)
24:     else
25:         D = reflection_dir
26:         reflection_color = r * RADIANCE(P, D)
```

2.4.4 Ray Termination

Rays can not fly eternally. Therefore, each of them must be terminated at some time. The project uses a method called *Russian Roulette*, which is described in the book *Physically Based Rendering: From Theory to Implementation*[36], to terminate rays. There is a selected depth d, from which Russian Roulette is carried out, a value c[37] and a termination probability p. If the depth of a ray is greater or equal to d, a random number $r \in [0, 1]$ is generated. If r is less or equal to p, the ray is terminated, it returns the emissive color of the hit object and color $[c, c, c]$ instead of the diffuse color of the object. Otherwise, it modifies the diffuse color by some rule. These steps are shown in algorithm 5.

[36] See [35, ch. 13, p. 7].
[37] In this project, it is equal to 0.

Algorithm 5 The function for Russian Roulette.

Output: whether the ray was terminated or not.

```
1:  function RUSSIAN_ROULETTE()
2:      if depth < d then
3:          return false
4:      else
5:          c = 0
6:          r = RANDOM_FLOAT.RANGE(0, 1)
7:          if r < p then
8:              diffuse_color = (c c c)
9:              return true
10:         else
11:             diffuse_color = (diffuse_color − p ∗ c) / (1 − p)
12:             return false
```

2.4.5 Denoising

Ray tracing algorithms cause noise in the image. There is a number of ways to eliminate it. It may be eliminated by a higher number of samples per pixel. However, it significantly increases the rendering time. It may be eliminated by a deep Russian Roulette depth. But it has little influence on noise and also, increases the rendering time.

The fastest solution is to use the average of all rendered images from the last change in the scene[38]. The program has a buffer in which the last frame is stored. When the image for the current frame is rendered, the program interpolates between it and the image for the last frame. When something in the scene changes, the buffer is erased. If not to do it, the final image will be influenced by images that do not match the current view. This way of denoising has the same effect as the higher number of samples per pixel, but may be used for static scenes only. The way of denoising is featured in algorithm 6.

Figure 2.6 shows a noisy image, a denoised image using the higher number of samples per pixel, and another denoised image which is the average of 100 frames.

Algorithm 6 The part of the code providing denoising.

```
1: if frame_from_last_scene_change > 0 then
2:     alpha = 1 / (frame_from_last_scene_change + 1)
3:     previous_color = buffer[pixel_id]
4:     new_color = lerp(previous_color, calculated_color, alpha)
5:     buffer[pixel_id] = new_color
```

[38] It is meant that there happened something that changed what the user sees. That is, the camera changed its position or rotation, an object moved, etc.

In addition, as it was mentioned in subsection 2.3, noise may be eliminated by AI-accelerated algorithms. With OptiX API, Nvidia provides such an algorithm.

(a) : The noisy image. 1 sample per pixel.

(b) : The first of the denoised images. 1 sample per pixel, but it is the result of the average of 100 frames.

(c) : The second of the denoised images. 100 samples per pixel.

Figure 2.6: Figure showing a noisy image and two denoised by different methods.

Chapter 3

Scene

This chapter describes parts of the work related to the scene. That is, what primitives[1] the program has, how it detects intersections between them and rays. It introduces the scene objects, describes how they are created, textured, and painted; what is required to do that and how to detect intersections between them and rays using the primitives. Special techniques for lighting at night used in this project are also described here.

To compare the procedural object representation with the traditional one, that is, polygons stored in the memory, the program is able to create polygons for all object types except one[2] and store them.

The scene in this project is a city consisting of buildings, roads, sidewalks, street lamps, a terrain, and a skydome.

3.1 Primitives

Since this project is about procedural geometry generation at run time, it is required to implement intersection programs and define primitives[3]. Each object in the scene has an axis-aligned bounding box (AABB). If a ray hits an AABB, Nvidia OptiX API calls the intersection program. In this intersection program, the programmer has to create the geometry and check if there is an intersection with it. This answers the question of how to decide when to create the geometry. For example, a sphere has a bounding box, the API calls the intersection program, and based on some data for the sphere (its center point and its radius), it checks if there is an intersection. Like this, a building has a bounding box. Based on some stored data for the building, the intersection program checks if there is an intersection. If so, it also determines with what building part the intersection is. The project supports the following primitives: a triangle, a quadrangle, a sphere, a disk,

[1] Primitives are elementary objects which are used to compose more complex objects. For example, a triangle and a sphere may be primitives in the path tracing algorithm.

[2] Except the street lamps. Further, in section 3.3.2, it will be explained why that is so.

[3] As was explained in section 2.3, OptiX API requires to implement the function which generates an object's geometry when a ray hits its bounding box.

a cylinder aligned with the vertical axis, and an arbitrary horizontal polygon. Each quadrangle may be divided into two triangles. However, the use of quadrangles is more convenient. That is why it is considered a primitive.

Let us look at how to detect an intersection between a ray and the mentioned primitives. But first, let us explain how to detect an intersection between a ray and a plane, how to find the distance between a point and a line, how to find the distance between two lines, how to sort vectors clockwise, and how to check if a point is inside a cuboid, because this knowledge is required further[4].

3.1.1 Clockwise Vector Sorting

This knowledge is used in road creation.

There are vectors $v_x = [1, 0]$ and $v_y = [0, 1]$ which are used to help with sorting. For example, we want to sort 5 vectors: $v_1 = [1, 2]$, $v_2 = [2, 2]$, $v_3 = [1, -2]$, $v_4 = [-1, -1]$ and $v_5 = [-1, 1]$ in 2D. It is possible to imagine that all of them start at the same point. This situation is visualized in figure 3.1. The dot products of the given vectors[5] and vectors v_x and v_y are used as criteria in sorting. First, the given vectors have to be sorted by the quadrants of the coordinate plane. The dot products of vectors in the 1st quadrant are nonnegative with v_x and v_y. In the 4th, they are nonnegative with v_x and negative with v_y. In the 3rd, they are negative with v_x and v_y. In the 2nd, they are negative with v_x and nonnegative with v_y. When the vectors are sorted by the quadrants, they have to be sorted in each of them. In the 1st and in the 4th quadrants, the greater the dot product with v_y is, the less the position in the sorted array is. That is, the dot product of v_1 and v_y is greater than of v_2 and v_y, so, v_1 precedes v_2. In the 2nd and in the 3rd quadrants, the less the dot product with v_y is, the less the position in the sorted array is.

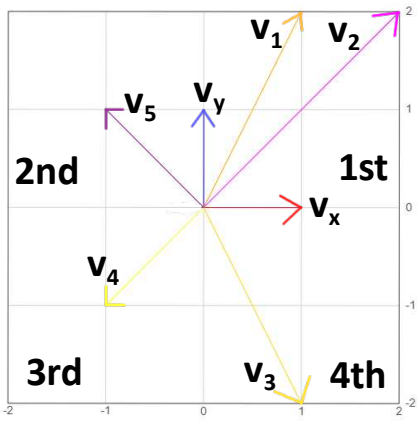


Figure 3.1: Figure showing vectors in 2D[6].

[4] As a matter of interest, I created the pictures with 3D models below with Blender [2].

[5] I assume that these vectors are normalized before the dot products are calculated.

3.1.2 Relative Position Of Cuboid And Point

This knowledge is used to optimize the building geometry creation.

There is cuboid $ABCDA'B'C'D'$, point M which is the center of $ABCDA'B'C'D'$ and another point P. We want to check if P is inside $ABCDA'B'C'D'$ or outside. If P is inside, the projections[7] of PM onto each edge of the cuboid are less than or equal to these edges' half-lengths. That is, $\text{proj}_{AB}(PM) \leq 0.5 \cdot \|AB\|$, $\text{proj}_{BC}(PM) \leq 0.5 \cdot \|BC\|$ and $\text{proj}_{AA'}(PM) \leq 0.5 \cdot \|AA'\|$.[8] [9]

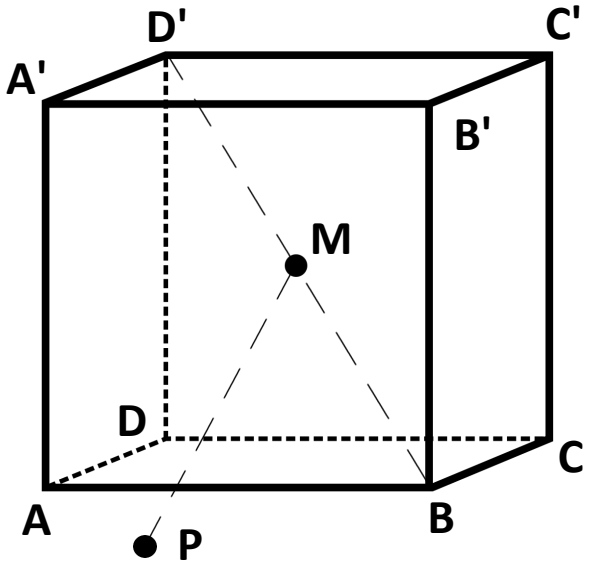


Figure 3.2: A cuboid and a point which is outside of it.

3.1.3 Ray-Plane Intersection

Each plane can be written as $(P - P_0) \cdot N = 0$, where P and P_0 are two different points on the plane and N is its normal vector. The vector between two arbitrary nonidentical points on the plane are perpendicular to the normal vector. Thus, the dot product of this vector and the normal is equal to 0. Each line may be written as $O + t \cdot D$, where O is a point on a line, t is an arbitrary real number and D is the direction. A ray is also a line, but it has a start point. So, t in the ray equation may be only nonnegative. Let P represents the intersection point between the ray and the plane, O represents the ray's origin and D represents the ray's direction. So,

[6] Made with `https://academo.org/demos/3d-vector-plotter/`.

[7] The projection is a simple mathematical operation with the cosine function. See `https://en.wikipedia.org/wiki/Vector_projection`.

[8] M is in the center. So, the projection of MP onto each edge is less then or equal to half of the edge length if P is inside.

[9] When the projection onto AB is checked, it is redundant to check the projection onto DC, $A'B'$ and $D'C'$. The same for the other edges.

$$P = O + t_p \cdot D,$$
$$(O + t_p \cdot D - P_0) \cdot N = 0,$$
$$t_p \cdot D \cdot N + (O - P_0) \cdot N = 0,$$
$$t_p = \frac{-(O - P_0) \cdot N}{D \cdot N}.$$

As it was explained, t_p must be nonnegative. If it is negative, the plane is behind the ray's origin. Figure 3.3 shows these designations.

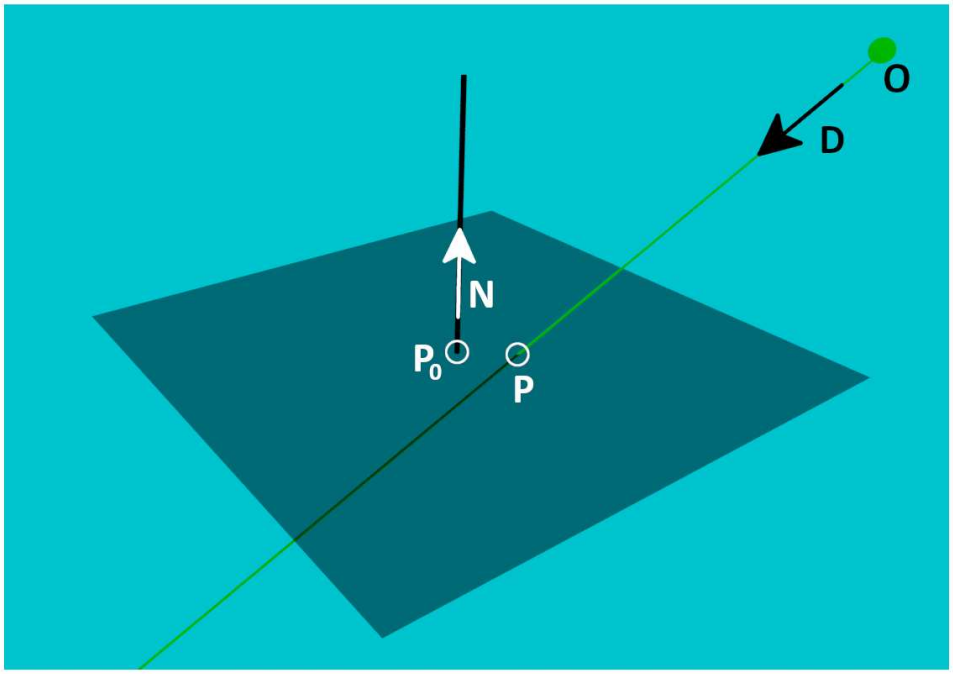


Figure 3.3: Visualization of ray-plane intersection.

3.1.4 Line-Point Distance

Let O represents a line's arbitrary point, D represents the line's direction, P represents the point to which we want to find the distance from the line and P_c represents the closest point to P on the line[10]. O, P and P_c define a triangle. From this triangle, we can obtain the rectangle OP_cPP_c'. $OP \times D$ is equal to the area $S_{OP_cPP_c'}$ of OP_cPP_c'. Another way to find the area $S_{OP_cPP_c'}$ is $\|PP_c\| \cdot \|OP_c\|$. Thus, the distance between the given line and P, that is, $\|PP_c\|$, can be calculated as $\frac{S_{OP_cPP_c'}}{\|OP_c\|}$. Figure 3.4 illustrates these designations and helps to understand the steps.

[10]The vector between a point and the closest point on a line to it is perpendicular to the line.

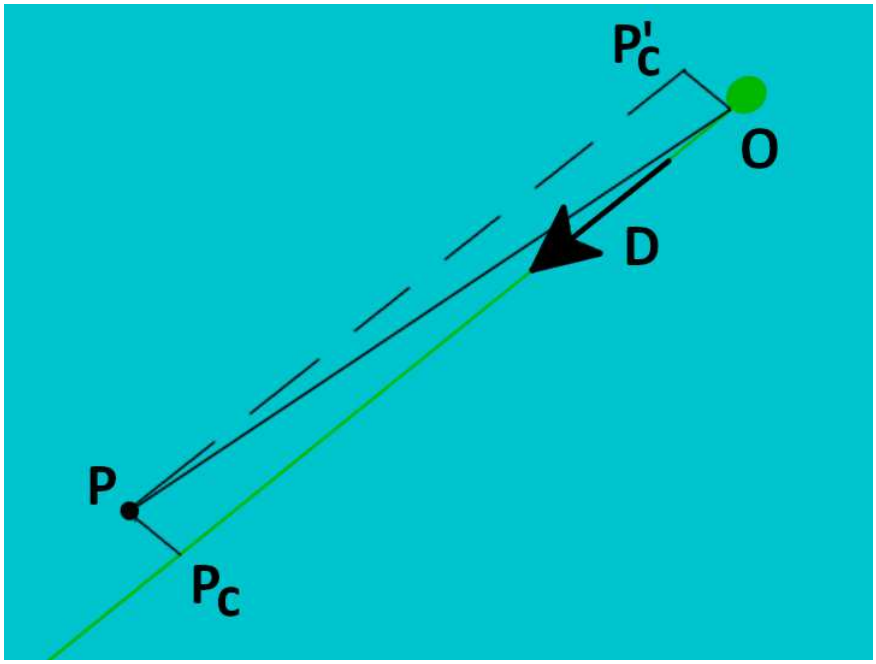


Figure 3.4: Visualization of line-point distance.

3.1.5 Line-Line Distance

Consider a line with direction L_1 and a point P_1 on it and another line with direction L_2 and a point P_2 on it. These two lines define a plane. Its normal vector N can be found as $L_1 \times L_2$. The distance d between the lines is the projection of P_1P_2 onto N. Thus, $d = \frac{P_1P_2 \cdot (L_1 \times L_2)}{\|L_1 \times L_2\|}$. Figure 3.5 helps to understand these steps.

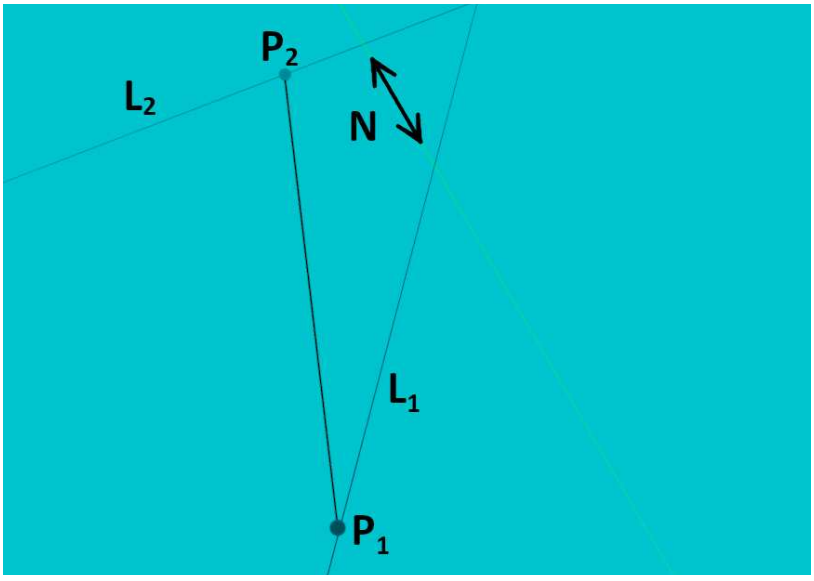


Figure 3.5: Figure visualizing markings used to calculate the distance between two lines.

3.1.6 Ray-Triangle Intersection

The most efficient way is to use the Möller–Trumbore ray-triangle intersection algorithm[11]. It is robust and perfectly suitable for ray tracing.

However, another algorithm was used previously. The first step was to find intersection P with the plane in which a given triangle with points A, B, C lies. Then, three triangles were created: PAB, PBC, and PCA. After it, the algorithm calculated area S_{ABC} of triangle ABC and areas S_{PAB}, S_{PBC}, and S_{PCA} of triangles PAB, PBC, and PCA. If sum $S_{PAB} + S_{PBC} + S_{PCA}$ was equal to S_{ABC}, this meant there was an intersection between the ray and the triangle. Unfortunately, due to floating-point arithmetic, this algorithm is not precise, it produces many holes. So, it can not be used. The Möller–Trumbore ray-triangle intersection algorithm is much more accurate and does not cause such errors.

3.1.7 Ray-Quadrangle Intersection

Each quadrangle is divided into two triangles and the ray-triangle intersection detection algorithm is used.

3.1.8 Ray-Arbitrary Horizontal Polygon Intersection

To detect an intersection between a ray and an arbitrary horizontal polygon is not trivial. Normally, such polygons are cut into triangles. However, it requires more memory. For this project, it is much better to have a single polygon instead of triangles.

Two algorithms to detect such an intersection were found. They are called the Crossing Number method and the Winding Number method[12].

Of course, these algorithms may be used for nonhorizontal polygons, that is, arbitrarily rotated in 3D space. The points of a polygon may be transformed into a space in which the polygon is horizontal. Since the scene has only horizontal polygons[13], this transformation is not implemented.

3.1.9 Ray-Disk Intersection

To detect an intersection between a ray and a disk, it is required to check if the ray intersects the plane in which the disk lies and to check if the distance between the disk's center and the intersection point of the ray and the plane is less than the disk's radius. Let C represents the disk's center, P represents the intersection point between the disk's plane and the ray, r represents the disk's radius, D represents the ray's direction and O represents the ray's origin. Then, the ray intersects the disk if and only if $\|P - C\| < r$ Figure 3.6 shows these designations.

[11] See [24].
[12] See [42].
[13] Arbitrary horizontal polygons are the bottom and upper parts of the buildings.

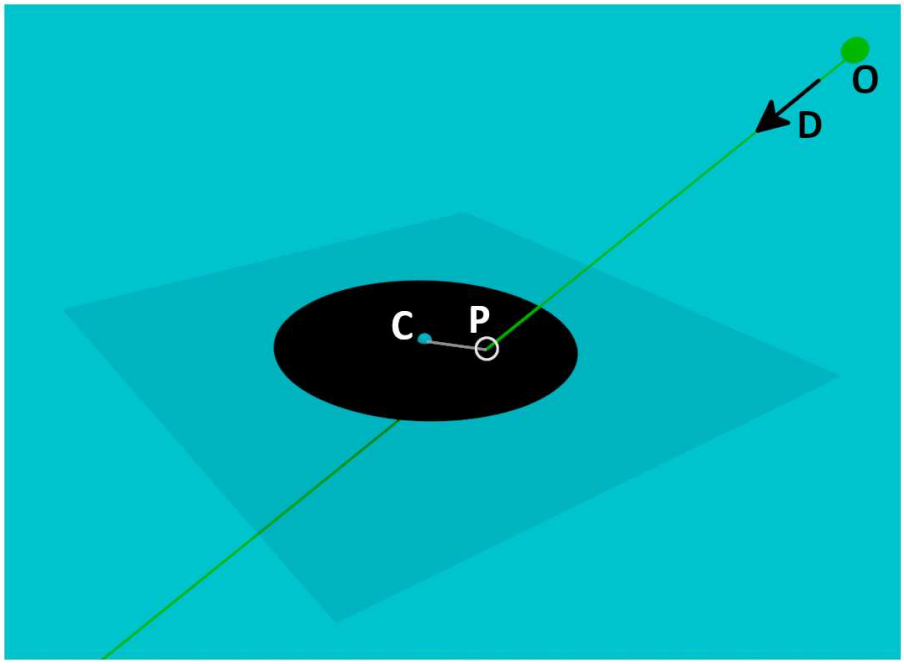


Figure 3.6: Visualization of ray-disk intersection.

3.1.10 Ray-Sphere Intersection

Intersections between rays and spheres are detected by using analytical geometry. The algorithm using it turned out to be very precise.

Let O represents a ray's origin, D represents the ray's direction, C represents a sphere's center, r represents the sphere's radius, $P = O + t_p \cdot D$ represents the intersection point between the ray and the sphere and P_c represents the closest point to the sphere's center on the ray. First, find the distance between the ray and the sphere center which is equal to $\|P_cC\|$. If it is greater than the radius, the ray does not intersect the sphere. Then, find the distance between the ray's origin and the sphere's center which is equal to $\|OC\|$. The length of $\|PC\|$ is equal to the radius. P_cC is perpendicular to the ray's direction D[14]. Thus, $\|PP_c\| = r^2 - \|P_cC\|^2$ and $\|OP_c\| = \|OC\|^2 - \|P_cC\|^2$. Now, we can obtain t_p which is equal to $\|OP\|$ as $\|OP_c\| - \|PP_c\|$[15]. Figure 3.7 illustrates these designations.

But there is a simpler algorithm to do the same job. It is to use the parametric equations of a line and a sphere. This method is very inaccurate in terms of floating-point arithmetic and does not work with spheres with a small radius.

[14] Because the vector between a point and the closest point on a line to it is perpendicular to this line.

[15] Also, we can find the intersection point P' between the ray's line and the sphere on its other side as $\|OP_c\| + \|PP_c\|$, but we're interested only in the closest intersection point.

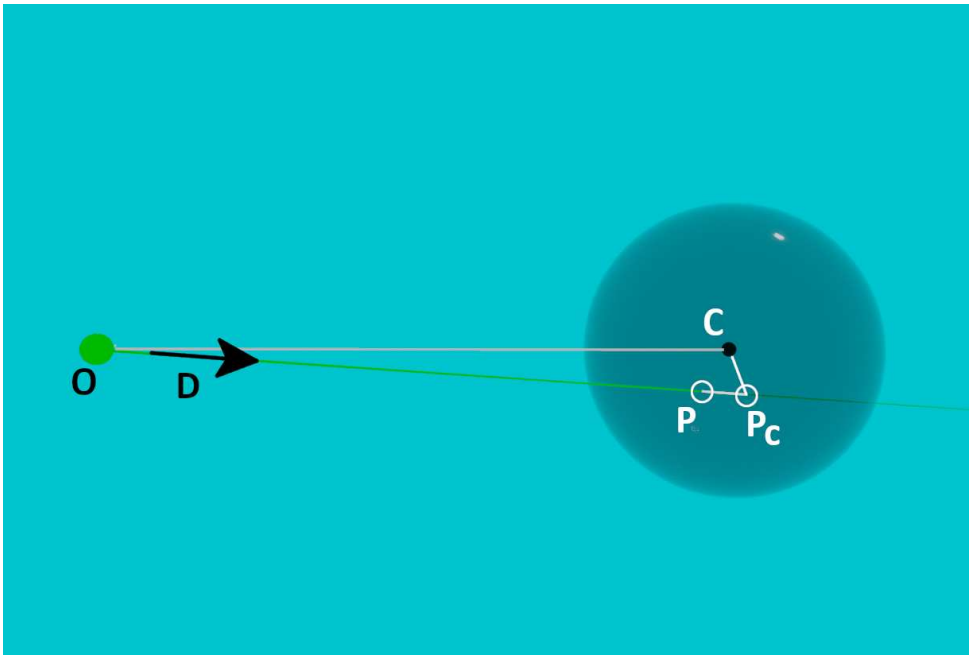


Figure 3.7: Visualization of ray-sphere intersection.

3.1.11 Ray-Vertically Aligned Cylinder Intersection

To detect an intersection with such a primitive, it is required to check an intersection with the covers of the cylinder (that is, disks) and its body. As with the sphere, there is an algorithm using analytical geometry to check an intersection with the cylinder's body.

Let R represents a ray, O represents its origin, D represents the direction of R, C represents the closest point between the cylinder's center line, which is marked with L, and R, r represents the cylinder's radius, $P = O + t_p \cdot D$ represents the intersection point between R and the cylinder, P_c represents the closest point to L on R. Then, assume the projection of R onto the plane perpendicular to L. The new ray will be marked with R'. $D' = \frac{[D_x,0,D_z]}{\|[D_x,0,D_z]\|}$ represents the direction of R', C' represents a the closest point between L and R', $P' = O + t_{p'} \cdot D'$ represents the intersection point between R' and the cylinder, P'_c represents the closest point to L on R'. $\|P'_cC'\|$ is the distance between R' and L. If it is greater than r, R does not intersect the cylinder. $\|C'P'\|$ is equal to r. Thus, $\|P'_cP'\| = \|P'C'\|^2 - \|P'_cC'\|$. P'_cC' is perpendicular to D'. Hence, $\|P'P'_c\| = r^2 - \|P'_cC'\|^2$ and $\|OP'_c\| = \|OC'\|^2 - \|P'_cC'\|^2$. Now, $\|OP_c\| = \frac{\|OP'_c\|}{\cos\angle P'OP}$.[16] Since this algorithm is used to detect an intersection with a vertically aligned cylinder, it is required to check y value of any possible intersection. It must be in some range which is used for the current cylinder. Figure 3.8 helps to understand these steps.

[16] Again, we can find the intersection point P'' between the ray's line and the cylinder on its other side as $\frac{\|OP'_c\|+\|P'P'_c\|}{\cos\angle P'OP}$, but we are interested only in the closest intersection point.

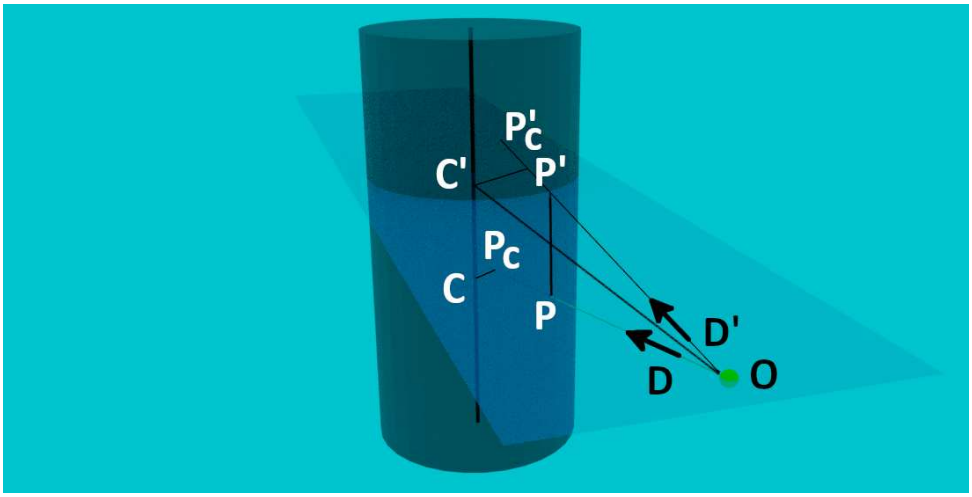


Figure 3.8: Visualization of ray-cylinder intersection.

3.2 Scene Data Preparation

This section describes how the data which the rendering application uses to create 3D objects is prepared. The project uses OpenStreetMap (OSM)[17]. To process the OpenStreetMap data, I developed a program. It reads a file with data exported from the OSM website and saves the required data in an own simple format.

3.2.1 OpenStreetMap

OpenStreetMap is a free map that can be edited by any person. This way of map maintaining allows people to obtain actual information about the city infrastructure in our world. On the other hand, it may result in inaccurate data. OSM allows exporting its map data in the XML format.

3.2.2 OpenStreetMap Data Representation

This project uses data about roads and buildings only. Other data is not used.

In OSM, each building is represented by an ordered list of points that define the building's contour (polygon), a tag that defines what type of building it is, and other information. Often, the buildings do not have information about their height. Figure 3.9 visualizes building polygons in Unity. Each color represents a single building.

Almost like the buildings, the roads are represented by two nodes, a tag that defines the type of the road, and other information. The edge between

[17] See [4].

these two nodes defines the centerline of the road. Unfortunately, there is no data defining the shape of the road. Figure 3.10 gives a visual example of road edges. Each color represents a single road edge.

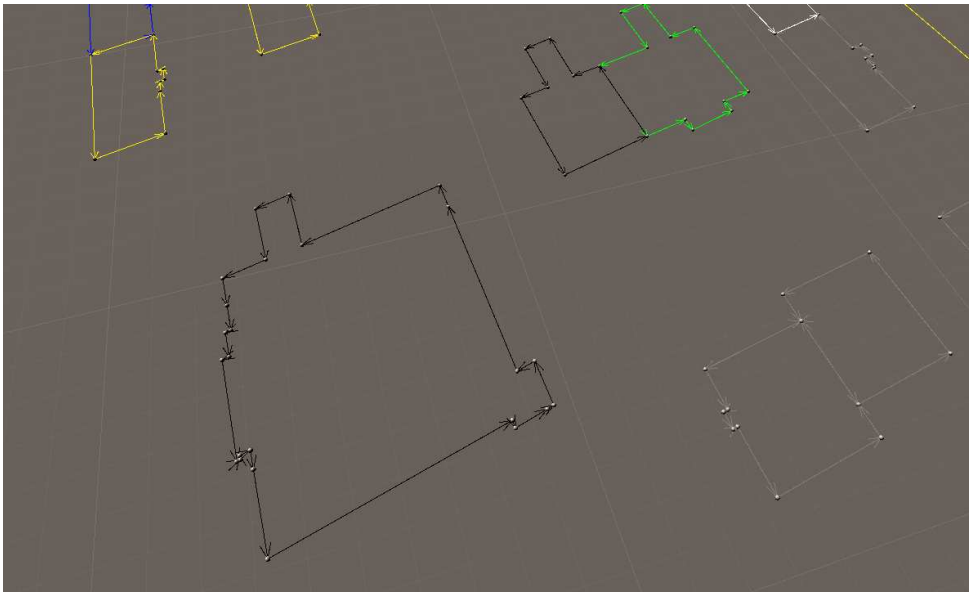


Figure 3.9: Visualization of building contours which can be retrieved from OSM.

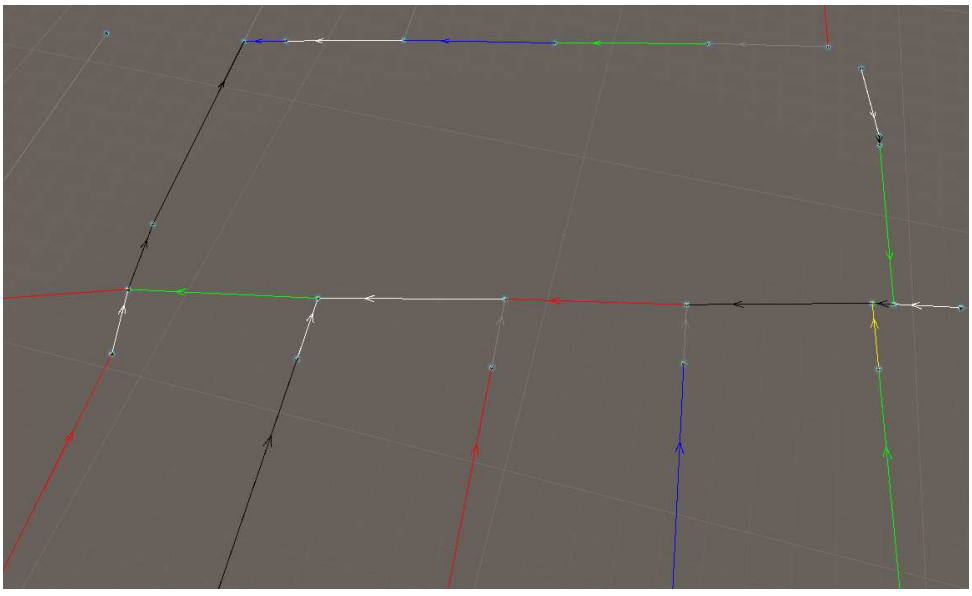


Figure 3.10: Visualization of road edges which can be retrieved from OSM.

3.2.3 OpenStreetMap Data Processing

The rendering application requires some data which is used to create the 3D models. That is why there is an application to process OSM data. It uses

the OsmSharp library[18]. It allows to easily obtain data about building and road nodes and other information and to store it for the required purposes. Since rendering application is not able to generate the geometry of complex objects, such as a church or buildings with nontrivial architecture; and OSM has some data which are not required, such as data about crossings; it is reasonable to ignore buildings and roads of some selected types[19] when processing a file.

There are many other works that solve the same or a similar problem - the use of OSM to create a model of a city. For example, Xingjiang Yu solves the problem of road network creation using the OSM data[20]. The article *Generating web-based 3D City Models from OpenStreetMap: The current situation in Germany*[21] discusses the generation of city models based on OSM.

Processing Data For Roads And Sidewalks

Because there is no information about the road shapes, this application has to prepare it. For this purpose, it uses the nodes of roads that are not crossings and the nodes of buildings. For each road edge, the program produces either a convex quadrangle or no geometry if it is not possible to create it[22]. Each road edge is assigned an initial width. The program makes a rectangle from each road edge with this width and the edge's length. Then, it checks if any point of the rectangle lies inside any building's polygon or if any building's node lies inside the rectangle. If so, the assigned width is decreased and these steps are repeated until this width is less than the selected threshold. If not, there is an edge road defined by this rectangle. Then, the application creates the geometry for the crossings and adjusts the roads' rectangles so they are properly connected. Thus, the rectangles become quadrangles[23]. If two road edges start at the same node, it only adjusts the quadrangles so they are connected. If three or more edges start at the same node, a joint road is created at this node. The program sorts all edges of the node clockwise. Then, it calculates the intersection points of the corresponding edges of these quadrangles. Using these intersection points, the program adjusts the road quadrangles so they are connected properly. The intersection points define a polygon. This polygon is used for the joint road.

To prepare data for sidewalks, an additional simple thing is required. This thing is to add a narrower rectangle inside the rectangle mentioned

[18] See [5].

[19] To remember, the buildings and the roads has tags. Entities with selected tags may be ignored while reading a file.

[20] See [49].

[21] See [31].

[22] Often because the edge overlaps a building.

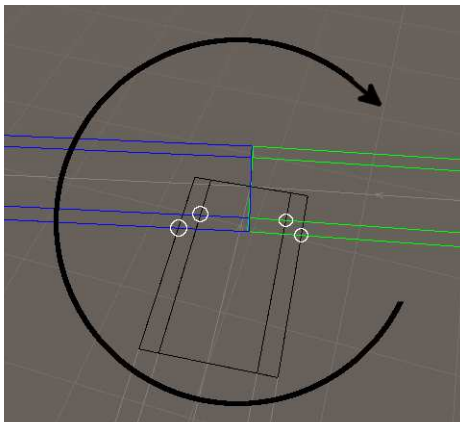
[23] Further, I use the word *rectangle* to describe figure which is assigned to a road edge at the beginning. After, I use the word *quadrangle* to describe the assigned figure, because it is not correct to use the word *rectangle* again since it is not actually a rectangle after the adjustment.

above. When adjusting the shape of the bigger quadrangle, the program also adjusts the shape of the narrower quadrangle. The described actions are applied to both quadrangles. The space between the quadrangles is used for sidewalks.

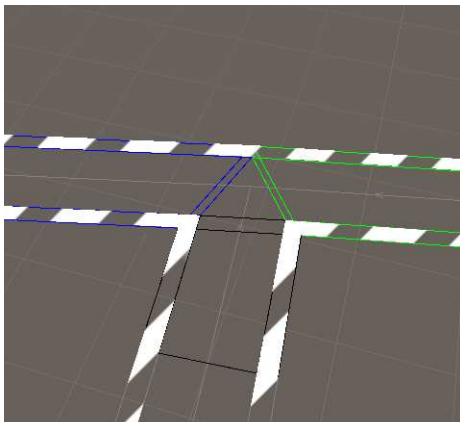
Figure 3.11 summarizes these steps. Figure 3.13 shows edges obtained from OSM, rectangles created from the edges, and these rectangles transformed into quadrangles which are connected to each other. Figure 3.12 shows a node at which three edges start, how the edges' rectangles become connected quadrangles, and what space is used for sidewalks.

1. From each edge, create an inner and an outer rectangle using the forward and left vector and the current width.
2. If these rectangles overlap any building, decrease the current width and return to the 1st step. Otherwise, proceed to the next step.
3. For each node,
   (a) If two edges start here, adjust the inner and the outer quadrangles of them using the intersection points of the corresponding edges.
   (b) If three or more edges start here,
      (1) Sort the edges clockwise.
      (2) Find the intersection points for the corresponding edges of each rectangle.
      (3) Using these intersection points, adjust the points of the quadrangles.
      (4) From these intersection points, create a polygon and use it for a joint road.

Figure 3.11: Steps of road and sidewalk creation.

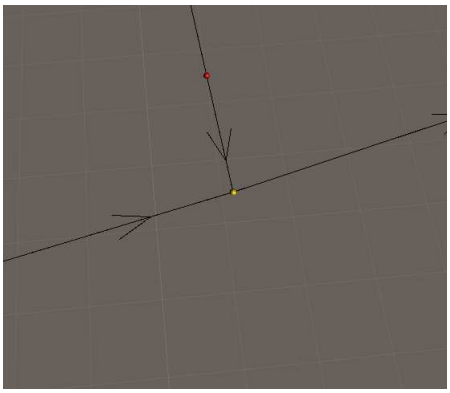


(a) : The direction of road edge sorting and the intersection points of the road edges' quadrangles.

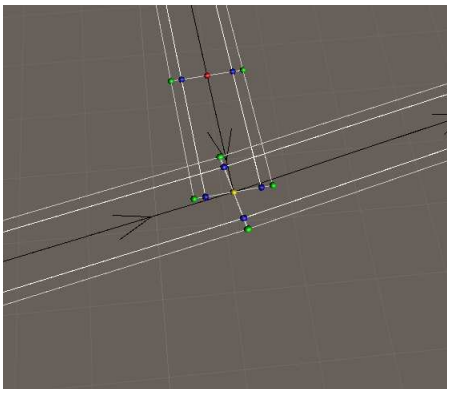


(b) : The quadrangles are adjusted. The place for the sidewalks is shaded in white.

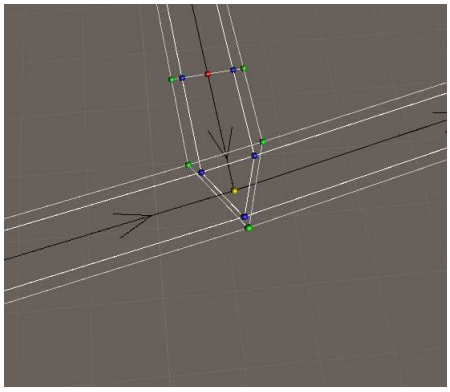
Figure 3.12: Figure explaining some steps of road and sidewalk creation.



(a) : There are edges only.



(b) : There is an outer and an inner rectangle for each edge created.



(c) : The shapes of the quadrangles are adjusted. The space between the quadrangle is used for sidewalks.

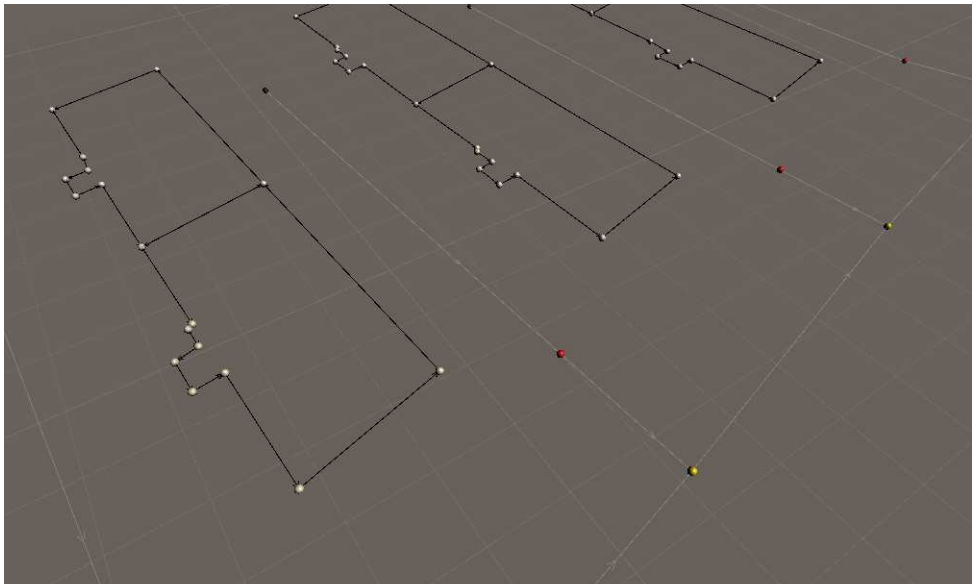
Figure 3.13: Process of road and sidewalk creation.

Simplification And Manual Editing

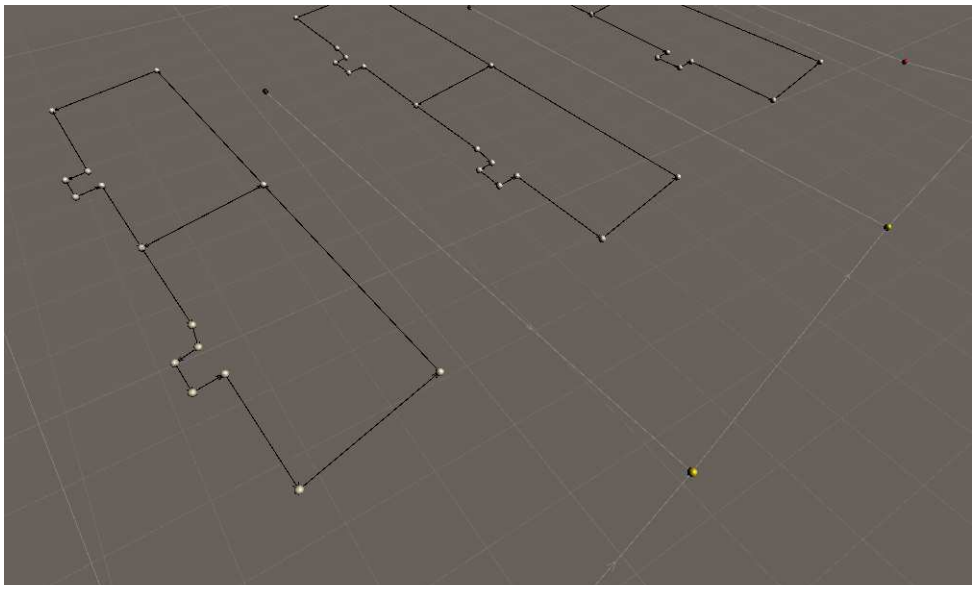
It is worth simplifying the obtained data, because it may reduce the rendering time without unacceptable losses. If the angle between two road edges is less than the selected threshold, these edges are merged into one. Similarly, if the angle between two edges of a building's polygon is less than the selected threshold, they are merged into one. An example of simplification is shown in figure 3.14. The first screenshot of the figure contains more nodes than the second.

Sometimes, the described algorithm for road creation does not work, because OSM data may be misleading or inaccurate. It may be difficult to choose the types of roads and buildings which have to be ignored when processing the file, because there is a large number of them. If the user sees everything, they can understand what should be deleted or slightly edited.

For this purpose, there is a graphical user interface based on Unity which allows to set the parameters for OSM data parsing and to manually edit the parsed data.



(a) : No simplification.



(b) : The graph was simplified. Some nodes disappeared.

Figure 3.14: Figures showing graph simplification. The nodes are designated by spheres.

3.3 Objects

3.3.1 Buildings

There is a number of works that solve the problem of building geometry creation. The main source of inspiration for me was a programming language called *CGA shape grammar*[24]. Other works related to this topic are *Wall Grammar for Building Generation*[25], *Advanced Procedural Modeling of Architecture*[26] *Instant Architecture*[27], *Interactive sketching of urban procedural models*[28].

In this project, a building is created from a polygon that defines its contour. This polygon is extruded in the vertical direction. Hence, it is easy to obtain the walls from the extruded polygon. The walls are used as a basis for facades. The program tries to fit the facade parts into the walls. A building polygon, the polygon after extrusion, and the extruded polygon with a highlighted wall are shown in figure 3.15.

Facades

In real life, the ground floor may be used as a shop. Often, such shops have big windows, because they need to have showcases. Based on the selected probability, the program decides if the ground floor of a building will be a floor with shops. If so, it takes initial shop window parameters: the size and the distance between windows and adjusts them so the windows fit into the wall.

There are two variants for the other floors of the building: they may have windows only or windows with balconies. Like the shop windows, the program takes initial parameters. However, in this case, it does not change the size of the windows and the balconies[29].

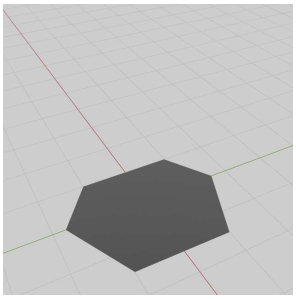
So, in this project, the ground floor of the building may have shop windows or may not have. The other floors of the buildings and the ground floor if it has no shop windows may have either windows only or windows and balconies. Look at figure 3.17 to get an idea of what the facades look like.

Let us explain how the facades are created. The program has initial data for a building: an extruded polygon. The walls of it are split into floors, the floors are split into slots. To start the creation of a window or a balcony in a slot, it is required to determine an initial rectangle within the slot which is used as a basis for the window or balcony geometry. The initial rectangles are

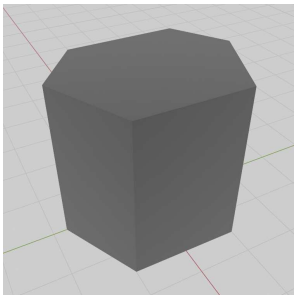
[24] See [14, CGA].
[25] See [20].
[26] See [38].
[27] See [48].
[28] See [25].
[29] In the case of the shop windows, they are big and they look realistic with whatever size. So, a shop window may have its size either 2 m × 4 m or 2.5 m × 5 m and it looks good. But in the case of the windows on the other floors, that is, the windows for apartments, they should have a determined size.

featured in figure 3.16. Based on the predetermined values, the program uses this rectangle and its normal vector to create the 3D geometry. Figure 3.18 explains how the walls are split and how the initial rectangles are prepared for windows and balconies, figures 3.19 and 3.20 show steps of window and balcony creation from the initial rectangles. Figure 3.21 visualizes the steps of how the facade with windows and balconies is created from a wall[30].

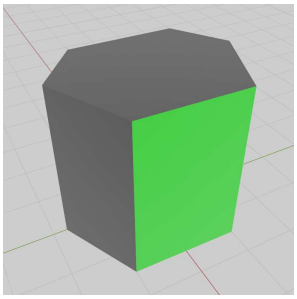
The algorithm of facade creation is used for both geometry representations. The difference is that for the traditional geometry representation, the geometry is created once and it is stored in the memory.



(a) : The building polygon.



(b) : The polygon after extrusion..



(c) : The extruded polygon with a highlighted wall.

Figure 3.15: Figure showing a building polygon, the polygon after extrusion and the extruded polygon with a highlighted wall.

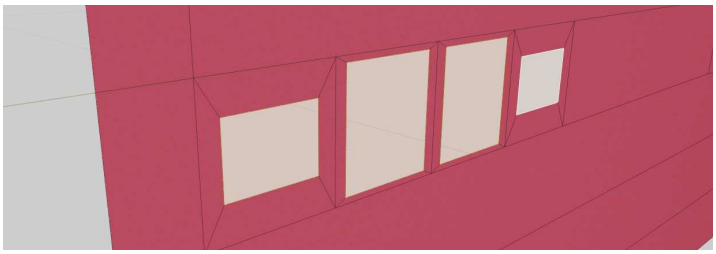


Figure 3.16: Figure demonstrating initial rectangles for windows and balconies.

[30] See appendix D for the images in a larger resolution.

(a) : Windows only.

(b) : Windows and balconies.

(c) : Windows, balconies, and shop windows.

Figure 3.17: Figure showing the facades used in the project.

Having a polygon of a building,

1. For each wall, split it into floors and cut some space at the wall's top and the bottom edges if needed.
2. For each floor, split it into slots and cut some space at the wall's edges if needed.
3. For each slot,
   (a) Determine an initial rectangle for a window and create the window geometry.
   (b) Determine initial rectangles for 2 windows and 2 balconies. Create the geometry for the windows and the balconies.

Figure 3.18: Steps explaining how a building wall is split and how are prepared initial rectangles for windows and balconies.

From a determined initial rectangle $R_{initial}$, to create a window,

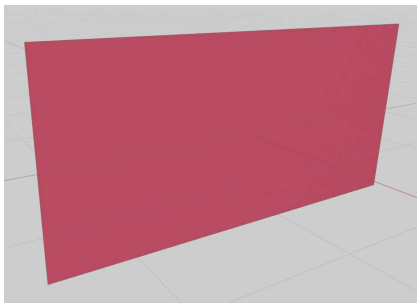
1. Intrude a copy of $R_{initial}$ inside of the polygon by a given distance using its normal vector and remove $R_{initial}$.
2. Use the intruded rectangle R_{glass} for the glass.
   - $R_{initial}$ (imagine it was not removed) and R_{glass} define a cuboid. Its walls except the wall corresponding to $R_{initial}$ (because it was actually removed) should have the same material as the building's wall.
3. Intrude a copy of R_{glass} inside by a given distance.
   - R_{glass} and the new rectangle $R_{room_{bg}}$ define a cuboid. Its walls except the walls corresponding to R_{glass} should have a room material.

Figure 3.19: Steps explaining how a window is created from an initial rectangle.

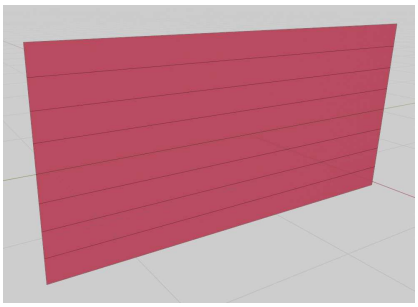
From a determined initial rectangle $R_{initial}$, to create a balcony,

1. Determine a rectangle $R_{initial}$ within a building's wall.
2. Cut horizontally a copy of $R_{initial}$ into two rectangles R_{glass} (upper) and $R_{fence_{fg}}$ (lower).
3. Use R_{glass} for the glass.
4. Use $R_{fence_{fg}}$ to create the fence.
5. Intrude a copy of $R_{fence_{fg}}$ inside by a given distance which is equal to the thickness of the fence.
   - $R_{fence_{fg}}$ and the new rectangle $R_{fence_{bg}}$ define a cuboid. Its walls should have a fence material.
6. Intrude $R_{initial}$ inside by a given distance and remove it.
   - $R_{initial}$ (imagine it was not removed) and the new rectangle $R_{balcony_{bg}}$ define a cuboid. Its walls except the wall corresponding to $R_{initial}$ should have a balcony material.

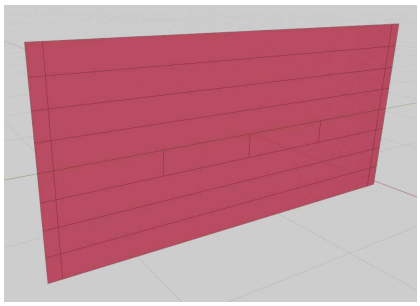
Figure 3.20: Steps explaining how a balcony is created from an initial rectangle.



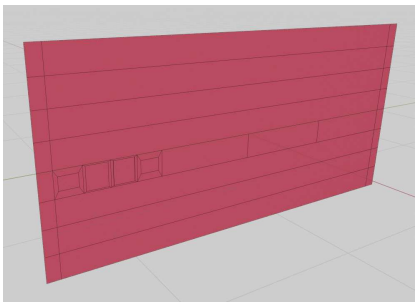
(a) : An initial wall which is used to create the geometry of the facade.



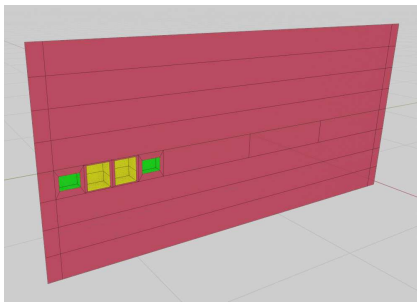
(b) : The wall is split into floors.



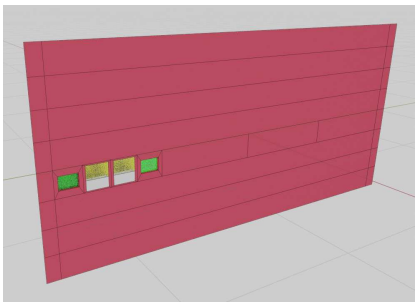
(c) : Some space at the wall's edges is cut. A floor is split into slots.



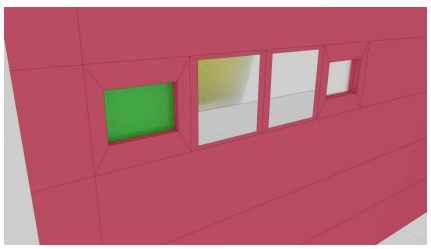
(d) : Initial rectangles for windows and balconies.



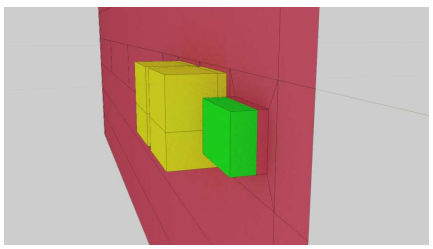
(e) : The rectangles were intruded.



(f) : The geometry for the windows and the balconies is created.



(g) : The geometry near from the front side.



(h) : The geometry near from the back side.

Figure 3.21: Figure showing how to create the facade with windows and balconies.

In the implementation, the walls of the buildings have tile textures. UV mapping is explained further in section 3.4. The other facade parts have a uniform color.

Building Intersection Program

This part of the text explains how the intersection program for the building works[31]. First, the program checks if there is an intersection with the covers of the building's initial polygon. Then, it checks if there is an intersection with the building's facade. That is, the geometry has to be created. The geometry created on wall W_i may be bounded by a box. The program checks if the ray intersects the bounding box of the wall's geometry or its origin lies inside of it[32]. If not, the ray can not intersect the geometry of W_i. In this case, it is irrational to create the geometry, because it is a waste of time. If the ray intersects the bounding box or it lies inside of it, the geometry on this wall has to be created. The program splits the wall and creates windows or/and balconies. Then, the program checks an intersection with primitives belonging to the windows and balconies only. If there is no intersection with any window and balcony, there may be an intersection with the parts of W_i which do not belong to the windows and balconies. If there is an intersection with such a part, it must lie within W_i. Thus, it is enough to check if the ray intersects W_i, there is no need to check an intersection with these parts separately[33]. Figure 3.22 summarizes the steps of the ray-building intersection detection and algorithm 7 shows pseudocode for these steps.

Having the polygon of a building,

1. Check if there is an intersection with the covers of the polygon.
   - If so, report the intersection, stop and ignore the next steps.
2. Iterate through the walls of the polygon.
   (1) Create a bounding box for the geometry on W_i and check if there is an intersection or if the ray's origin is inside of it.
   - If **not** so, proceed to the next wall.

   (2) Create the geometry for the windows and balconies on each wall W_i of the polygon and check if there is an intersection.
   - If so, report the intersection, stop for W_i, ignore the next step, and proceed to the next wall.

   (3) Check if there is an intersection with W_i.
   - If so, report the intersection and proceed to the next wall.

Figure 3.22: Steps to detect an intersection between a ray and a building.

[31] Remember that OptiX API requires implementing the function which detects intersections.
[32] This is an optimization to reduce the rendering time.
[33] It is another optimization.

Algorithm 7 The function to detect an intersection between a ray and a building.

```
1:  function RAY_BUILDING_INTERSECTION(ray, ext_pgn)
2:      P = (0, 0, 0)                    ▷ A variable for the possible intersection point.
3:      if INTERSECTION(P, ray, ext_pgn.upper_cover) then
4:          REPORT_INTERSECTION(P)
5:          return
6:      if INTERSECTION(P, ray, ext_pgn.lower_cover) then
7:          REPORT_INTERSECTION(P)
8:          return
9:      for each wall W_i in ext_pgn.walls do
10:         box = CREATE_BBOX_FOR_GEOMETRY(W_i)
11:         if not (INTERSECTION(P, ray, box) or INSIDE(ray, box)) then
12:             continue
13:         G = CREATE_GEOMETRY(W_i)
14:         if INTERSECTION(P, ray, G) then
15:             REPORT_INTERSECTION(P)
16:             continue
17:         if INTERSECTION(P, ray, W_i) then
18:             REPORT_INTERSECTION(P)
```

3.3.2 Street Lamps

A street lamp consists of three parts: a bulb, a stand, and a glass. The stand is represented by a cylinder. The bulb and the glass are represented by spheres slightly cut off at the bottom (truncated spheres).

The program does not support the street lamps stored in the memory as triangles. The main reason why not to store is that all parts of the street lamp are rounded. Triangulation would produce a less realistic and beautiful object.

All parts of the street lamp have a uniform color.

Street Lamp Intersection Program

The detection of a ray-street lamp intersection requires checking if there is an intersection with the cylinder and with the truncated spheres. To detect an intersection with a truncated sphere, the program checks if there is an intersection with the whole sphere. If so, it checks y value. y value of the whole sphere is in range $[y_{min}, y_{max}]$. But the truncated sphere's y value is in range $[y_{cut}, y_{max}]$, because it is cut off at the bottom. Figure 3.23 summarizes these steps.

Having the data for a street lamp,

1. Create a sphere and check if there is an intersection. y value for all points of the sphere is in the range $[y_{min}, y_{max}]$.
   - If so, check if y value of the intersection point is in the range $[y_{cut}, y_{max}]$. In this case, report the intersection.
2. Create a cylinder and check if there is an intersection.
   - If so, report the intersection.

Figure 3.23: Steps to detect an intersection between a ray and a street lamp.

3.3.3 Roads

As it was mentioned above, there are edge roads and joint roads. The edge roads are extruded quadrangles. The joint roads are extruded arbitrary polygons. Hence, the detection of a road-ray intersection is trivial. Each edge road may have a texture, but it does not have if it is too narrow. The joint roads and narrow edge roads have a uniform color.

Edge Road Texturing

Textures for the edge roads are created procedurally at run time. Actually, the program does not use UV mapping in this case. It uses a special algorithm which is explained here.

The program starts with quadrangle $ABCD$ of an edge road and creates rectangle $A'B'C'D'$ which is the biggest rectangle that may fit into $ABCD$. The edges of $ABCD$ may be divided into the left - DC -, right - AB -, start - AD -, and end - BC. The same for $A'B'C'D'$: $D'C'$, $A'B'$, $A'D'$ and $B'C'$. The program finds middle points M_s and M_e on the start and end edges. From these points, it creates the forward vector v_f as $normalize(M_e - M_s)$. In 2D space, it is possible to find the left vector v_l from v_f as $normalize([-v_{f_x}, 0, v_{f_z}])$. The left and right edges of $A'B'C'D'$ are equal to the corresponding edges of $ABCD$. The start and end edges have the same orientation as v_l. From the points A, B, C and D of $ABCD$. The program determines which points of $A'B'C'D'$ lie inside of $ABCD$[34] [35]. Then, the program takes the initial parameters for road marking lines: the widths and the distances between them. Then, rectangle $A'B'C'D'$ is used for texturing. It is trivial to split the rectangle into stripes[36]. Based on

[34] Two points of $A'B'C'D'$ are the same as two points of $ABCD$, but the other two points lie inside $ABCD$.

[35] The points are determined by the dot products of vectors M_sA, $M_sB]$ and v_f. That is, $dot(M_sA, v_f)$ and $dot(M_sB, v_f)$. The signs of the dot products say which points are inside.

[36] Points A' and B' are moved in the direction of v_l.

it, the program determines how many lanes fit into it and how many road marking lines may be there and adjusts the distances between them. Each road marking line has the form of a rectangle. The program starts at a point of $ABCD$ and in a for-loop, checks if the intersection point lies inside any road marking line. If not, it uses the asphalt color. Figure 3.24 visualizes the designations for the road texturing.

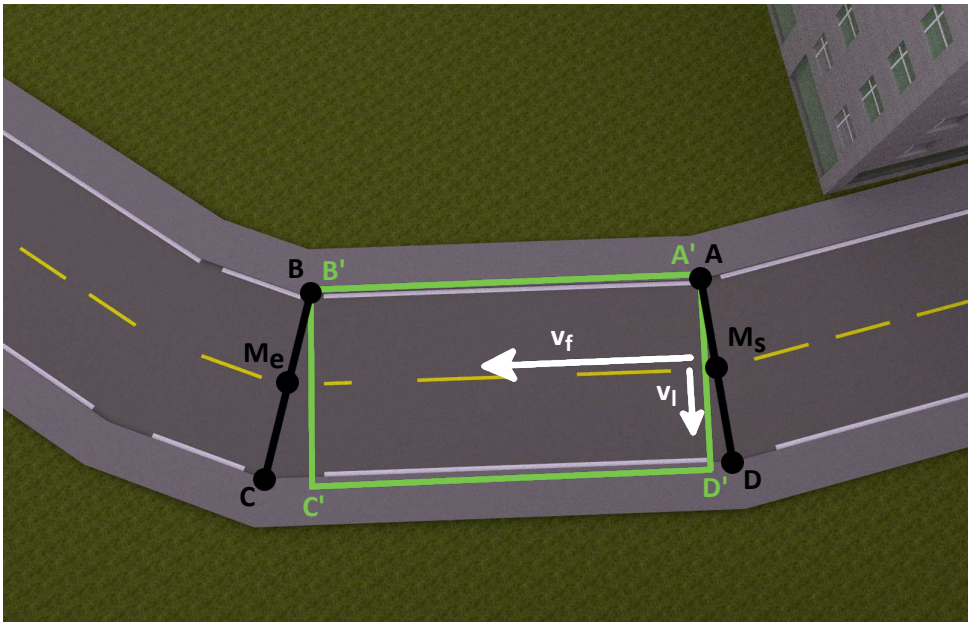


Figure 3.24: Figure visualizing the designations for the road texturing.

3.3.4 Sidewalks

Sidewalks are similar to the edge roads: they are extruded quadrangles. They have no texture, just a uniform color.

3.3.5 Terrain

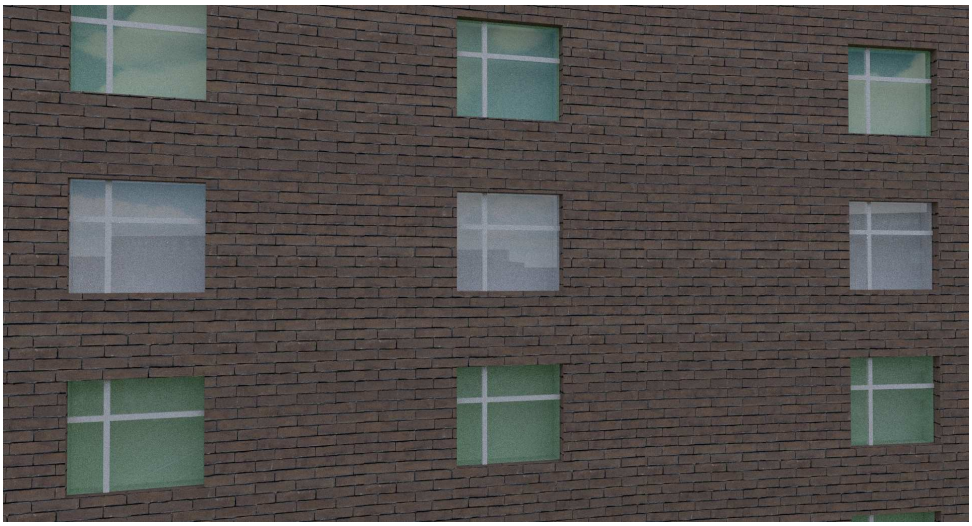
The terrain in the project is a cuboid with a tile texture.

3.4 Tile Textures And UV Mapping

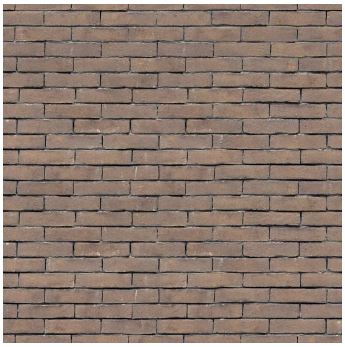
In this project, tile textures are used for the buildings and terrain. A tile texture is a type of texture that is repeated within a surface. Hence, such a texture has to be seamless. An example of such a texture and a screenshot from the rendering application is shown in figure 3.25. Each tile texture in

this project has parameters t_h and t_v which define how frequently it should be repeated within a face. For example, if t_h is 1, a texture would repeat every meter in the horizontal direction.

There are two ways how to map an intersection point into a tile texture. One is used for the procedural geometry representation, the other is used for the traditional one.



(a) : The screenshot of the tile texture.



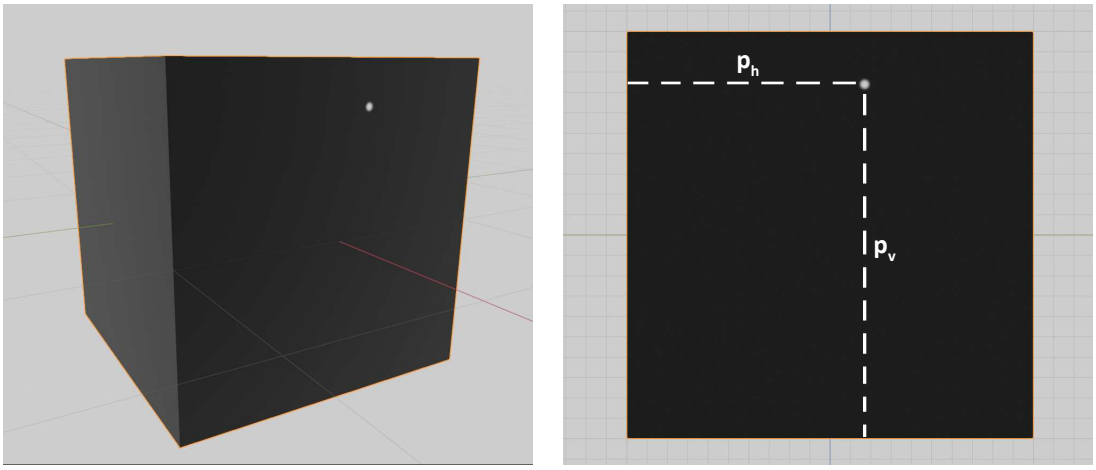
(b) : The tile texture[37].

Figure 3.25: Figure showing a tile texture and a screenshot from the rendering application with it.

[37] Taken from [3].

3.4.1 UV Mapping And Procedural Geometry

An intersection point has coordinates p_h and p_v in a plane defined by the face. These coordinates are the distances from the plane edges. They are visualized in figure 3.26. Algorithm 8 is used to get a color from the texture.



(a) : The cube and the intersection point. **(b) :** p_h and p_v values of the intersection point.

Figure 3.26: Figure showing a cube, an intersection point on it (the white point) and p_h and p_v values for UV mapping.

Algorithm 8 The function for UV mapping for tile textures.
This function maps each coordinate of an intersection point into a tile texture.
value is p_h or p_v, that is, the distance from a plane edge.
t is t_h or t_v.
It returns a value in the range [0, 1].

```
1: function UV_MAP(value, t)
2:     value = value mod t
3:     value = value/t                ▷ convert from [0, t] to [0, 1]
4:     return value
```

3.4.2 UV Mapping And Geometry Stored In The Memory

If the geometry is not created at run time, the UV coordinates for each vertex are stored in the memory. They do not have to be in range $[0, 1]$, because the texture has to be repeated. If a ray intersects a triangle ABC, the intersection point P may be written as $P = A + b_x \cdot AB + b_y \cdot AC$. The coordinates $[b_x, b_y]$ are called barycentric coordinates[38]. The points A, B and C have UV coordinates $A_{uv} = [A_{uv_x}, A_{uv_y}]$, $B_{uv} = [B_{uv_x}, B_{uv_y}]$ and $C_{uv} = [C_{uv_x}, C_{uv_y}]$. So, the UV coordinates for P may be found as $P_{uv} = A_{uv} + b_x \cdot A_{uv}B_{uv} + b_y \cdot A_{uv}C_{uv}$. Then, algorithm 8 is used to get a

[38] See [6, Volume 1: Foundations of 3D Rendering, Ray Tracing: Rendering a Triangle].

color from the texture. In the case when it is required to texture a building's wall or the terrain, p_h and p_v values, which are shown in figure 3.26, are used as the UV coordinates. The building's wall is cut into triangles. For each vertex of each triangle, p_h and p_v values are calculated and stored in memory.

3.5 Skydome

3.5.1 Textures

The skydome uses textures to make the environment more realistic. The program loads them from files and procedurally generates one: a texture of a night sky. Such a texture is a black texture with white points - stars. An example of it is shown in figure 3.27 and the algorithm 9 explains how the texture is created.

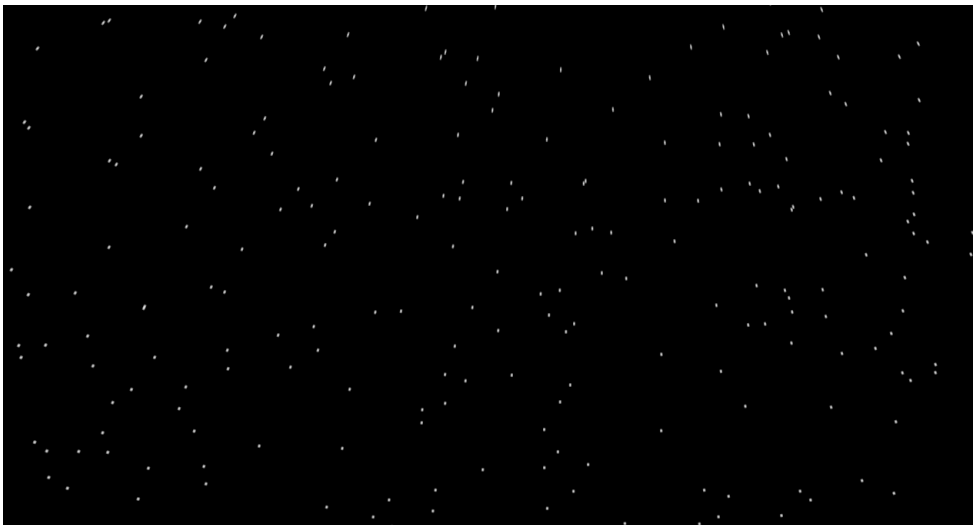


Figure 3.27: Figure showing a texture of a night sky.

Algorithm 9 The function to create a texture of a night sky.

```
1:  function CREATE_NIGHT_SKY(width, height, star_density)
2:      texture = new Texture(width, height)
3:      for y in range(0, height) do
4:          for x in range(0, width) do
5:              r = RANDOM_FLOAT.RANGE(0, 1)
6:              if r < star_density then
7:                  texture[x, y] = black_color
8:              else
9:                  texture[x, y] = white_color
10:     return texture
```

3.5.2 UV Mapping

If a ray hits the skydome, its direction is used to calculate the UV coordinates for the hit point. To calculate v value, y coordinate of the direction is normalized[39]. To calculate u value, it is necessary to use x and the z coordinates. The most suitable way for this purpose is to use the $atan2$[40] function. Figure 3.28 shows what it exactly does. When we have this angle, we have to normalize it. Then, we have u and v coordinates for the texture.

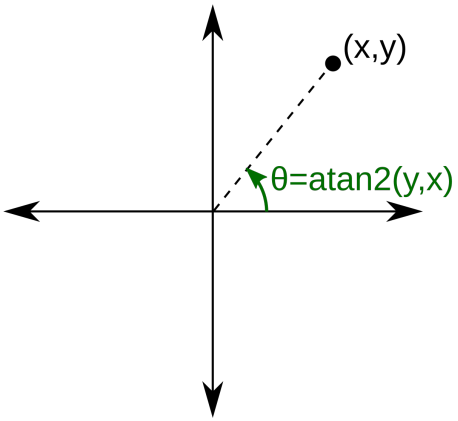


Figure 3.28: Visualization of the function $atan2$.

Algorithm 10 The function for UV mapping for the skydome.

```
1: function UV_MAP_FOR_SKYDOME(ray_dir)
2:     v = ray_dir.y * 0.5 + 0.5   ▷ The direction is normalized, y ∈ [−1, 1]
3:     u = 0.5 + (atan2(ray_dir.x, ray_dir.z)/(2 * PI))
4:     return u, v
```

3.6 Special Techniques For Lighting At Night

3.6.1 Increase Of Street Lamp Influence

Street lamps are small objects. Therefore, it is hardly possible to hit one and they have almost no influence.

To increase the influence of the street lamps, each of them has an auxiliary object: a collider. Each collider stores information about to which street lamp it belongs. If a ray hits a collider, the algorithm casts the ray again, but in

[39] Transformed so it is in range $[0, 1]$.

[40] See `https://en.wikipedia.org/wiki/Atan2`.

the direction towards the street lamp and ignoring the other colliders[41]. The colliders are placed slightly above the street lamps. Experiments showed that this position produces the best result. Figure 3.29 shows how the colliders would look if they were rendered: the black cubes. In the implementation, the size of the colliders is variable and the user can change it.

This method has two significant disadvantages. First, if an object has a mirror-like material, the direction of reflection is not random. As a result, the collider is reflected in this object. To prevent this artifact, the rays ignore the colliders if they are reflected from a mirror-like material. However, the street lamps don't have any influence again. The second, if a collider overlaps another object in the scene. In this case, all rays from the points on the overlapped part will be redirected to the street lamp. It results in a situation when there is a visible border between the overlapped part and the nonoverlapped part. They will be lit differently. The easiest way how to prevent this is to place all colliders properly so they overlap nothing.

Another way is to add some emissive energy to the glass sphere if the ray does not have a direct influence on a pixel[42]. Let us call it the *street lamp secondary emissive energy factor*. Thanks to it, the surrounding area of each street lamp is lit more. In the implementation, this parameter is variable and the user can adjust it.

Figure 3.30 shows all these methods in different variants: with the colliders, without them, with the secondary emissive energy factor, and without it.

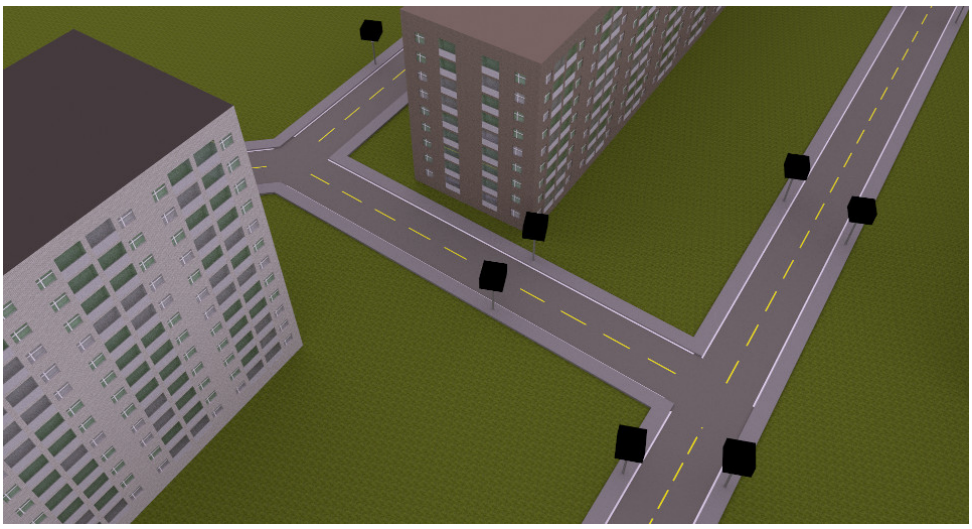
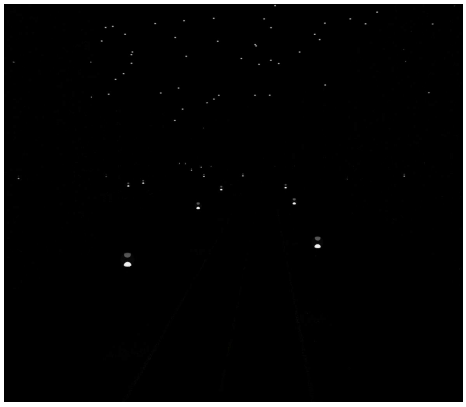


Figure 3.29: Figure showing rendered street lamp colliders.

[41] If the rays did not ignore the colliders after recasting, it would result in an infinite loop.

[42] Look at 2.4 to remember what it means.



(a) : No colliders, the street lamp emissive color is $[1.0, 1.0, 1.0]$, and the secondary emissive energy factor 0.0.

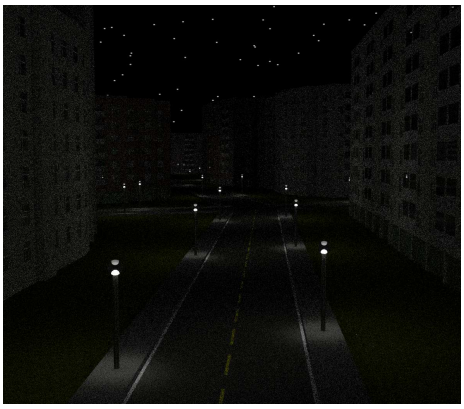
(b) : No colliders, the street lamp emissive color is $[2.0, 2.0, 2.0]$, and the secondary emissive energy factor 1.0.

(c) : Colliders, the street lamp emissive color is $[1.0, 1.0, 1.0]$, and the secondary emissive energy factor 0.0.

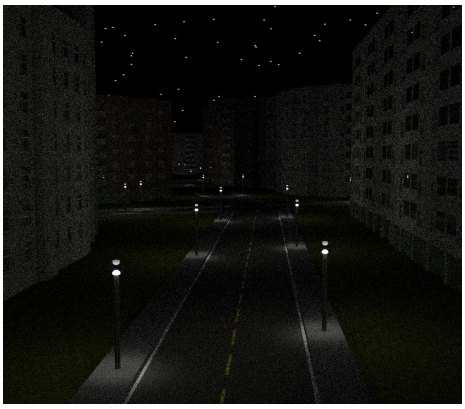
(d) : Colliders, the street lamp emissive color is $[1.5, 1.5, 1.5]$, and the secondary emissive energy factor 0.0.

(e) : Colliders, the street lamp emissive color is $[1.5, 1.5, 1.5]$, and the secondary emissive energy factor 0.2.

(f) : Colliders, the street lamp emissive color is $[1.5, 1.5, 1.5]$, and the secondary emissive energy factor 0.5.



(g) : Colliders, the street lamp emissive color is $[2.0, 2.0, 2.0]$, and the secondary emissive energy factor 2.0.



(h) : Colliders, the street lamp emissive color is $[3.0, 3.0, 3.0]$, and the secondary emissive energy factor 3.0.

Figure 3.30: Figure demonstrating the methods to increase the influence of street lamp lighting.

3.6.2 Emissive Color Of Windows

A window is a light source[43], but light intensity within its surface is not uniform. In some places it is brighter, in other places, it is darker. There is a great way how to simulate such a property - to use a texture of Perlin noise[44]. Article *Understanding Perlin Noise*[45] explains what Perlin noise is in a simple way. An example of such a texture is shown in figure 3.31. Figure 3.33 shows a render of a window.

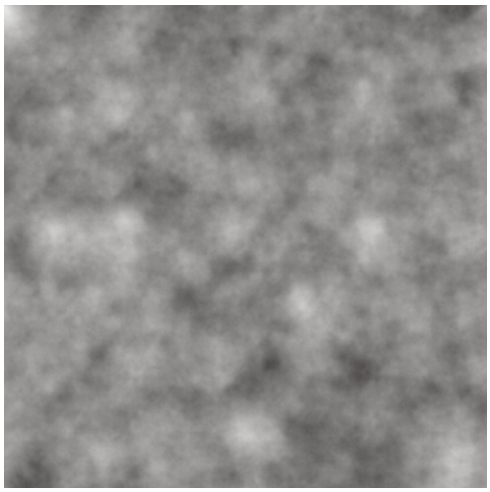


Figure 3.31: A Perlin noise texture.[46]

[43] In this project, room walls are light sources. Look at figure 3.19 to remember how the window geometry is created.

[44] See [34], [33].

[45] See [11].

[46] Created by `https://cpetry.github.io/TextureGenerator-Online/`.

Perlin noise functions give values in interval $[-1, 1]$. However, in this application, negative values are not possible for colors. Also, the user may want to have the right border of the interval greater than 1. Therefore, the Perlin noise values must be remapped to another interval $[p_{min}, p_{max}]$. The borders are variable in the implementation. However, if p_{max} is small, the windows are dim. If it is big, the scene is overlit, that is, there is too much light and the image does not look like a night. To avoid this problem, the Perlin noise value should be multiplied by value f, which is also a variable parameter in the implementation, if the ray does not have a direct influence on a pixel.[47] [48] Let us call it the *window emissive energy factor*. To make the described steps more understandable, they are summarized in figure 3.32. Figure 3.34 shows examples with different settings.

Because of the target of this project, Perlin noise textures are not stored in the memory and created in run time. Thanks to it, each window in the scene has a unique Perlin noise texture. The rendering program uses an open-source library *cudaNoise*[49] to generate these textures.

1. Call the Perlin noise function to get a value at the required position in the texture. This value is in interval $[-1, 1]$.
2. Remap the value to interval $[p_{min}, p_{max}]$.
3. If the ray does not have a direct influence on the pixel, multiply the value by window emissive energy factor f.

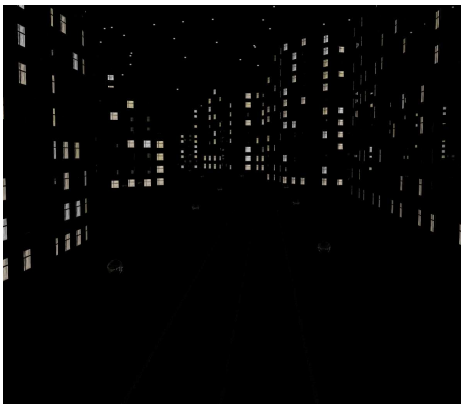
Figure 3.32: Steps value from the Perlin noise texture values modification.

Figure 3.33: Figure showing a window with a Perlin noise texture.

[47] Look at 2.4 to remember what it means.

[48] This multiplication may seem like a bug of the rendering algorithm, but actually, it is not so. In real life, a small bulb in the room makes the whole space lit. In the window, we see it. But in this project, the whole room is considered as a light source. It is much bigger than the bulb. So, it is necessary to somehow adjust the influence of the window lighting.

[49] See [22].



(a) : The Perlin noise values are in range $[0.0, 1.0]$ and the window emissive energy factor is 0.0.

(b) : The Perlin noise values are in range $[0.0, 1.0]$ and the window emissive energy factor is 1.0.

(c) : The Perlin noise values are in range $[0.0, 2.0]$ and the window emissive energy factor is 0.5.

(d) : The Perlin noise values are in range $[0.25, 2.5]$ and the window emissive energy factor is 0.1.

Figure 3.34: Figure showing renders with different settings for the windows.

Chapter 4

Results

This chapter provides measurements of memory consumption and the rendering time for each geometry representation, measurements of how the Perlin noise texture generation and the road texturing influence the rendering time, how the path tracing parameters influence it and the image. The project was developed and tested with the graphics card Nvidia GTX 1070[1], which has 8GB of memory. The rendering application was tested on the scenes listed in table 4.1.

№	Name	Size	Buildings	Road pieces	Street lamps
1.	Prague Zličín	958 m x 672 m	49	117	227
2.	Belgrade	1736 m x 2313 m	570	373	420
3.	New York	2230 m x 1497 m	778	316	514
4.	Berlin	2123 m x 1507 m	672	457	1406
5.	Krasnoyarsk	4922 m x 3668 m	1384	1981	4869

Table 4.1: Scenes used for tests.

These scenes have a fixed number of objects, but it is possible to use various ranges of building heights in them. That allows making more different comparisons. Figure 4.1 shows the number of triangles in each scene for the average building height 40 m and 80 m to help to understand how complex the scenes are.

The sections below provide information about the used scenes and parameters. Unless otherwise stated, there was used **1 ray per pixel**, the **depth of Russian Roulette** was **3** and its **probability** was **0.5**.

[1] `https://www.nvidia.com/en-gb/geforce/graphics-cards/geforce-gtx-1070/specifications/`

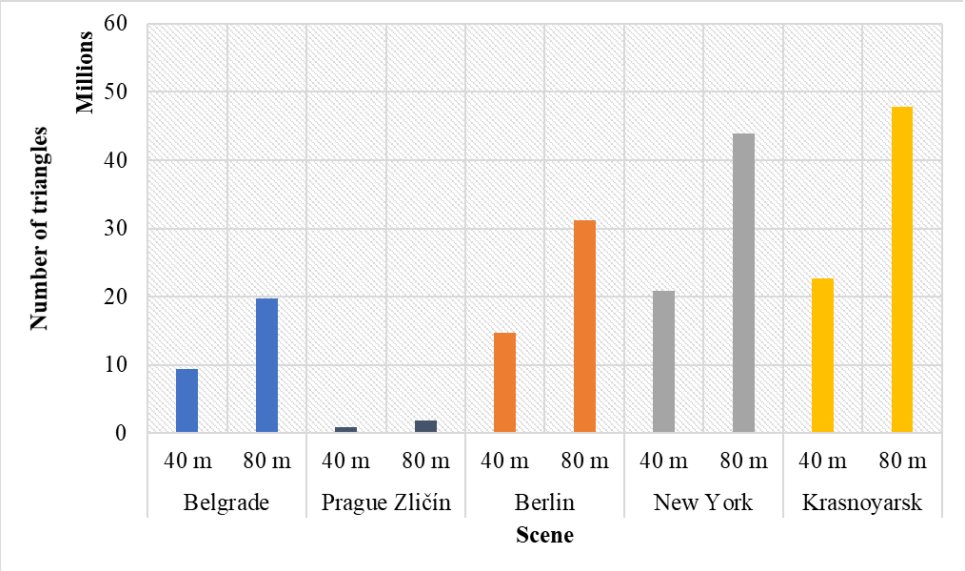


Figure 4.1: The number of triangles in each scene for the average building height 40 m and 80 m.

4.1 Memory Consumption

This section compares how memory consumption differs when using the traditional geometry representation or the procedural one.

The more the average building height is, the more vertices the buildings have. Hence, they require more memory. The amount of memory required for procedural geometry generation does not depend on the average building height. Thanks to it, it is constant for whatever value. The differences in the approximated amounts of memory required for each geometry representation are significant: the procedural geometry representation may require slightly more than 1 MB, but the traditional one may require thousands of MB. In some tests, it was impossible to store the geometry in the memory. If the geometry was created procedurally, there was not such a problem.

Belgrade

Figure 4.3 shows how the number of triangles depends on the average building height in *Belgrade*, and figure 4.2 shows an approximation of the required amount of memory for this scene.

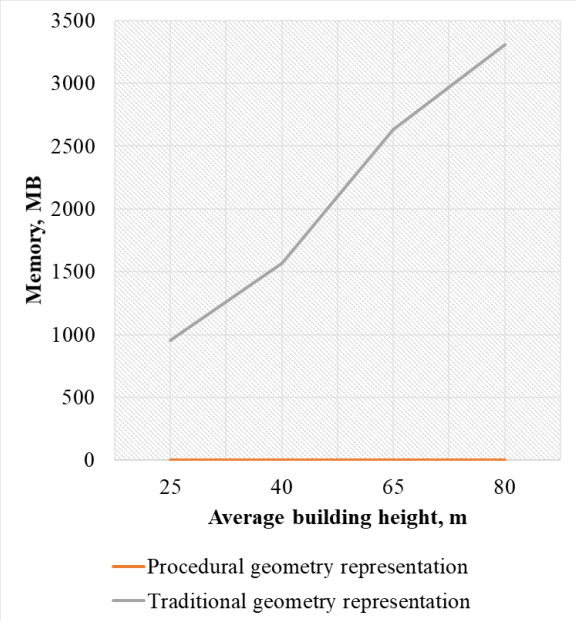


Figure 4.2: Approximation of the required amount of memory in *Belgrade* for each geometry representation with various average building heights.

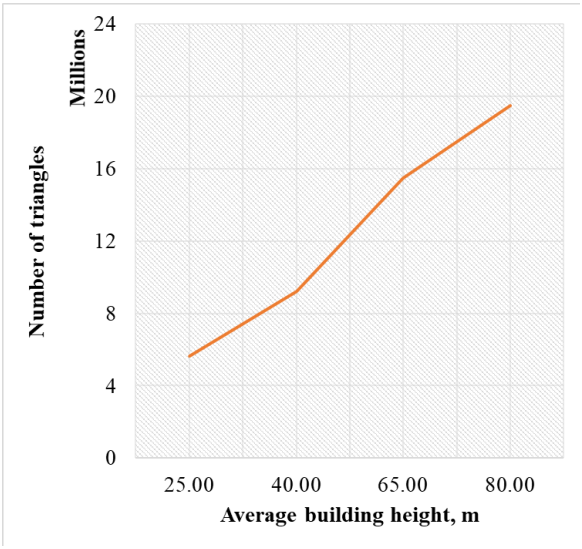


Figure 4.3: Dependence of the number of triangles on the average building height in *Belgrade*.

New York

In this test in *New York*, the average height was 180 m. The scene contains a big number of large buildings. Due to these reasons, the geometry required a greater amount of memory than the graphics card had. So, it was impossible to perform tests with the traditional geometry representation and the tests were with the procedural one. Since the amount of required memory for procedural geometry representation does not depend on the average building height, it does not make sense to perform measurements for various values. Figure 4.4 shows an approximation of the amount of memory required to store the scene objects.

Prague Zličín

Figure 4.5 shows an approximation of the required amount of memory for the scene *Prague Zličín*. The results are similar to *Belgrade*. However, the amount of memory required to store the geometry is less, because the number of buildings is less.

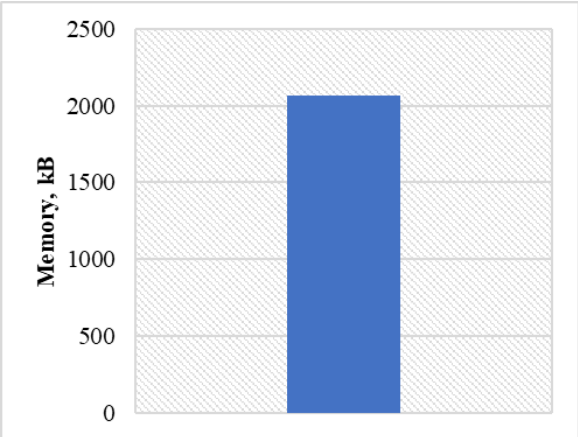


Figure 4.4: Approximation of the amount of memory required to store the scene objects in *New York* when the geometry is created procedurally.

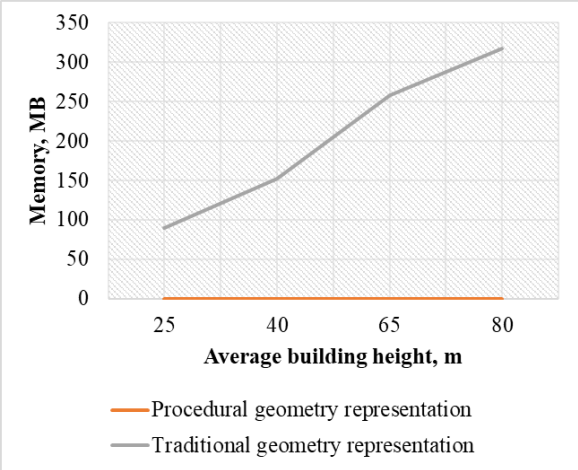
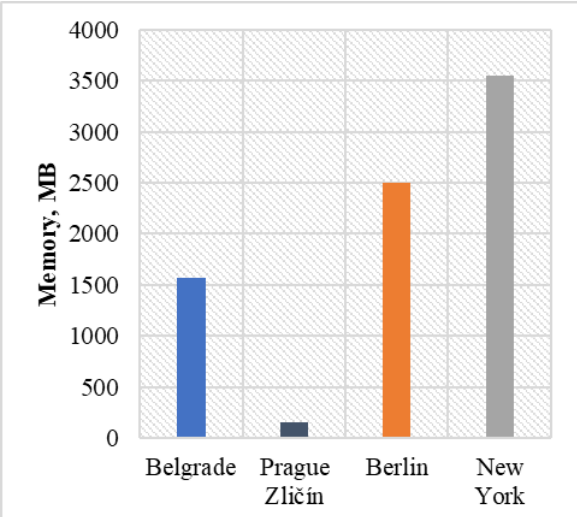


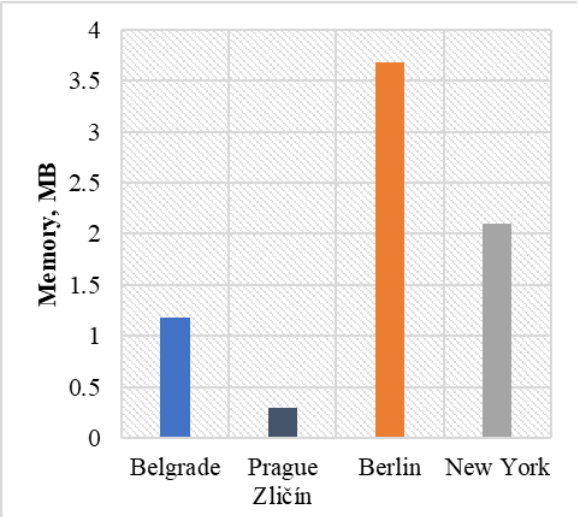
Figure 4.5: Approximation of the required amount of memory in *Prague Zličín* for each geometry representation with various average building heights.

Memory Consumption In Several Scenes

This paragraph provides charts of used memory in several scenes for each geometry representation. The left part of figure 4.6 shows data for the traditional geometry representation and the right part shows data for procedural. With the procedural geometry representation, used memory in *Berlin* is more than in *New York*. However, with the traditional geometry representation, the difference was the opposite. It was caused by a higher number of street lamps.



(a) : Traditional geometry representation.

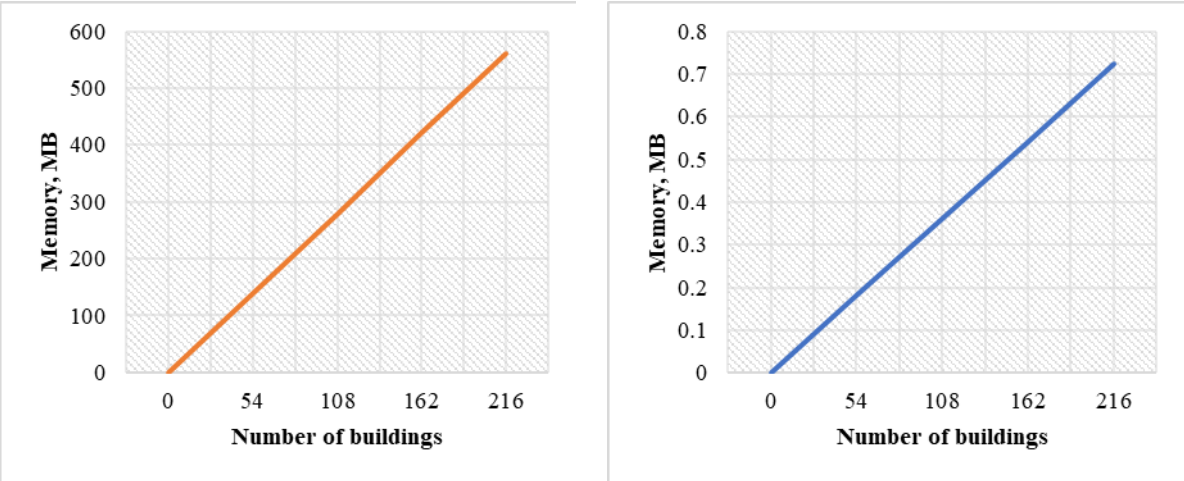


(b) : Procedural geometry representation.

Figure 4.6: Used memory in several scenes with the average building height 40 m for each geometry representation.

Dependence Of Memory Consumption On The Number Of Buildings

The more the number of buildings is, the more they require regardless of the geometry representation. The dependencies of the amount of used memory on the number of buildings for each geometry representation with the average building height 40 m are demonstrated in figure 4.7[2].



(a) : Traditional geometry representation.

(b) : Procedural geometry representation.

Figure 4.7: Dependence of memory consumption on the number of buildings for each geometry representation.

4.2 Rendering Time Dependence On Geometry Representation

This section provides comparisons of how the rendering time depends on the geometry representation. In contrast to memory consumption, procedural generation requires significantly much more time to render a frame. Again, there is a dependence on the average building height, because it influences how many vertices are stored in the memory and how many primitives have to be created at run time.

Belgrade

In the first test *Belgrade*, the average building height was 27 m. The geometry was stored in the memory, not created procedurally. It was performed from 3 views from different positions which are shown in figure 4.10. Figure 4.8 shows how the rendering time differs for both the geometry representations from the mentioned views.

[2]The amounts of used memory for a different number of buildings were measured. Then, the lines to show the dependencies were found by the method of least squares.

In the second test, there was measured how the rendering time depends on the average building height for each geometry representation. For both of them, the dependence looks sublinear. Figure 4.9 shows graphs for this test.

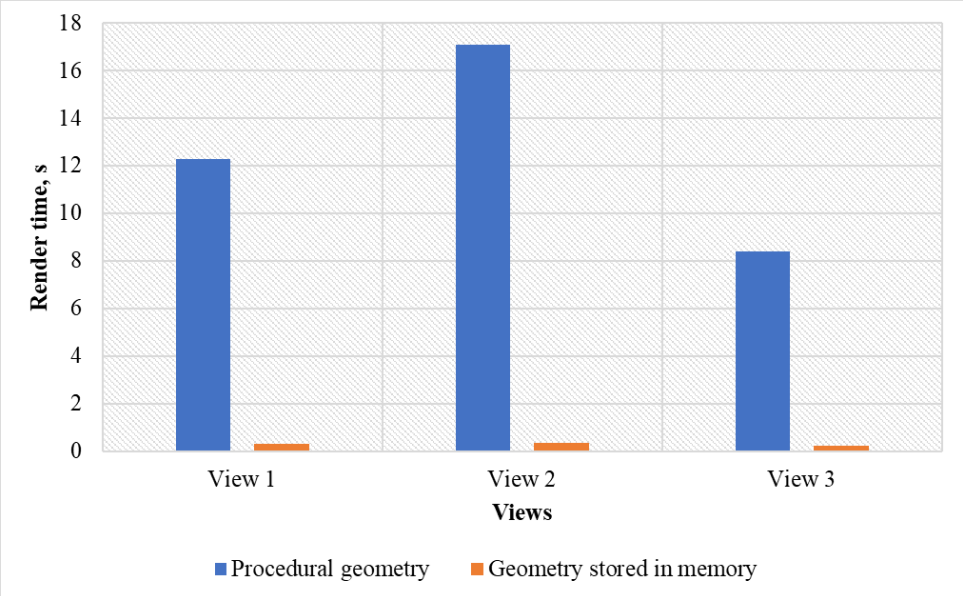
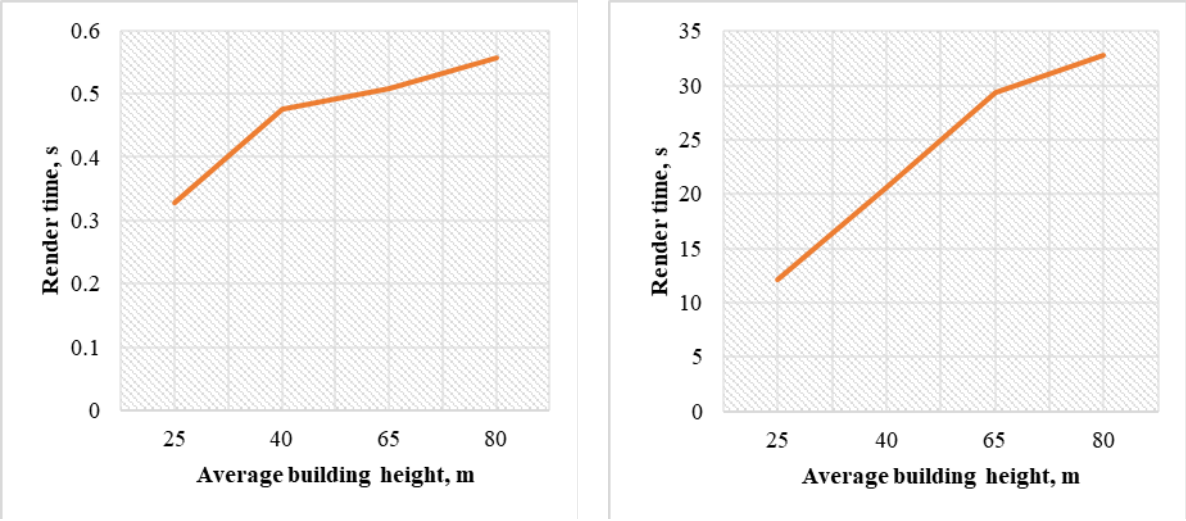


Figure 4.8: Comparison of the rendering time for two geometry representations in *Belgrade* with views from different positions.



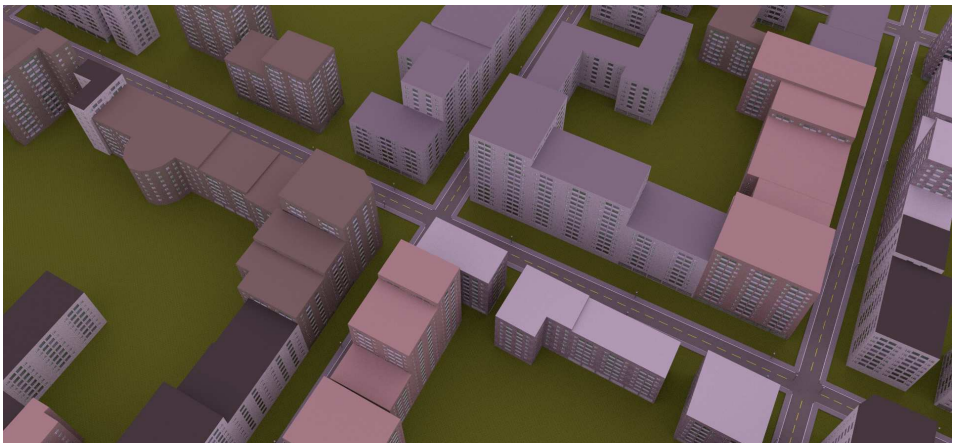
(a) : Traditional geometry representation.

(b) : Procedural geometry representation.

Figure 4.9: Dependence of the rendering time on the average building height for each geometry representation in *Belgrade*.

(a) : View 1.

(b) : View 2.



(c) : View 3.

Figure 4.10: Views in *Belgrade*.

4.3 Rendering Time Dependence On Perlin Noise Texture Generation

Since Perlin noise textures are created at run time for each window, it is obvious that it influences the rendering time. When a window has lights off, no texture is created. Hence, the more windows have lights on, the longer the rendering time. The more the average building height is, the more windows the scene has. The tests below show that the generation of a Perlin noise texture has a small influence on the rendering time. In cases with the procedural geometry representation, it has no noticeable influence, because the generation of the geometry requires much time.

Belgrade

The test was performed on two variants of the scene *Belgrade*: with the average building height 30 m and 61 m. The first variant had 92716 windows,

the second had 195431 windows. The geometry was stored in the memory. The camera was in the same position and had the same direction. Since the second variant had more vertices, the rendering time there was greater than in the first case. The dependencies in the tests are shown in figure 4.11. They look similar. However, in the first variant, the increase was 27% of the rendering time, but in the second variant, it was 22%. Figure 4.12 compares how these variants differ visually. Figure 4.13 shows how the first variant looks like with a different number of windows with lights on.

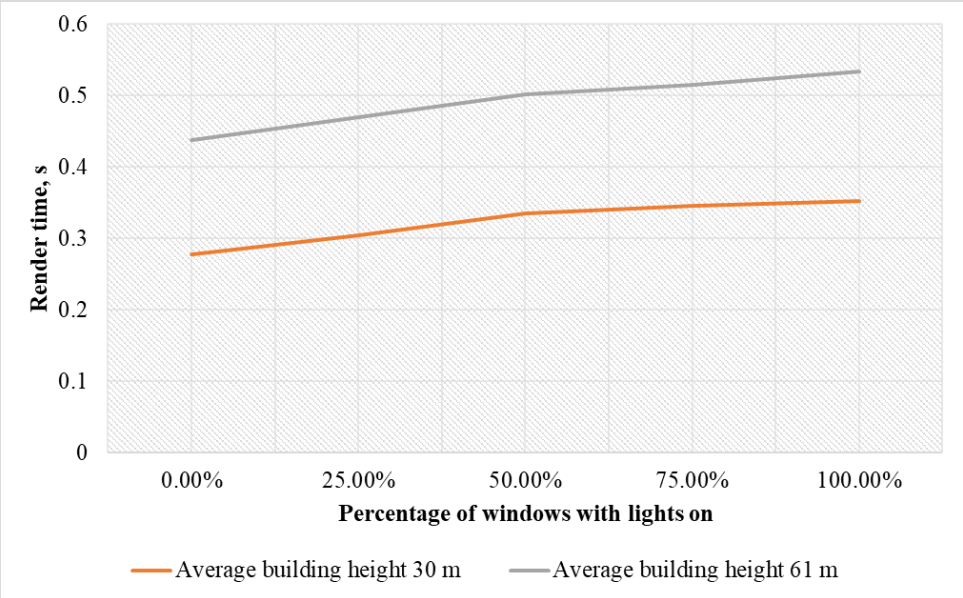
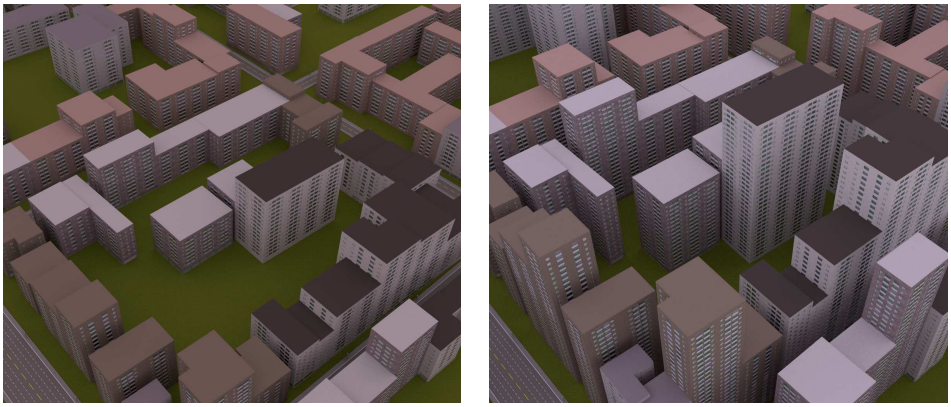


Figure 4.11: Dependence of the rendering time on the number of windows with lights on in *Belgrade*.

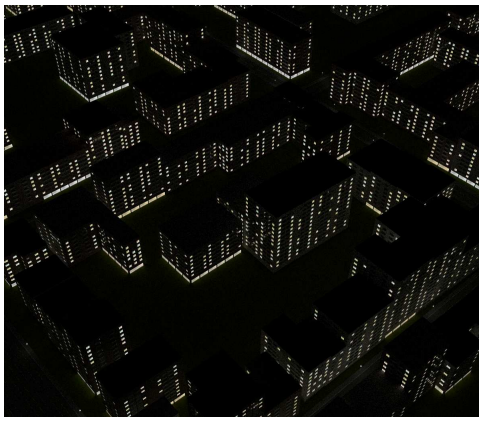


(a) : Average building height 30 m. **(b) :** Average building height 61 m

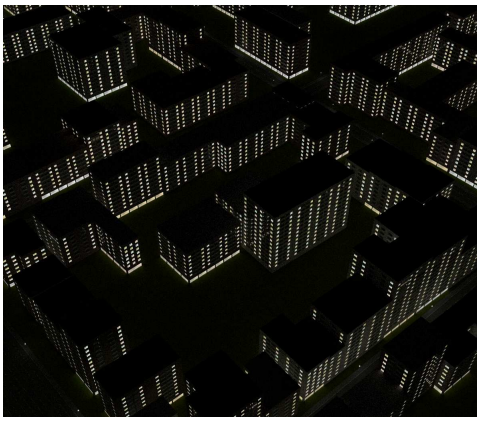
Figure 4.12: Visual comparison of two variants of *Belgrade* used for the test.

(a) : 25% of windows has lights on.

(b) : 50% of windows has lights on.



(c) : 75% of windows has lights on.



(d) : 100% of windows has lights on.

Figure 4.13: Visual comparison of a different number of windows with lights on in *Belgrade* with the average building height 30 m.

New York

In this test in *New York*, the average building height was 180 m. The test was performed with the procedural geometry representation. Since the scene contains a big number of buildings and they are large and tall, there is much geometry to create and the creation takes a lot of time. Figure 4.14 shows how the rendering time depends on the number of windows with lights on. The rendering time in the case with no window with lights on is greater than in the case with all windows with lights on. That means the creation of a Perlin noise texture has an imperceptible influence on the rendering time. The generation of the geometry takes more time. Thus, the generation of the Perlin noise texture is not visible in the graph.

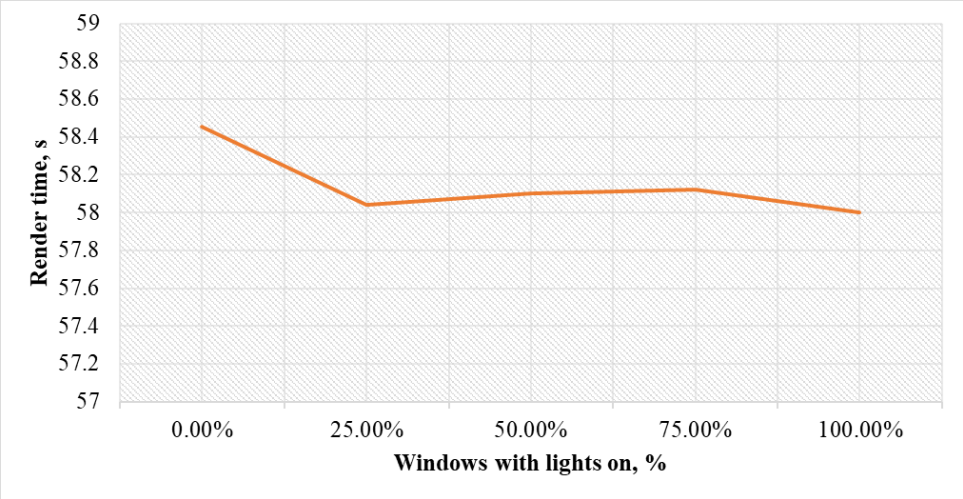
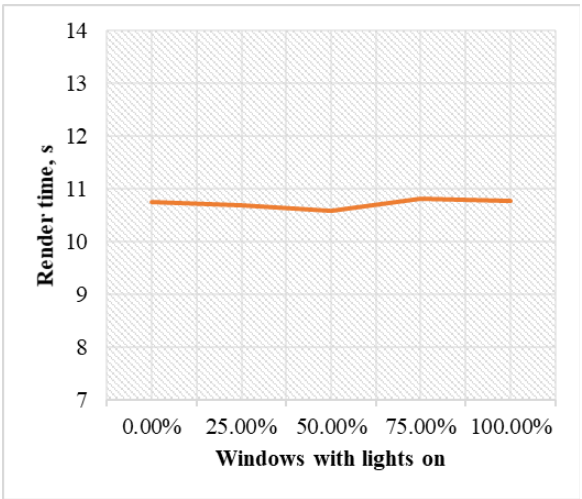


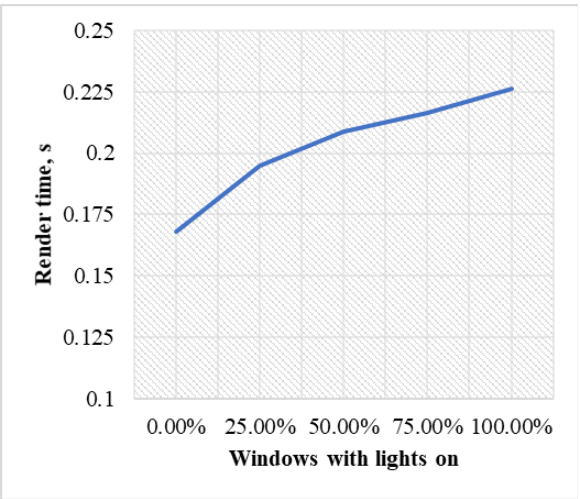
Figure 4.14: Dependence of the rendering time on the number of windows with lights on in *New York*.

Berlin

In the scene *Berlin*, there were performed tests for each geometry representation. In both, the average building height was 27 m. The dependence for each geometry representation is shown in figure 4.15. In the first case, the graph is similar to the graph in the test in *New York*, in which the procedural geometry representation was used. In the second case, the graph is similar to the graph in the test in *Belgrade*, in which the traditional geometry representation was used.



(a) : Procedural geometry representation.



(b) : Traditional geometry representation.

Figure 4.15: Dependence of the rendering time on the number of windows with lights on in *Berlin* for each geometry representation.

4.4 Rendering Time Dependence On Road Texturing

Since the textures for the roads are created at run time, it is likely that it influences the rendering time. This section provides measurements of how the rendering time differs. They shows that it is not increased too much. The road texturing has a small influence on it[3].

Berlin

This test was performed in *Berlin*. The difference in the rendering time is shown in figure 4.16. The test was done from the view which is demonstrated in figure 4.17.

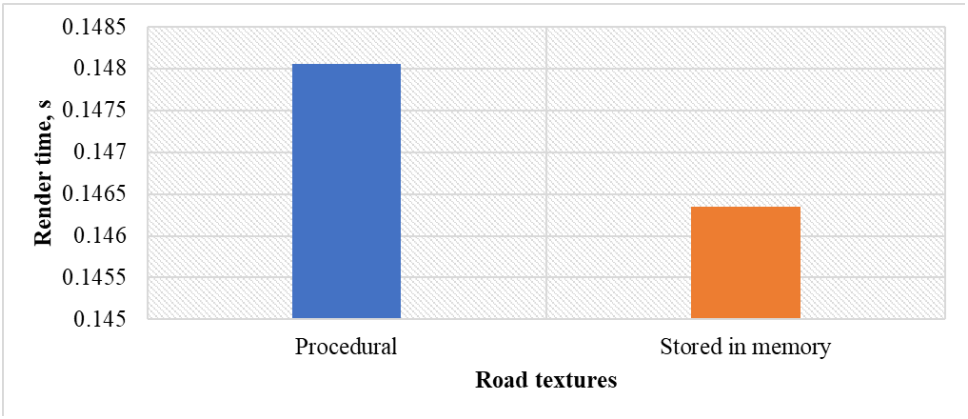


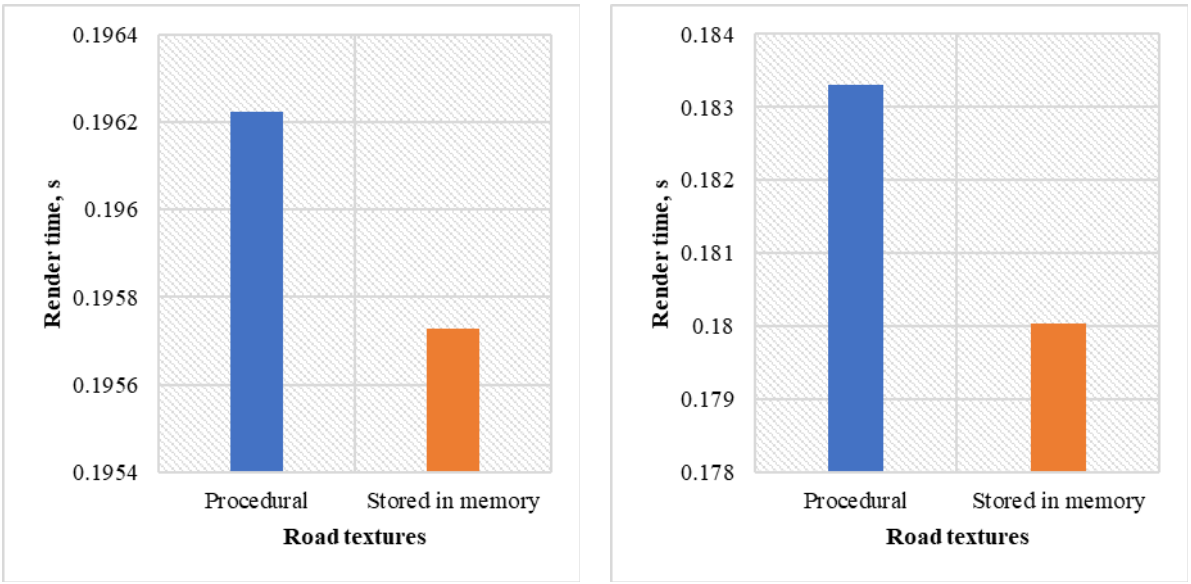
Figure 4.16: Rendering time difference between the procedural road texturing and the use of textures stored in the memory in *Berlin*.

Figure 4.17: View used in the test with the road texturing in *Berlin*.

[3]In this section, I mention textures stored in the memory. Actually, the program does not support it. I suppose that the use of the textures stored in the memory has constant complexity and it is approximately equal to the use of a uniform color.

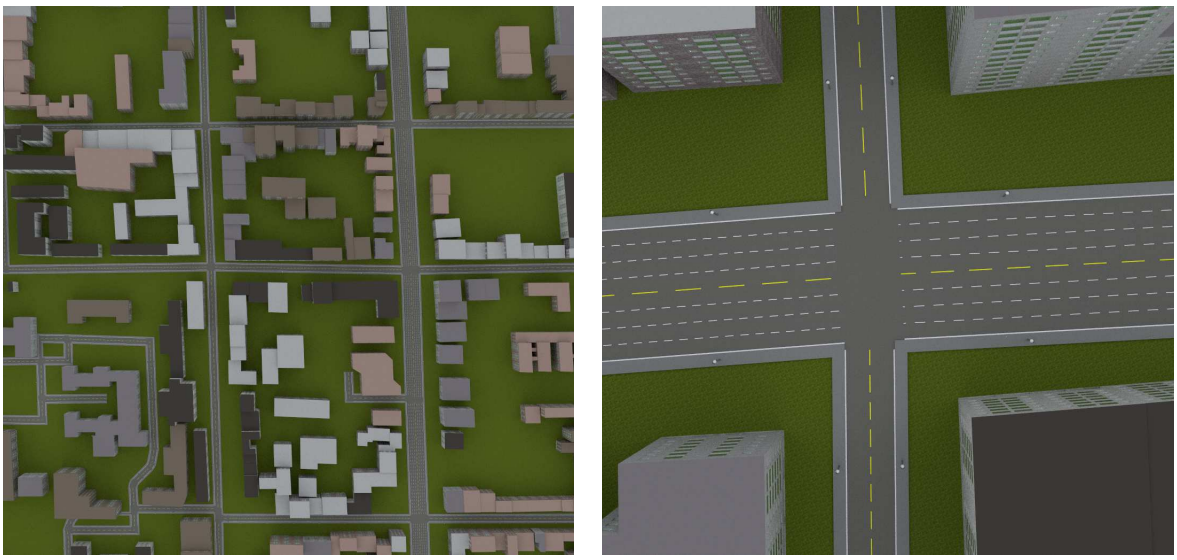
Belgrade

In the scene *Belgrade*, two tests were performed. The results of them are shown in figure 4.18. Figure 4.19 shows the used views.



(a) : View 1. **(b) :** View 2.

Figure 4.18: Rendering time difference between the procedural road texturing and the use of textures stored in the memory in *Belgrade*.



(a) : View 1. **(b) :** View 2.

Figure 4.19: Views used in the tests of road texturing in *Belgrade*.

Prague Zličín

This test was performed in the scene *Prague Zličín*. The difference in the rendering time is shown in figure 4.20. The test was done from the view which is demonstrated in figure 4.21. In this case, the procedural road texturing took less time. However, the difference is insignificant.

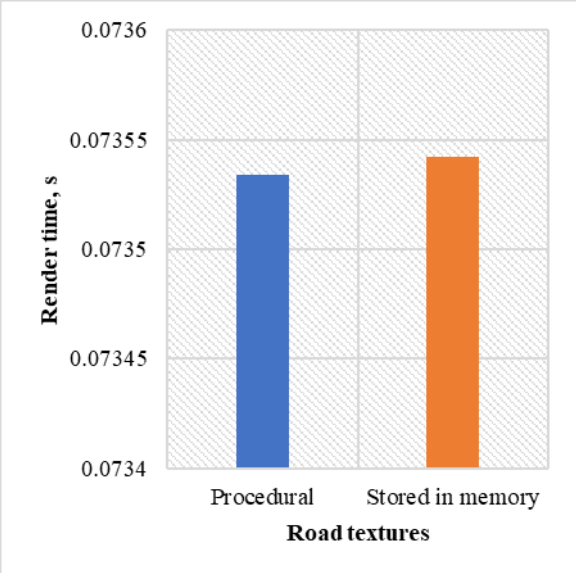


Figure 4.20: Rendering time difference between the procedural road texturing and the use of textures stored in the memory in *Prague Zličín*.

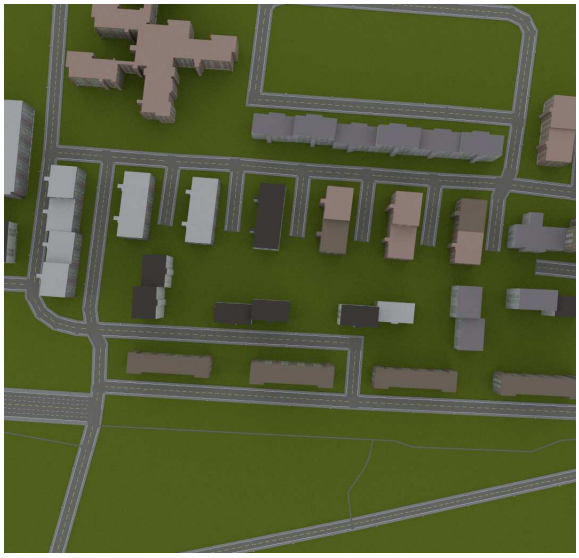


Figure 4.21: View used in the test of road texturing in *Prague Zličín*.

4.5 Influence Of Street Lamp Colliders On Rendering Time

This section provides measurements of how the use of the street lamp colliders influences the rendering time. The performed tests showed that it is increased slightly.

Berlin

In the test in *Berlin*, the average building height was 27 m. The test was performed from the view shown in figure 4.23. The results are featured in figure 4.22.

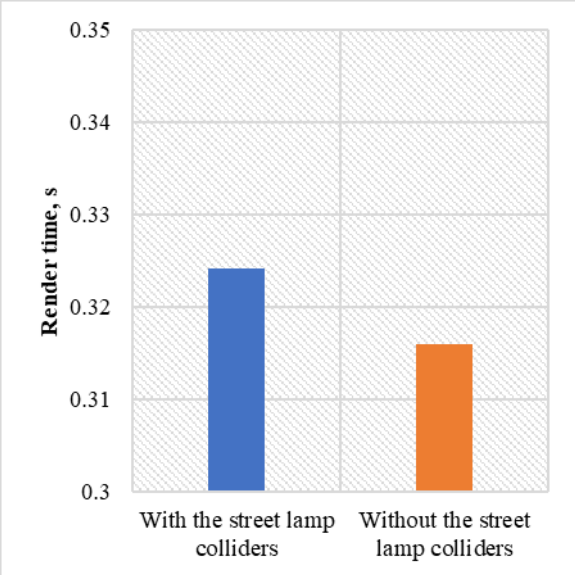


Figure 4.22: Rendering time difference with the street lamp colliders and without them in *Berlin*.

Figure 4.23: View for the test of the street lamp colliders in *Berlin*.

Krasnoyarsk

In the test in *Krasnoyarsk*, the average building height was 35 m. The view for this test is shown in figure 4.25. The results are featured in figure 4.24.

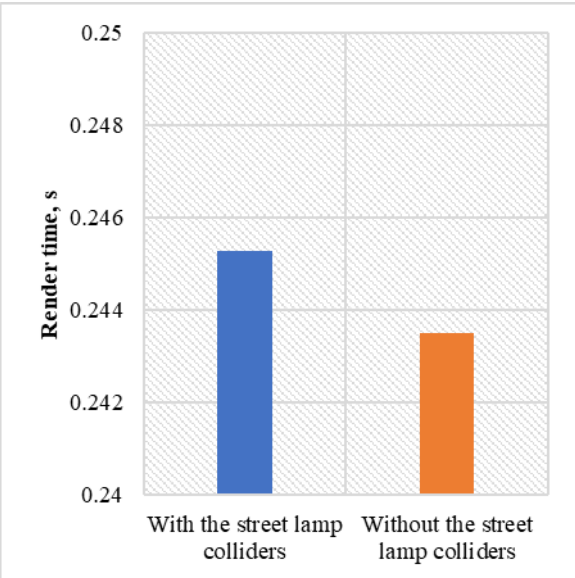


Figure 4.24: Rendering time difference with the street lamp colliders and without them in *Krasnoyarsk*.

Figure 4.25: View for the test with the street lamp colliders in *Krasnoyarsk*.

4.6 Rendering Time Dependence On Path Tracing Parameters

Such path tracing parameters as the number of samples per pixel, the Russian Roulette depth, and the Russian Roulette probability influence the image and the rendering time. This section provides measurements and tests with them.

4.6.1 Number Of Samples Per Pixel

This section shows how the rendering time depends on the number of samples per pixel. As expected, the greater the number is, the greater the rendering time is.

Belgrade

In the scene *Belgrade*, the average building height was 27 m, the geometry was stored in the memory. Figure 4.26 shows the dependence in this scene variant. Figure 4.27 shows the view from which the test was performed.

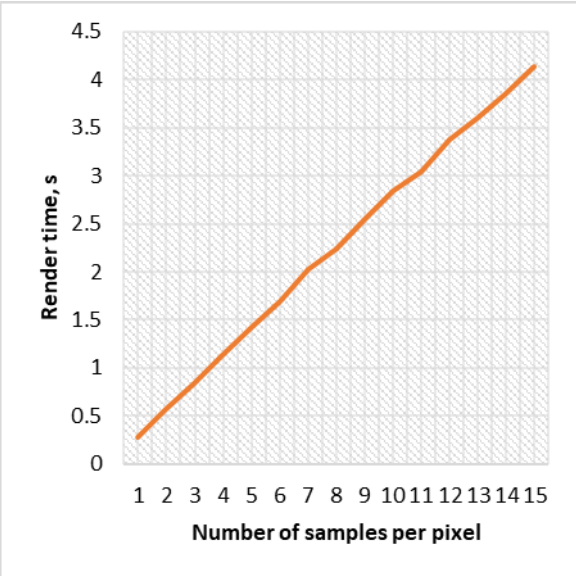


Figure 4.26: Dependence of the rendering time on the number of samples in *Belgrade*.

Figure 4.27: View in *Belgrade*.

4.6.2 Russian Roulette Depth

This section shows how the rendering time depends on the Russian Roulette depth. As with the number of samples per pixel, the greater the depth is, the greater the rendering time is.

Belgrade

In this test in *Belgrade*, there was used the same scene variant and the same view as in 4.6.1. The results are shown in figure 4.28.

4.6.3 Russian Roulette Probability

This section shows how the rendering time depends on the Russian Roulette probability. The lower the probability is, the greater the rendering time is.

Belgrade

Again, in this text in *Belgrade*, there was used the same scene variant and the same view as in 4.6.1. The results are shown in figure 4.29.

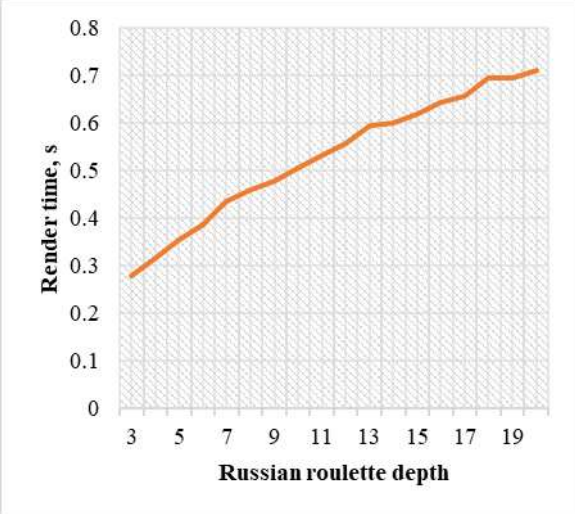


Figure 4.28: Dependence of the rendering time on the Russian Roulette depth in *Belgrade*.

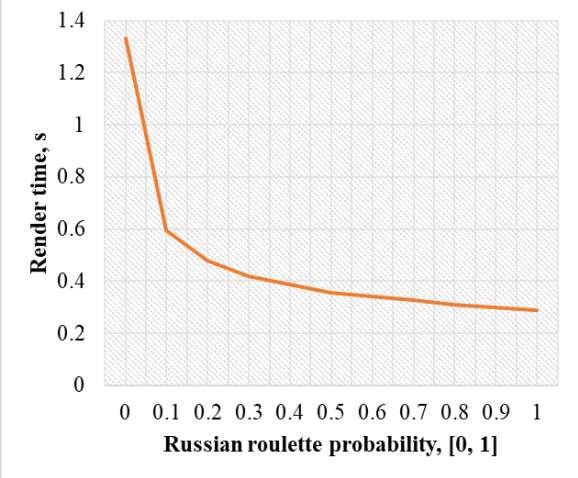


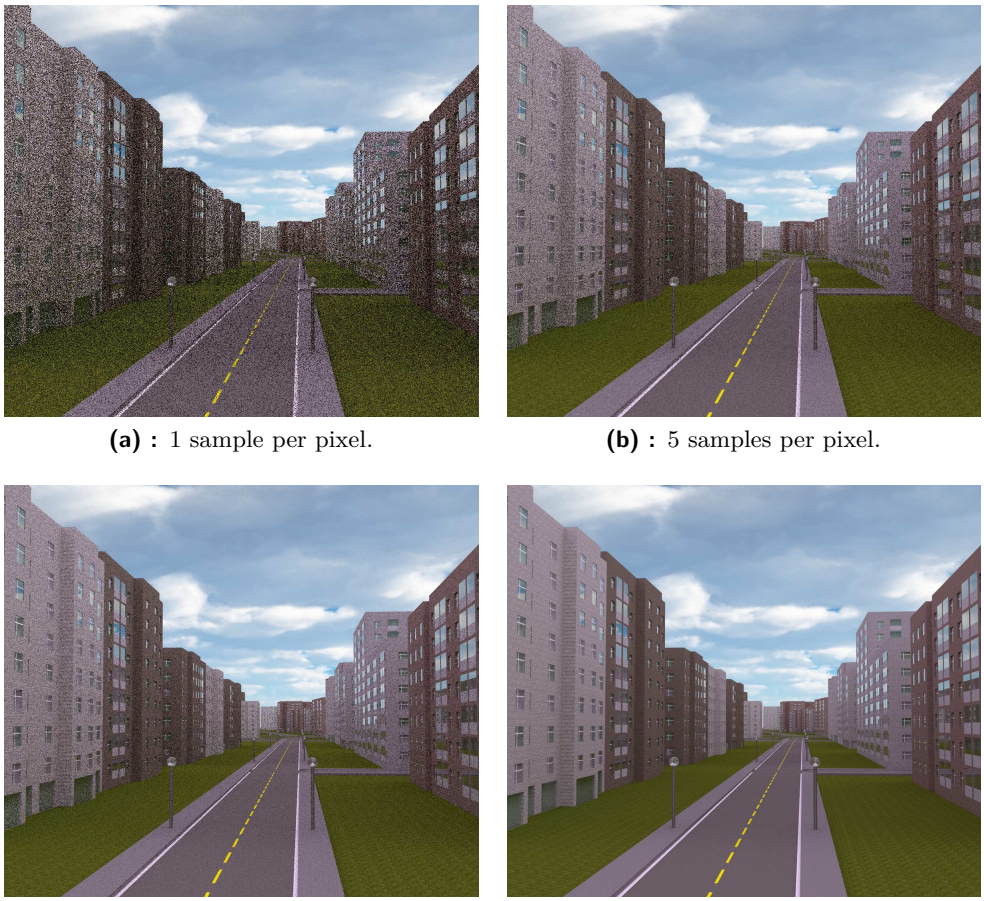
Figure 4.29: Dependence of the rendering time on the Russian Roulette probability in *Belgrade*.

4.7 Image Dependence On Path Tracing Parameters

This section provides tests of the image dependence on the number of samples per pixel, the Russian Roulette depth, and the Russian Roulette probability.

4.7.1 Number Of Samples Per Pixel

The number of samples per pixel influences how noisy the picture is. Figure 4.30 compares images with 1 sample, 5 samples, 10 samples, and 100 samples.



(a) : 1 sample per pixel.

(b) : 5 samples per pixel.

(c) : 10 samples per pixel.

(d) : 100 samples per pixel.

Figure 4.30: Visual comparison of a different number of samples.

4.7.2 Russian Roulette Depth

The Russian Roulette depth has a big influence on the amount of noise. Especially, it is more visible on the glass material. Figure 4.31 compares images with a different Russian Roulette depth. In the first image, the depth was 3. In the second, it was 30. Also, the number of samples per pixel was 5 and the images were not denoised in any other way.

4.7.3 Russian Roulette Probability

The Russian Roulette probability also has an influence on the amount of noise, which is better visible on the glass material. Figure 4.32 shows how the images differ with a different Russian Roulette probability.

(a) : The Russian Roulette depth is 3.

(b) : The Russian Roulette depth is 30.

Figure 4.31: Visual comparison of a different Russian Roulette depth.

(a) : The Russian Roulette probability 0.01.

(b) : The Russian Roulette probability 0.5.

(c) : The Russian Roulette probability 0.99.

Figure 4.32: Visual comparison of a different Russian Roulette probability.

4.8 Comparison With Photos

This section compares renders from the application with real photos. It shows that the rendering application is able to create pretty realistic models of cities and simulate lighting effects well.

Figure 4.33: Render.

Figure 4.34: Real photo[4].

[4]Taken from `https://mapio.net/pic/p-44578244/`.

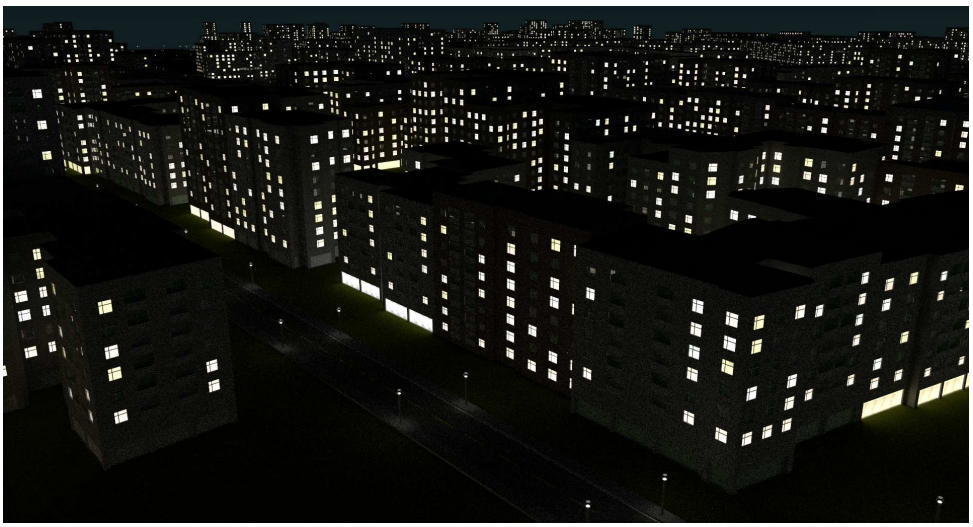


Figure 4.35: Render.

Figure 4.36: Real photo[5].

[5]Taken from `https://wallpaperscraft.ru/download/nochnoj_gorod_ogni_goroda_vid_sverhu_137646/8256x5504`.

Figure 4.37: Render.

Figure 4.38: Real photo[6].

[6]Taken from `https://club.foto.ru/gallery/photos/photo.php?photo_id=1670749`.

Chapter 5

Conclusion

I developed an algorithm of procedural city generation on demand and a rendering algorithm with a focus on lighting at night. For them, I implemented a rendering application. In addition to this, I described a way of OpenStreetMap data processing and created a program for this purpose.

The rendering application is able to render beautiful and pretty realistic images. In terms of memory, the procedural geometry representation wins the traditional one with a huge advantage. When the traditional geometry representation may use all graphics card memory, the procedural one requires units of MB. However, in terms of the rendering time, it completely loses. The rendering time may be more than a minute, but for the traditional geometry representation, it is mostly less than one second.[1] This project is little optimized. Much redundant geometry is created. It increases the rendering time. It is likely that the rendering time might be significantly reduced if the program was better optimized. In contrast to the geometry representation, procedural texturing turned out to be a very useful and efficient technique. It does not consume memory and it does not significantly increase the rendering time. Also, the used techniques for lighting turned out to be very efficient.

For the mentioned reasons, procedural geometry generation at run time on demand seems to be not the best approach. Much better could be an approach in which the geometry of the buildings close to the camera was cached and in which the program could intelligently determine the building part for which the geometry has to be created. However, as the performance of computers rapidly increases, it may turn out that this program will be able to be used in real time.

If it was possible to use such a project in real time, it may perfectly fit into game development, because procedural generation may save years of work. Such a project may be well suitable for flight simulators. There is a number of flight simulators that offer the users to fly around the whole planet. The most popular of them are X-Plane, Prepar3D, and Microsoft Flight Simulator. It is not possible to manually create all cities that exist in the world. Therefore, they are created procedurally[2]. The more detailed

[1] If there is 1 sample per pixel, the Russian Roulette depth is 3, and its probability is 0.5.

[2] See [19], [1], [43], [44]

a prepared object is and the more objects the game has, the more memory and the more data movements it requires. These simulators are famous for their large size on the disk. Moreover, Microsoft Flight Simulator downloads some data at run time to reduce memory consumption. It would be a perfect solution to use the technique from this project. In addition, the users of flight simulators fancy flying at night. Good-looking night cities, which are a result of this work, would be a perfect feature. On the other hand, it does not have to be used in real time only and it may be used to render some animations.

The knowledge received during the development may be useful in various cases. It may be used as an idea of how to make lighting in the scene better, as a study material for OptiX API, because there is a lack of open-source projects, and for how to create cities using OpenStreetMap.

Bibliography

[1] Blackshark.ai. https://blackshark.ai. Accessed: 2021-05-18.

[2] Blender. https://www.blender.org. Accessed: 2021-05-18.

[3] Digital pictures of all sorts of materials. https://www.textures.com. Accessed: 2021-05-18.

[4] OpenStreetMap. https://www.openstreetmap.org. Accessed: 2021-05-18.

[5] OsmSharp. http://www.osmsharp.com. Accessed: 2021-05-18.

[6] Scratchapixel - Learn Computer Graphics From Scratch! https://www.scratchapixel.com. Accessed: 2021-05-18.

[7] Unity. https://unity.com. Accessed: 2021-05-18.

[8] Avionx. Skybox Series Free. https://assetstore.unity.com/packages/2d/textures-materials/sky/skybox-series-free-103633. Accessed: 2021-05-20.

[9] Beason, K. smallpt: Global Illumination in 99 lines of C++. https://www.kevinbeason.com/smallpt/.

[10] Beneš, B., Šťava, O., Měch, R., and Miller, G. Guided procedural modeling. *Computer Graphics Forum 30*, 2 (Apr. 2011), 325–334.

[11] Biagioli, A. Understanding Perlin Noise. https://adrianb.io/2014/08/09/perlinnoise.html. Accessed: 2021-05-18.

[12] Boechat, P. Procedural City. https://github.com/pboechat/ProceduralCity. Accessed: 2021-05-20.

[13] de Vries, J. LearnOpenGL. https://learnopengl.com.

[14] Esri. ArcGIS CityEngine. https://www.esri.com/en-us/arcgis/products/arcgis-cityengine/overview. Accessed: 2021-05-18.

[15] Hoffman, N. Background: Physics and Math of Shading.

[16] Kajiya, J. T. The Rendering Equation. *SIGGRAPH Comput. Graph. 20*, 4 (Aug. 1986), 143–150.

[17] Khan Academy. Perlin noise. https://www.khanacademy.org/computing/computer-programming/programming-natural-simulations/programming-noise/a/perlin-noise. Accessed: 2021-05-20.

[18] Kravtchenko, I. HDR Image Reader. https://www.flipcode.com/archives/HDR_Image_Reader.shtml. Accessed: 2021-05-20.

[19] Lardinois, F. Meet the startup that helped Microsoft build the world of Flight Simulator. https://techcrunch.com/2020/08/17/meet-the-startup-that-helped-microsoft-build-the-world-of-flight-simulator/, 2020. Accessed: 2021-05-18.

[20] Larive, M., and Gaildrat, V. Wall Grammar for Building Generation. In *Proceedings of the 4th International Conference on Computer Graphics and Interactive Techniques in Australasia and Southeast Asia* (New York, NY, USA, 2006), GRAPHITE '06, Association for Computing Machinery, p. 429–437.

[21] Lawrence, J. Importance Sampling of the Phong Reflectance Model. https://www.cs.princeton.edu/courses/archive/fall16/cos526/papers/importance.pdf.

[22] Lehtinen, H. cudaNoise. https://github.com/covexp/cuda-noise. Accessed: 2021-05-18.

[23] Li, V. Raytracing - Rendering Equation Insight. http://viclw17.github.io/2018/06/30/raytracing-rendering-equation/, 2018. Accessed: 2021-05-18.

[24] Möller, T., and Trumbore, B. Fast, Minimum Storage Ray-Triangle Intersection. *Journal of Graphics Tools 2*, 1 (1997), 21–28.

[25] Nishida, G., Garcia-Dorado, I., Aliaga, D. G., Benes, B., and Bousseau, A. Interactive sketching of urban procedural models. *ACM Transactions on Graphics (TOG) 35*, 4 (2016), 1–11.

[26] Nvidia. OptiX AI-Accelerated Denoiser. https://developer.nvidia.com/optix-denoiser.

[27] Nvidia. OptiX Ray Tracing Engine. https://developer.nvidia.com/optix.

[28] Nvidia. Ray Tracing. https://developer.nvidia.com/discover/ray-tracing.

[29] Nvidia. What's the Difference Between Ray Tracing and Rasterization? https://developer.nvidia.com/optix.

[30] OTOY. OctaneRender. https://home.otoy.com/render/octane-render/.

[31] Over, M., Schilling, A., Neubauer, S., and Zipf, A. Generating web-based 3d city models from openstreetmap: The current situation in germany. *Computers, Environment and Urban Systems 34*, 6 (2010), 496–507. GeoVisualization and the Digital City.

[32] Parker, S. G., Bigler, J., Dietrich, A., Friedrich, H., Hoberock, J., Luebke, D., McAllister, D., McGuire, M., Morley, K., Robison, A., and Stich, M. Optix: A general purpose ray tracing engine. *ACM Transactions on Graphics* (August 2010).

[33] Perlin, K. Improved Noise reference implementation. https://mrl.cs.nyu.edu/~perlin/noise/. Accessed: 2021-05-18.

[34] Perlin, K. Improving noise. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 2002), SIGGRAPH '02, Association for Computing Machinery, p. 681–682.

[35] Pharr, M., Jakob, W., and Humphreys, G. *Physically Based Rendering: From Theory to Implementation*, 3rd ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2016.

[36] ProAssets. Free HDR Sky. https://assetstore.unity.com/packages/2d/textures-materials/sky/free-hdr-sky-61217. Accessed: 2021-05-20.

[37] Schlick, C. An Inexpensive BRDF Model for Physically-based Rendering. *Computer Graphics Forum* (1994).

[38] Schwarz, M., and Müller, P. Advanced Procedural Modeling of Architecture. *ACM Trans. Graph. 34*, 4 (July 2015).

[39] Smelik, R. M., Tutenel, T., Bidarra, R., and Benes, B. A survey on procedural modelling for virtual worlds. *Computer Graphics Forum 33*, 6 (Jan. 2014), 31–50.

[40] Steiger, A. How Realtime Ray Tracing (RTX) Will Revolutionise Video Game Graphics. https://unitydevelopers.co.uk/how-ray-tracing-rtx-will-revolutionise-video-game-graphics/.

[41] Steinberger, M., Kenzel, M., Kainz, B., Wonka, P., and Schmalstieg, D. On-the-fly generation and rendering of infinite cities on the GPU. *Computer Graphics Forum 33*, 2 (May 2014), 105–114.

[42] Sunday, D. *Practical Geometry Algorithms: with C++ Code*. 2021. http://geomalgorithms.com.

[43] Supnik, B. OpenStreetMap and X-Plane 10. https://developer.x-plane.com/2011/04/openstreetmap-and-x-plane-10/, 2011.

[44] Supnik, B. OSM: What Data Will the Global Scenery Use. https://developer.x-plane.com/2011/04/osm-what-data-will-the-global-scenery-use/, 2011.

[45] Tan, R. T. *Specularity, Specular Reflectance*. Springer US, Boston, MA, 2014, pp. 750–752.

[46] Vavera, V. Real-time ray tracing in unreal engine. Master's thesis, Czech Technical University in Prague, 2020. Accessed: 2021-05-19.

[47] Wald, I. Siggraph 2019/2020 OptiX 7/7.3 Course Tutorial Code. https://github.com/ingowald/optix7course.

[48] Wonka, P., Wimmer, M., Sillion, F., and Ribarsky, W. Instant architecture. *ACM Trans. Graph. 22*, 3 (July 2003), 669–677.

[49] Yu, X. *OSM-Based Automatic Road Network Geometries Generation on Unity*. PhD thesis, 2019.

Appendix A

Renders

This chapter provides renders from the developed rendering application. It contains images from different views taken during different parts of the day.

Figure A.1: Picture with a night view taken in *Berlin*.

Figure A.2: Picture with a night view taken in *Berlin*.

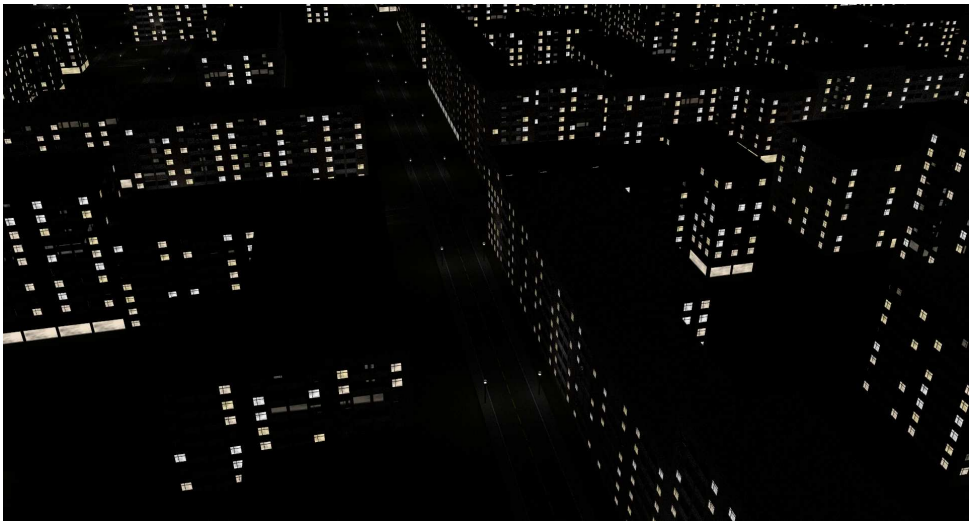


Figure A.3: Picture with a night view taken in *Belgrade*.

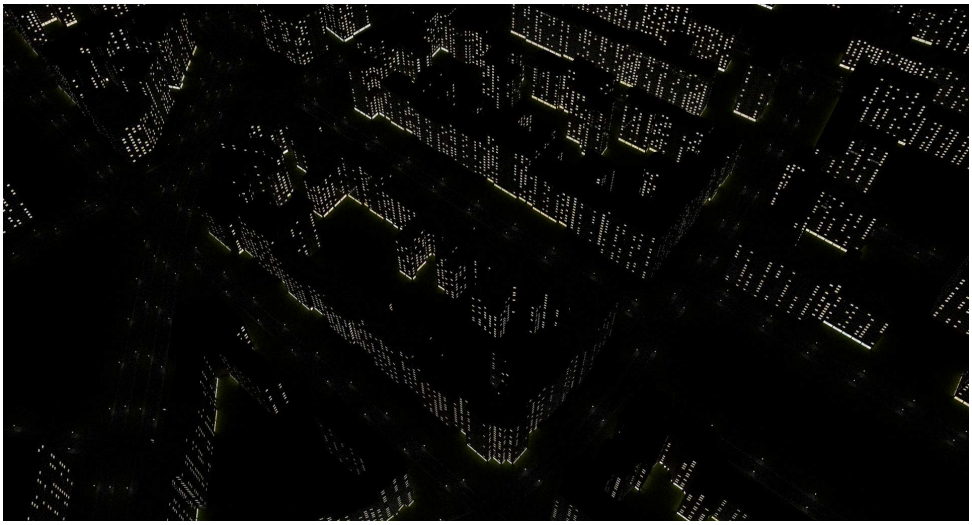


Figure A.4: Picture with a night view taken in *Berlin*.

Figure A.5: Picture with a view during sunrise taken in *Berlin*.

Figure A.6: Picture with a view during sunrise taken in *Berlin*.

Figure A.7: Picture with a daytime view taken in *Berlin*.

Figure A.8: Picture with a daytime view taken in *Berlin*.

Figure A.9: Picture with a view during sunset taken in *Berlin*.

Figure A.10: Picture with a view during sunset taken in *Berlin*.

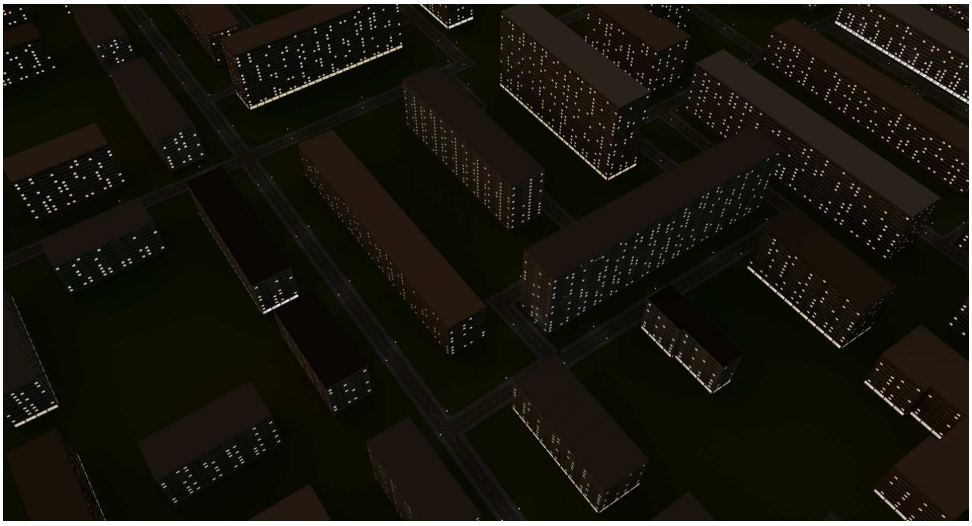


Figure A.11: Picture with a view during sunset taken in *Krasnoyarsk*.

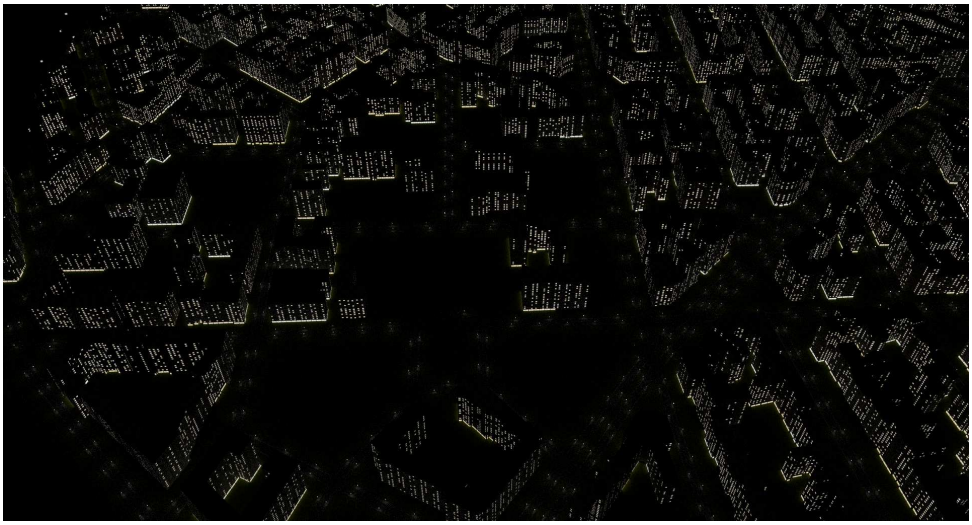


Figure A.12: Picture with a night view taken in *Berlin*.

Figure A.13: Picture with an evening view taken in *Berlin*.

Figure A.14: Picture with a night view taken in *Berlin*.

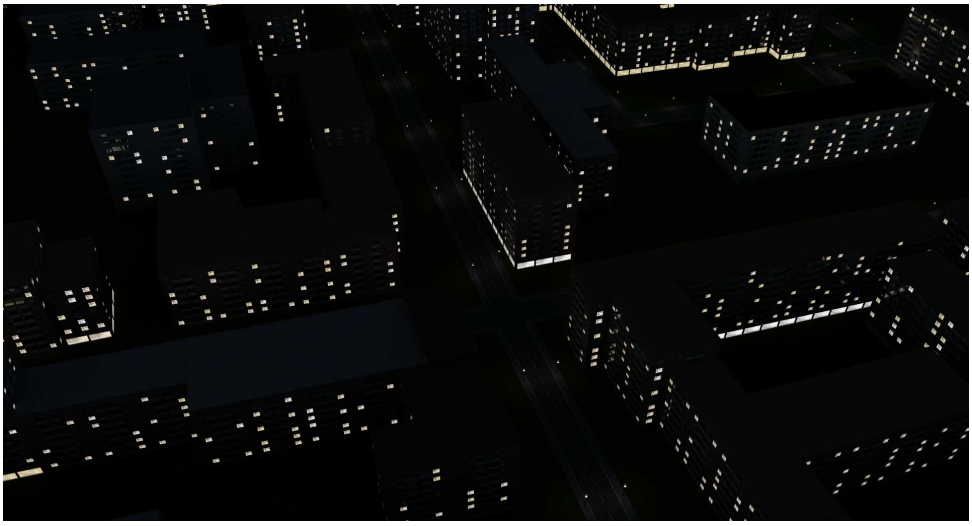


Figure A.15: Picture with an evening view taken in *Belgrade*.

Figure A.16: Picture demonstrating reflections in windows.

Figure A.17: Picture demonstrating reflections in windows.

Figure A.18: Picture demonstrating reflections in windows.

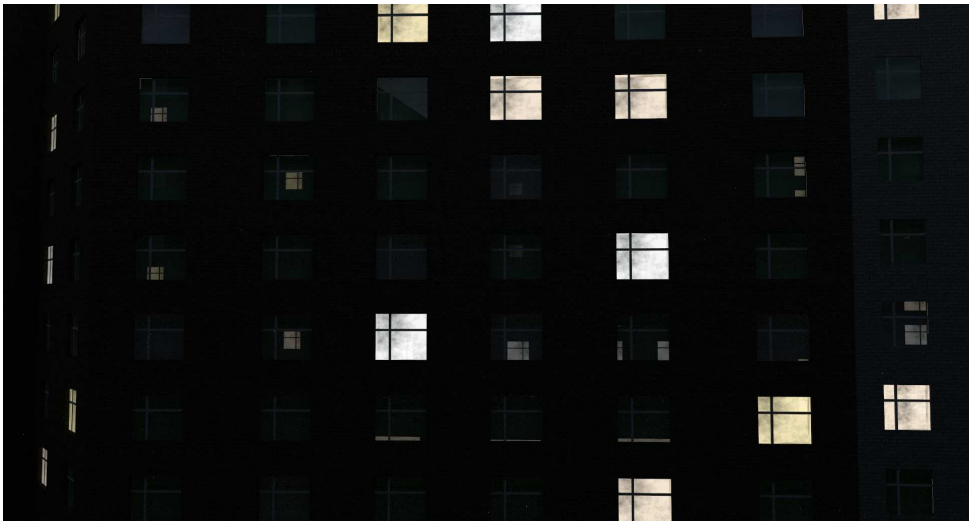


Figure A.19: Picture demonstrating windows with lights on.

Figure A.20: Picture demonstrating windows with lights on.

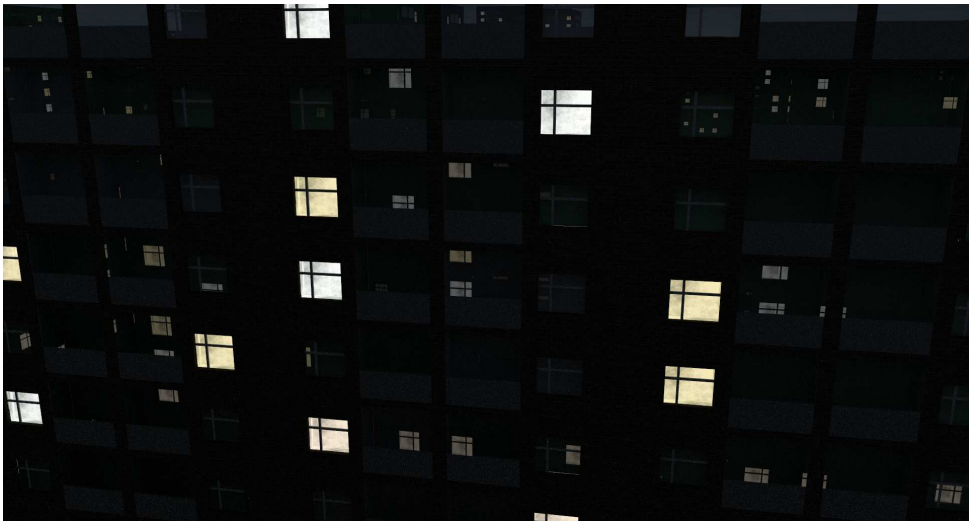


Figure A.21: Picture demonstrating windows with lights on.

Figure A.22: Render from *Belgrade*.

Figure A.23: Render from *Belgrade*.

Figure A.24: Render from *Belgrade*.

Figure A.25: Picture with a night view taken in *Krasnoyarsk*.

Figure A.26: Picture with a night view taken in *Krasnoyarsk*.

Figure A.27: Picture with a night view taken in *Krasnoyarsk*.

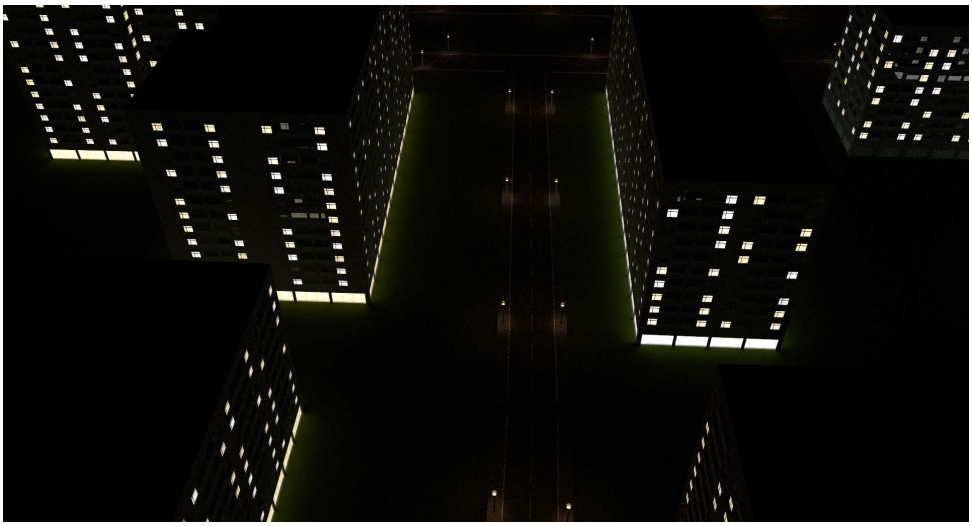


Figure A.28: Picture with a night view taken in *Krasnoyarsk*.

Figure A.29: Render from *Krasnoyarsk*.

Figure A.30: Render from *Krasnoyarsk*.

Appendix B

Rendering Application Manual

This chapter provides instructions for installing the rendering application and the project for Visual Studio. The project is available on GitLab[1]. First, you have to download and install CUDA 10.0[2] and Nvidia Optix 7.2.0.[3]

B.1 Visual Studio Project

To install the project, you must copy with replacement file `CMakeLists.txt` and folders `data`, `cuda`, `direct_rendering_of_procedural_models`, and `sutil` to `..\OptiX SDK 7.2.0\SDK\`. Build the rendering application and the sample projects provided with the API using CMake.

B.2 Executable File

The executable file accepts input arguments. They are listed in table B.1. All of them are optional, because the application has default values.

Argument	Value	Default value	Note
`--seed=`	Example: `1234`	`0`	The value to initialize the random number generator.
`--dim=`	Example: `1024x768`	`1920x1080`	The size of the application window.
`--building-min-h=`	Example: `25`	`20`	The minimum building height.
`--building-max-h=`	Example: `80`	`100`	The maximum building height.
`--city=`	Example: `"Belgrade_1"`	`"Berlin_1"`	The name of the city for the scene.
`--heightmap=`	Example: `heightmap_1`		The name of the heightmap for the scene. If you do not want to use any heightmap, do not use this argument.

[1] `https://gitlab.fel.cvut.cz/temnyale/model_rendering_with_ray_tracing`
[2] `https://developer.nvidia.com/cuda-10.0-download-archive`
[3] `https://developer.nvidia.com/designworks/optix/downloads/legacy`

--lamp-collider-size=	Example: 0.5	0.9	The size of the street lamp colliders.
--geometry-terrain=	0 or 1	0	0 for the traditional geometry representation, 1 for the procedural one.
--geometry-buildings=	0 or 1	0	0 for the traditional geometry representation, 1 for the procedural one.
--geometry-edge-roads=	0 or 1	0	0 for the traditional geometry representation, 1 for the procedural one.
--geometry-joint-roads=	0 or 1	0	0 for the traditional geometry representation, 1 for the procedural one.
--geometry-sidewalks=	0 or 1	0	0 for the traditional geometry representation, 1 for the procedural one.
--print-memory-info			If passed, the program prints the amount of the total, used and free memory.
--print-scene-info			If passed, the program prints the number of the buildings, roads, sidewalks and lamps.
--print-scene-geometry-info			If passed, the program prints an approximation of required memory for the scene.
--cu-out-buf=	CUDA_DEVICE, GL_INTEROP, ZERO_COPY or CUDA_P2P	GL_INTEROP	The CUDA output buffer type. The choice depends on the graphics card.

Table B.1: Input arguments for the rendering application.

So, it may be started from the command prompt with

```
direct_rendering_of_procedural_models.exe
    --seed=1155 --city="Berlin_1" --building-min-h=22
--building-max-h=44 --geometry-buildings=1 --print-scene-info .
```

In this case, the seed will be 1155, the scene will be Berlin_1, the height of the buildings will be in range [22m, 44m], the buildings' geometry will be procedural, the other objects' geometry will be traditional, the CUDA output buffer will be GL_INTEROP, and the program will print the information about the scene.

The GUI of the rendering application is shown in figure B.1. It shows the update, rendering, and display time. It has control elements to adjust the scene parameters. They are listed in table B.2.

(a) : Upper part. **(b) :** Middle part. **(c) :** Lower part.

Figure B.1: GUI of the rendering application.

Name of control	Description
Path tracing parameters/Real time/ Number of samples	The number of samples per pixel which is used in real time.
Path tracing parameters/Real time/ Russian Roulette depth	The Russian Roulette depth which is used in real time.
Path tracing parameters/Real time/ Ray termination probability	The ray termination probability in Russian Roulette which is used in real time.
Path tracing parameters/ Image rendering/ Number of samples	The number of samples per pixel which is used for image rendering.
Path tracing parameters/ Image rendering/ Russian Roulette depth	The Russian Roulette depth which is used for image rendering.
Path tracing parameters/ Image rendering/ Ray termination probability	The ray termination probability in Russian Roulette which is used for image rendering.
Scene parameters/ Skydome	The current skydome texture.
Scene parameters/ Skydome texture color factor	The factor by which the skydome texture color is multiplied.
Scene parameters/ Skydome diffuse color factor	The factor by which the skydome diffuse color is multiplied when the ray has an indirect influence on a pixel.
Scene parameters/ Skydome emissive color	The skydome emissive color.
Scene parameters/ Perlin noise parameters/ Scale	The scale parameter of the Perlin noise function.
Scene parameters/ Perlin noise parameters/ N	The N parameter of the Perlin noise function.
Scene parameters/ Perlin noise parameters/ Lacunarity	The lacunarity parameter of the Perlin noise function.
Scene parameters/ Perlin noise parameters/ Decay	The decay parameter of the Perlin noise function.
Scene parameters/ Windows with lights on	The percentage of the windows with lights on.

`Scene parameters/ Windows emissive energy factor`	The emissive energy factor for the windows.
`Scene parameters/ Shop windows with lights on`	The percentage of the shop windows with lights on.
`Scene parameters/ Shop windows emissive energy factor`	The emissive energy factor for the shop windows.
`Scene parameters/ Lamps on/off`	Whether the street lamps are switched on or off.
`Scene parameters/ Don't use the street lamp collider`	Whether to use the street lamp colliders or not.
`Scene parameters/ Street lamp emissive color`	The street lamp emissive color.
`Scene parameters/ Street lamp emissive energy factor`	The emissive energy factor for the street lamp by which the emissive color is multiplied.
`Scene parameters/ Street lamp secondary emissive energy factor`	The secondary emissive energy factor for the street lamp.
`Scene parameters/ Render building AABBs`	Render the AABBs of the buildings instead of their actual shape.
`Scene parameters/ Generate building geometry`	Whether to generate the building geometry or not.
`Scene parameters/ Render road and sidewalk AABBs`	Render the AABBs of the roads and sidewalks instead of their actual shape.
`Image rendering/ Number of frames`	The number of frames to render before saving of their average.
`Image rendering/ Render image`	Render the selected number of frames and save their average.
`Save current image`	Save the current image.

Table B.2: Control elements of the rendering application.

To move in the scene, the user has to use the **left** and **right mouse button** and the **mouse wheel**.

If the user's computer is not powerful enough to use this rendering application with high settings, it allows them to set the desired parameters

and start rendering. When the image is rendered, the rendering application saves it and returns to its previous state. The required steps for it are listed in figure B.2.

1. Set the path tracing parameters in section **Path tracing parameters/Image rendering**.
2. Set the number of frames above button **Render image**. The final image will be the average of the rendered frames.
3. Click **Render image**.

Figure B.2: Steps to render an image.

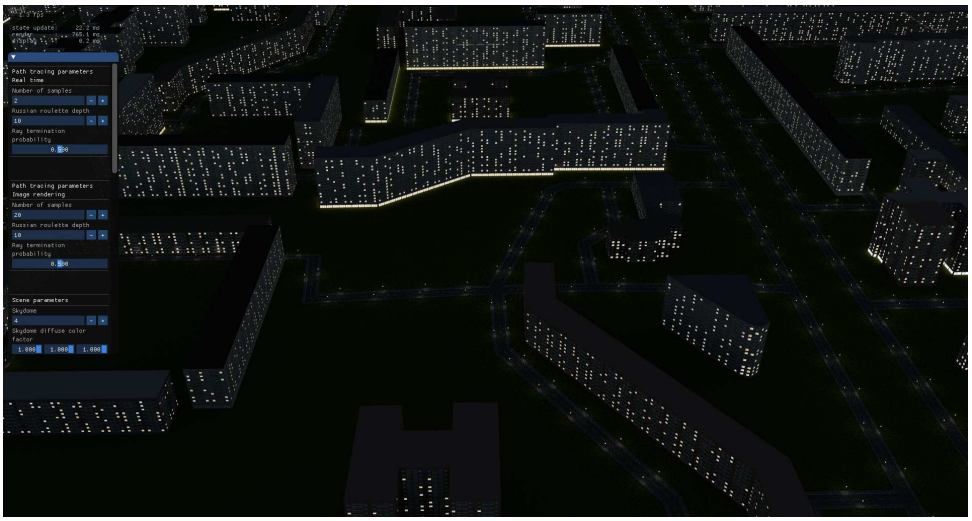


Figure B.3: Picture demonstrating the rendering application window.

Appendix C

OSM Parser Manual

The OSM processing application consists of a project for Visual Studio, which is available on GitLab[1], and a project for Unity that is used as the GUI, which is also available on GitLab[2]. It receives various arguments which are used to configure the process of data processing. It should receive the arguments through the GUI project in Unity.

C.1 Visual Studio Project

The application was developed in Visual Studio 2019.[3] All dependencies are included in the archive. You only have to open the solution in folder `OSM Parser\`.

C.2 Unity Project

The application was developed in Unity 2019.3.5f1[4]. You only have to open folder `OSM Parser GUI\` in Unity Hub. The built version of the OSM processing application should be placed in folder `OSM Parser GUI\Assets\Plugins\OSM Parser\`. Then, you may interact with OSM Parser through the GUI. The right panel allows adjusting and setting various parameters. Figure C.1 shows a screenshot of the project. The panel with the parameters is shown in figure C.2.

[1] `https://gitlab.fel.cvut.cz/temnyale/osm-parser`
[2] `https://gitlab.fel.cvut.cz/temnyale/visualiser-for-osm-parser`
[3] `https://visualstudio.microsoft.com/downloads/`
[4] `https://unity3d.com/get-unity/download/archive`

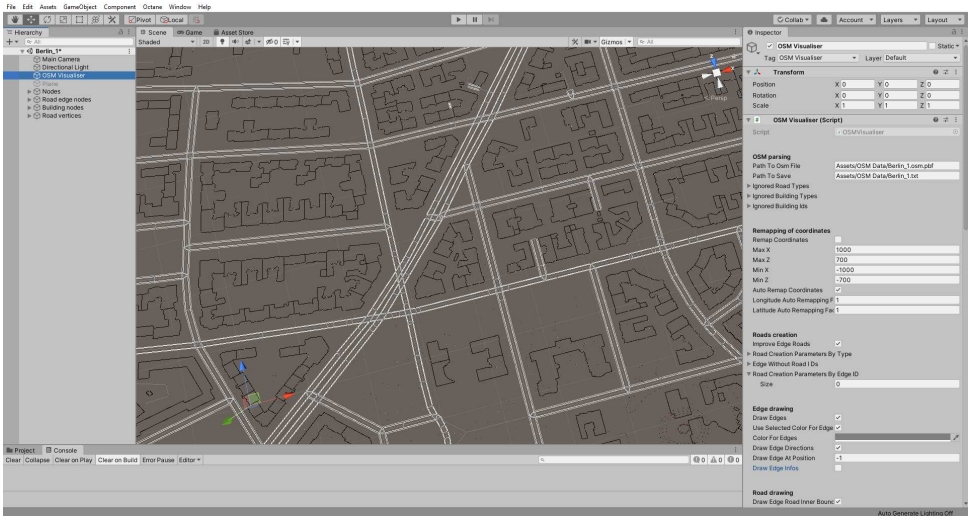


Figure C.1: Picture demonstrating the GUI of OSM Parser.

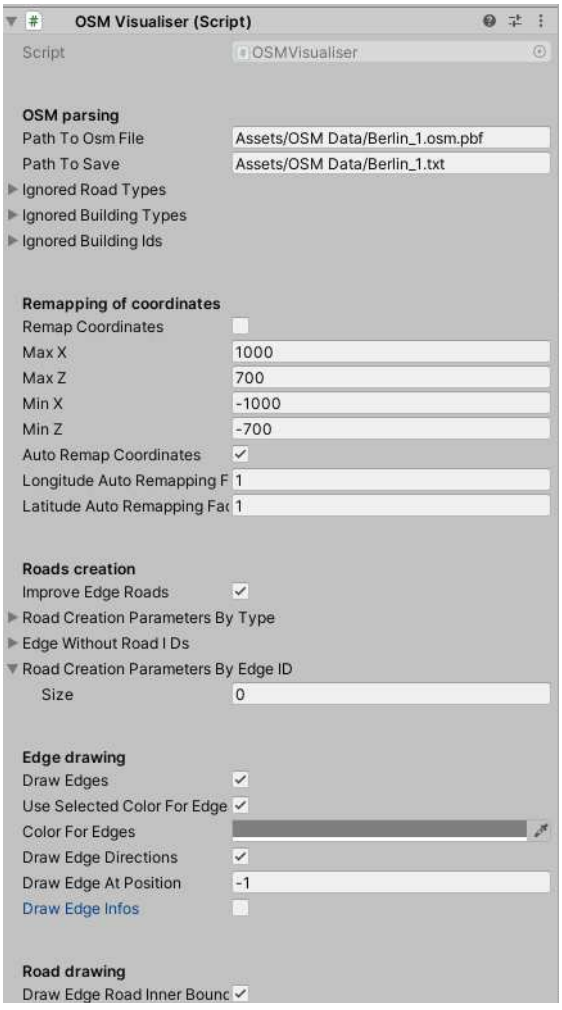
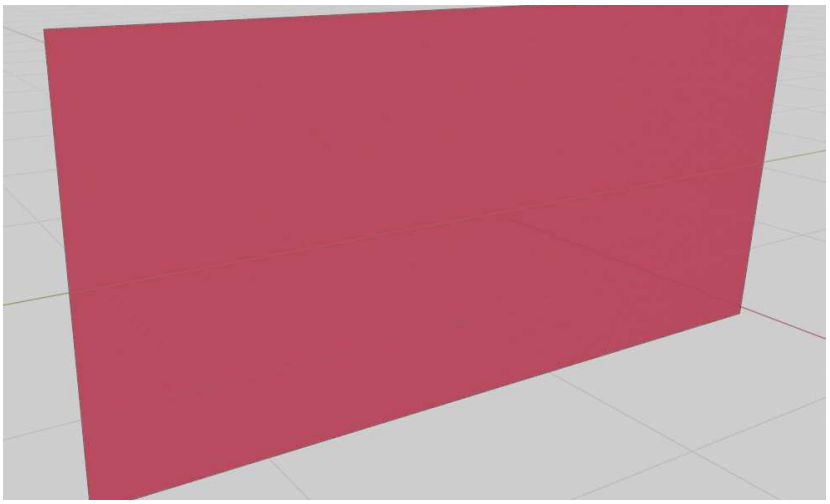


Figure C.2: Panel with the parameters of OSM Parser.

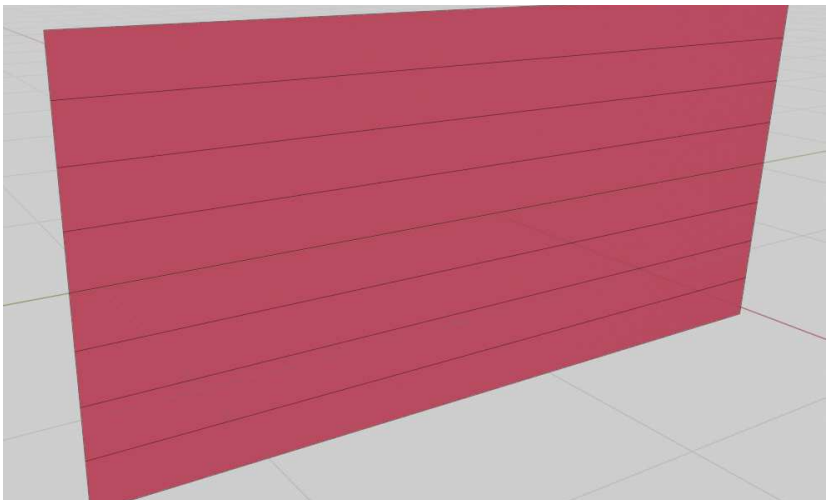
Appendix D

Facade Geometry Creation

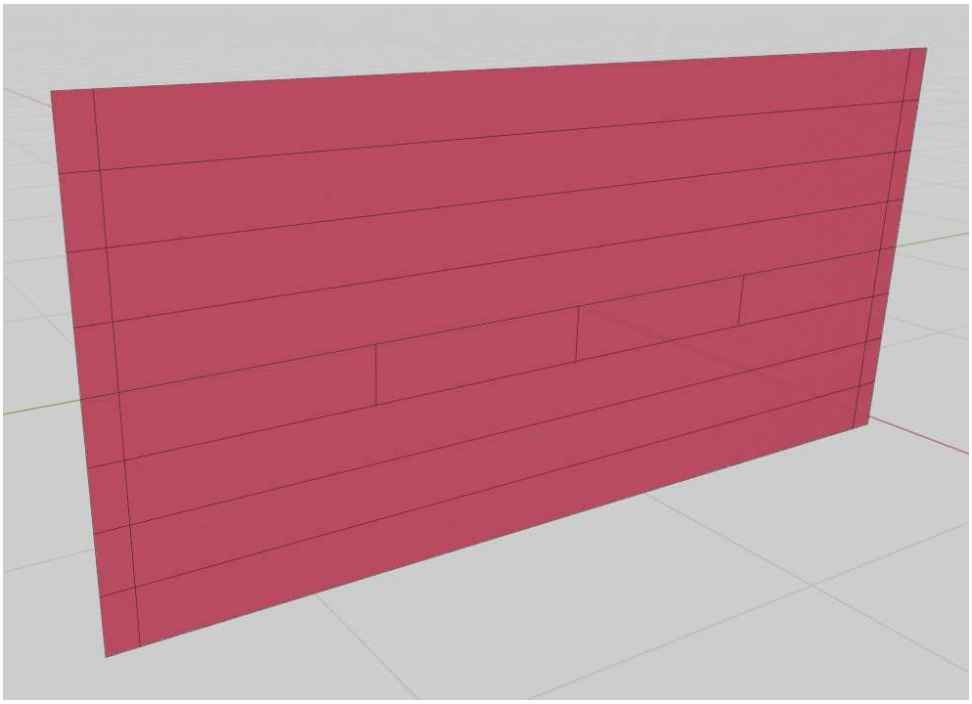
This chapter contains the images of the steps for the facade geometry creation in a larger resolution.



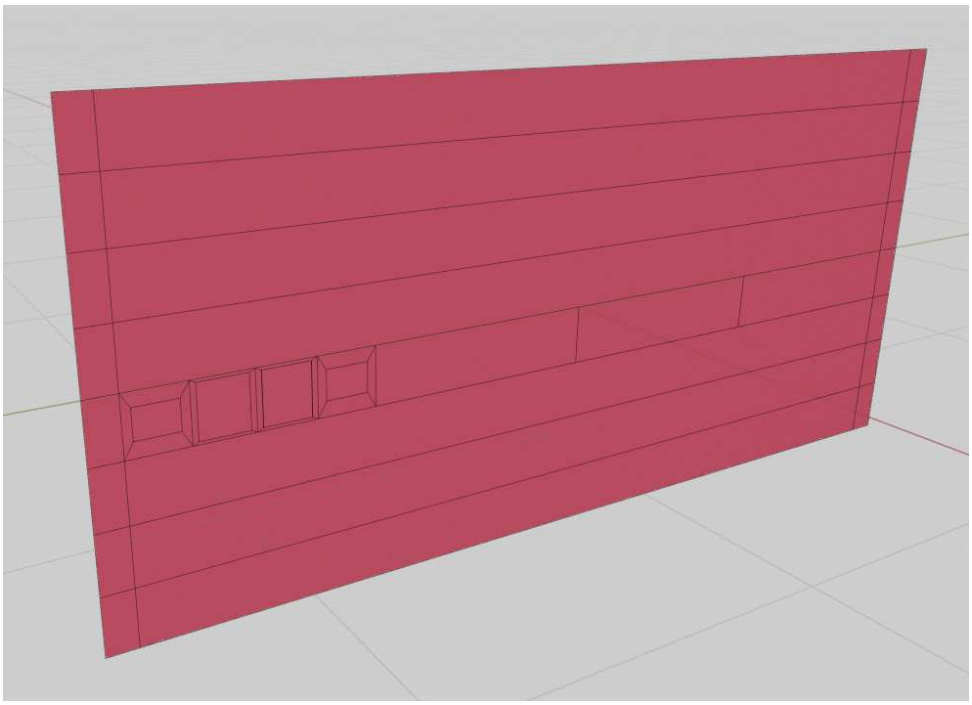
(a) : An initial wall which is used to create the geometry of the facade.



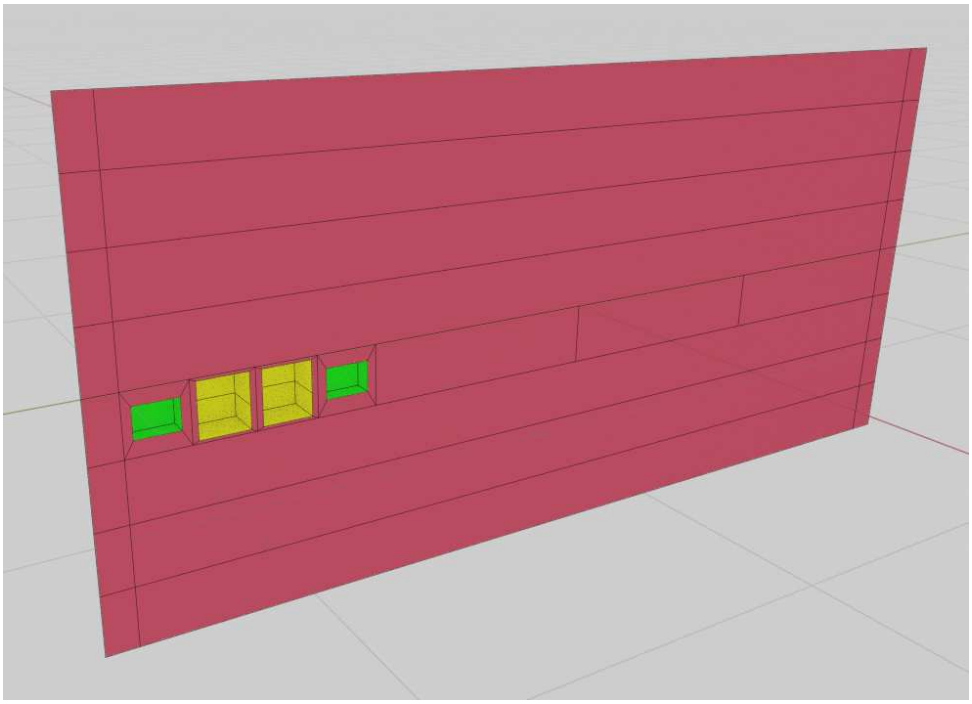
(b) : The wall is split into floors.



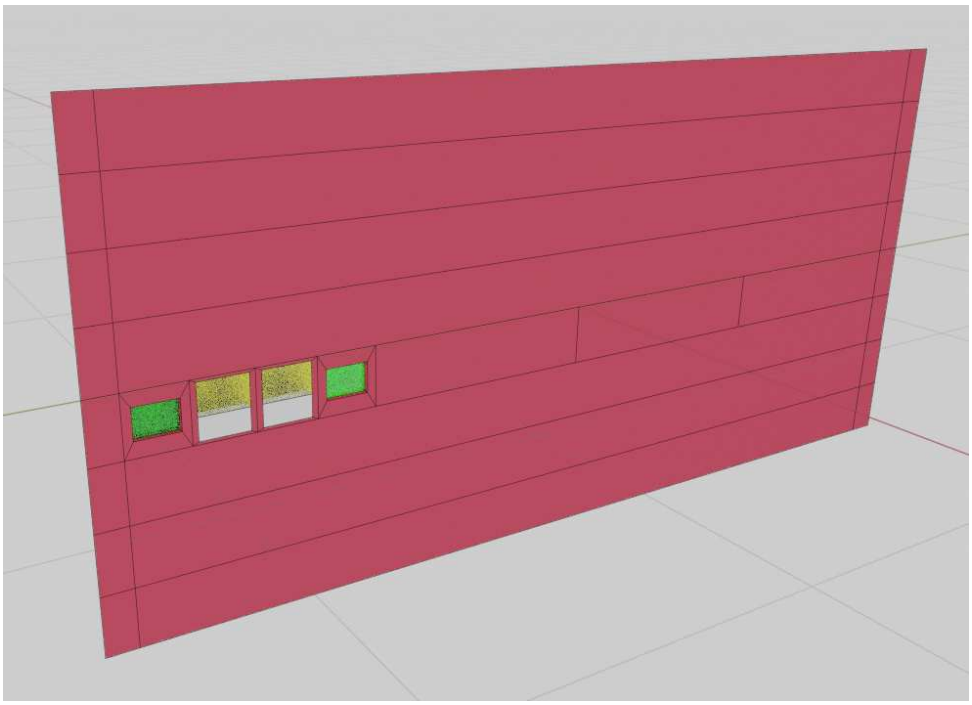
(c) : Some space at the wall's edges is cut. A floor is split into slots.



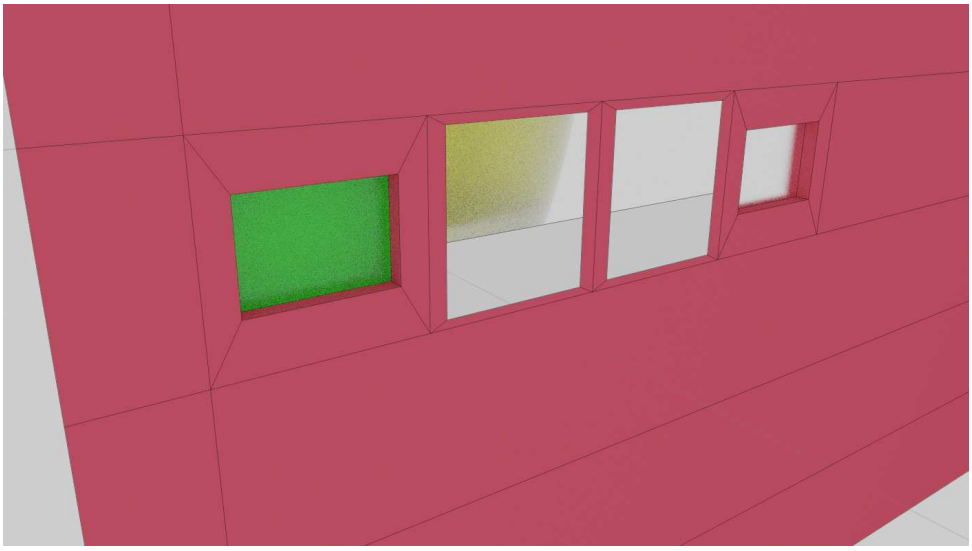
(d) : Initial rectangles for windows and balconies.



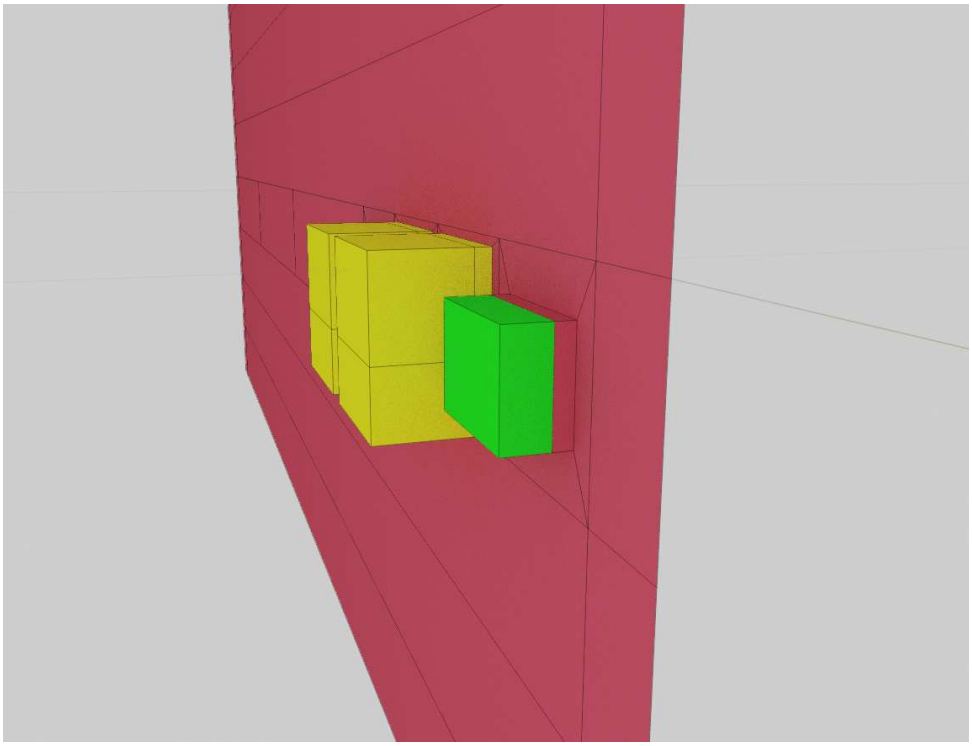
(e) : The rectangles were intruded.



(f) : The geometry for the windows and the balconies is created.



(g) : The geometry near from the front side.



(h) : The geometry near from the back side.

Figure D.1: Figure showing how to create the facade with windows and balconies.