



BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

GENERATING AND RENDERING OF LARGE VOXEL-BASED SCENES

GENEROVÁNÍ A ZOBRAZOVÁNÍ ROZSÁHLÝCH VOXELOVÝCH SCÉN

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR
AUTOR PRÁCE

Bc. DANIEL ČEJCHAN

SUPERVISOR
VEDOUCÍ PRÁCE

Ing. MICHAL MATÝŠEK

BRNO 2019

Zadání diplomové práce



Student: **Čejchan Daniel, Bc.**

Program: Informační technologie Obor: Počítačová grafika a multimédia

Název: **Generování a zobrazování rozsáhlých voxelových scén**

Generating and Rendering of Large Voxel-Based Scenes

Kategorie: Počítačová grafika

Zadání:

1. Nastudujte metody procedurálního generování a vizualizace voxelových scén.
2. Vybrané techniky popište. Navrhněte techniku pro generování, modifikaci a efektivní zobrazování komplexních voxelových scén s využitím GPU.
3. Vytvořte demonstrační aplikaci využívající zvolené techniky a ověřte její funkčnost na několika scénách.
4. Zhodnoťte dosažené výsledky, případně navrhněte možnosti pokračování.
5. Vytvořte video prezentující dosažené výsledky.

Literatura:

- Podle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Body 1 a 2, rozpracování bodu 3.

Podrobné závazné pokyny pro vypracování práce viz <http://www.fit.vutbr.cz/info/szz/>

Vedoucí práce: **Matýšek Michal, Ing.**

Vedoucí ústavu: Černocký Jan, doc. Dr. Ing.

Datum zadání: 1. listopadu 2018

Datum odevzdání: 22. května 2019

Datum schválení: 1. listopadu 2018

Abstract

This thesis focuses on creating an application for procedural generation and visualisation of a volumetric terrain using the OpenGL library. The terrain is considered to be mostly static, however with a possibility of modification of individual voxels. The project seeks a compromise between rendering performance and the aesthetics. The design is led in a way so that it could be further used as a foundation for a game. An emphasis is put on accelerating used methods on the GPU.

Abstrakt

Tato práce se věnuje vytvoření aplikace, která procedurálně generuje a následně vizualizuje volumetrický terén s využitím knihovny OpenGL. Uvažují se převážně statické scény s možností modifikace na úrovni jednotlivých voxelů. Projekt hledá kompromis mezi efektivitou a vzhledem; rozhodnutí jsou vedena se záměrem, aby výsledná aplikace šla využít jako základ pro vytvoření hry. Důraz je kladen na využití akcelerace nabízenou grafickým procesorem.

Keywords

voxels, volumetric terrain, procedural generation, OpenGL

Klíčová slova

voxely, volumetrický terén, procedurální generování, OpenGL

Reference

ČEJCHAN, Daniel. *Generating and Rendering of Large Voxel-Based Scenes*. Brno, 2019. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Michal Matýšek

Generating and Rendering of Large Voxel-Based Scenes

Declaration

Hereby I declare that this master's thesis was prepared as an original author's work under the supervision of Ing. Michal Matýšek. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Daniel Čejchan
September 7, 2019

Acknowledgements

Thank you to Ing. Matýšek, who took me under his wings and did not hesitate to dedicate his time to me. Thank you to Ing. Milet, who, although he will most definitely oppose me, was always willing to share his knowledge. Thank you to Ing. Chlubna for his ideas. Thank you to Jakub Šolín, whom I was using instead of a rubber duck.

Contents

1	Introduction	2
2	Volumetric terrain	4
2.1	Volumetric terrain procedural generation	5
2.2	Volumetric terrain representation	10
2.3	Volumetric terrain visualisation	12
3	Graphic effects	14
3.1	Lightinh	14
3.2	Additional graphic techniques	16
4	Design	19
4.1	World representation	19
4.2	Storing terrain on a disc	21
4.3	Lighting model	21
4.4	Terrain generation	24
5	Implementation	31
5.1	World representation	31
5.2	World generation and storage	32
5.3	World rendering	35
5.4	<i>Postprocessing</i>	45
5.5	<i>Skybox</i> a střídání dne a noci	50
6	Aplikace	54
6.1	Typy bloků	59
6.2	Výkon	60
7	Závěr	73
	Bibliography	74
A	Obsah přiloženého CD	76
A.1	Významné zdrojové soubory	77

Chapter 1

Introduction

In year 2009, Mojang developed the game Minecraft¹. During several years span the game has acquired a large gamer base across many age categories and became a phenomena of the game industry. It is characterized by a procedurally generated volumetric terrain, where players can control their game avatars without any specific goals. The game is suitable for multiple player types: creative ones can focus on building various models from voxels, technically focused ones can spend time inventing mechanisms (it is even possible to design your own CPU in the game), players than lone for adventure can dive into the *survival mode*, where they are required to acquire resources, build fortresses and fight for survival.

There were many titles inspired by the system introduced in Minecraft; it is even safe to say that Minecraft gave birth to a new game genre with procedurally generated worlds, survival and crafting mechanics. Minecraft itself was inspired from the game Infiniminer.



Figure 1.1: The Minecraft game

This thesis aims to examine methods usable for visualisation of a procedurally generated terrain in Minecraft-type games, then implement selected methods in a demo application and attempt to accelerate the methods on GPU. Design and implementation are lead in a way so it is possible to create a game from the application.

¹www.minecraft.net

In chapters 2 and 3 are dedicated to an overview of existing methods for procedural generation, representation and visualisation of voxel terrains and other techniques relevant to this paper. The chapters 4 and 5 document design and implementation of the application and chapter 6 evaluates the implementation.

Chapter 2

Volumetric terrain

Term “volumetric terrain representation” simply states that the data that describe terrain are always related with some finite-volume bodies in the terrain. The shape of the bodies can be generally different and even varying in a single representation. This paper considers the bodies to be uniform-sized cubes arranged to a regular grid. The term *voxel* that will be further used generally represents the smallest volume unit in 3-dimensional discrete space [14]. Data linked with the voxels can, in the simplest case, merely information, if the voxel represents a solid volume or an empty space. They can however also store more detailed information about the voxel, such as color, material, physical properties etc.

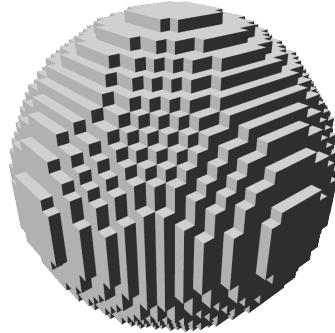


Figure 2.1: Volumetric representation of a sphere

The work with terrain can be conceptually divided into three parts: generation, management and rendering. Management stands for storing the terrain in memory and on a disc and also its manipulation in case of a non-static terrain. For this chapter, let us represent terrain with function

$$ter(x, y, z) : \mathcal{Z} \times \mathcal{Z} \times \mathcal{Z} \rightarrow \{\text{true}, \text{false}\}, \quad (2.1)$$

where x , y and z are coordinates in three-dimensional cartesian space and the function output value denotes presence of matter on the given coordinates. The axes x and y are situated into the horizontal plane, z axis represents elevation.

2.1 Volumetric terrain procedural generation

For the purposes of this thesis, procedural generation means finding such function ter (see Equation 2.1) that can be effectively implemented on present-day hardware. Furthermore, a certain level of realism is required. Traits such as mountains, plains, riverbeds, caves and such are expected. The function should also not generate a single terrain, but there should be a way to generate large amounts of unique terrains. This can be formally defined as

$$proc(\vec{pos}, seed) : \mathcal{Z}^3 \times \mathcal{N}_0 \rightarrow \{\text{true}, \text{false}\}, \quad (2.2)$$

where differing values of the $seed$ parameter differentiate between various terrains.

The methods mentioned below work with continuous space and have a continuous output:

$$proc_{\mathcal{R}}(\vec{pos}, seed) : \mathcal{R}^3 \times \mathcal{N}_0 \rightarrow \mathcal{R}. \quad (2.3)$$

For discrete output, a threshold parameter $t \in \mathcal{R}$ is defined, and also a scale parameter $\vec{s} \in \mathcal{R}^3$ relating the size of a voxel to a real world metrics:

$$proc(\vec{pos}, seed) = proc_{\mathcal{R}}(\vec{pos} \otimes \vec{s}, seed) > t, \quad (2.4)$$

where $\vec{a} \otimes \vec{b}$ is a component-wise scalar multiplication. For generating three-dimensional terrains, it is also possible to utilize functions working in 2D space. The value of a function in the point x, y then represents terrain slope height on the given coordinates:

$$proc(\vec{pos}, seed) = proc_{2\mathcal{R}}(\vec{pos}_{xy} \otimes \vec{s}_{xy}, seed) < \vec{pos}_z \cdot \vec{s}_z. \quad (2.5)$$

2.1.1 Perlin noise

Perlin noise is a widely utilized noise generation method created by Ken Perlin [19, 20]. The nature of the noise is applicable for a wide variety of applications, such as natural material textures, fire, smoke, clouds and so on. The method can be used for any number of dimensions. For outputs that change continuously in time, it is possible to add time as another dimension.

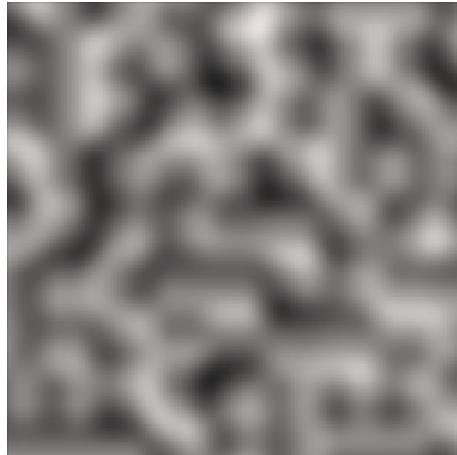


Figure 2.2: 2D Perlin noise

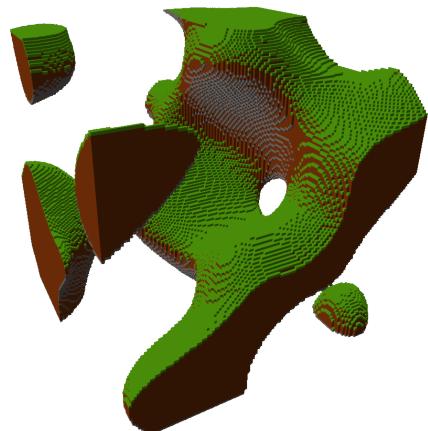


Figure 2.3: Volumetric terrain generated by 3D Perlin noise

The noise is calculated as following: a unitary orthogonal grid is established in the space. Value \overrightarrow{grad} is pseudorandomly determined for each node of the grid (D represents the noise dimensionality):

$$\overrightarrow{grad}(\overrightarrow{pos}, seed) : \mathcal{Z}^D \times \mathcal{N} \rightarrow \mathcal{Z}^D. \quad (2.6)$$

For 2D noise, unit-length vectors are recommended. For 3D, Perlin defines 12 specific vectors (vectors to centers of edges of a unitary cube) that are to be selected from pseudorandomly. Then, around a point \overrightarrow{pos} , where value of the noise is to be computed, 2^D nearest nodes $\overrightarrow{node}_{i \in \langle 1, 2^D \rangle}$ are selected, forming a D -dimensional unitary cube. For each of the nodes, dot value is calculated as

$$dot_i(\overrightarrow{pos}, seed) = (\overrightarrow{pos} - \overrightarrow{node}_i) \cdot \overrightarrow{grad}(\overrightarrow{node}_i, seed). \quad (2.7)$$

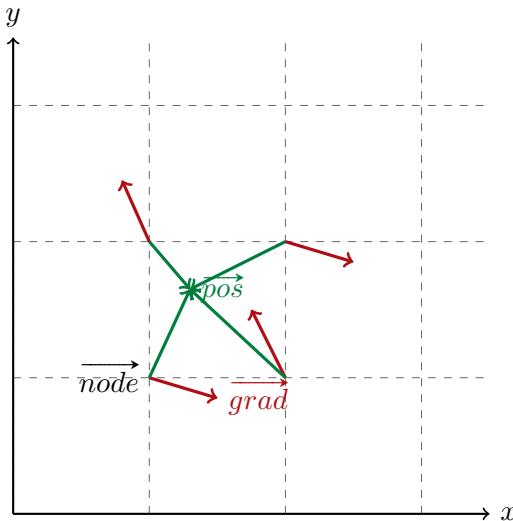


Figure 2.4: Visualisation of gradients and grid in 2D Perlin noise

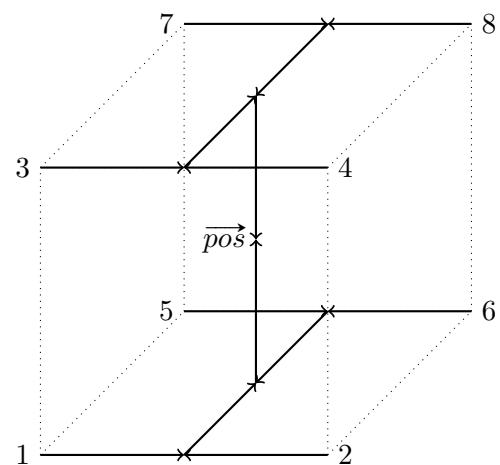


Figure 2.5: Visualisation of an interpolation between grid node gradients in 3D Perlin noise

The resulting noise value is then computed by interpolation between the dot_i values:

$$perlin(\overrightarrow{pos}, seed) = \sum_{i=1}^{2^D} dot_i(\overrightarrow{pos}) \cdot \prod_{j=1}^D ipol(1 - |\overrightarrow{pos}[j] - \overrightarrow{node}_i[j]|). \quad (2.8)$$

Because the used grid is unitary, $|\overrightarrow{pos}[j] - \overrightarrow{node}_i[j]|$ will always be in interval $\langle 0, 1 \rangle$. For the interpolation function $ipol$, Perlin [20] defines

$$ipol(x) = 6x^5 - 15x^4 + 10x^3 \quad (2.9)$$

with zero first- and second-order derivation in $x = 0$ and $x = 1$. The interpolation computation can be alternatively understood as gradual interpolation between pairs of values,

see Image 2.5. For better visual quality, the noise is usually combined in multiple scales and amplitudes:

$$moPerlin(\vec{pos}, seed) = \sum_{o=1}^O perlinNoise(\vec{pos} \cdot s^o, seed) \cdot p^{(o-1)}, \quad (2.10)$$

where O number of combined noises, s is the scale reduction a p is the amplitude reduction. Individual layers of such a noise are called octaves.

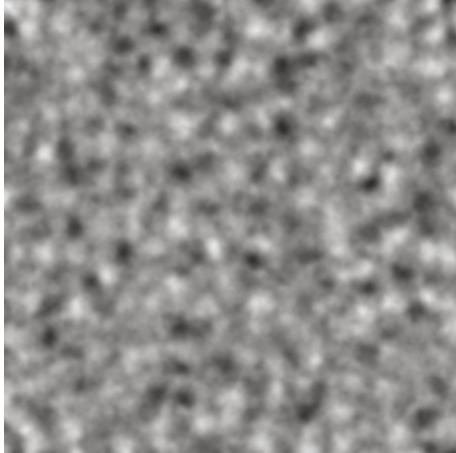


Figure 2.6: Result of a combination of multiple 2D Perlin noises with varying amplitudes and frequencies

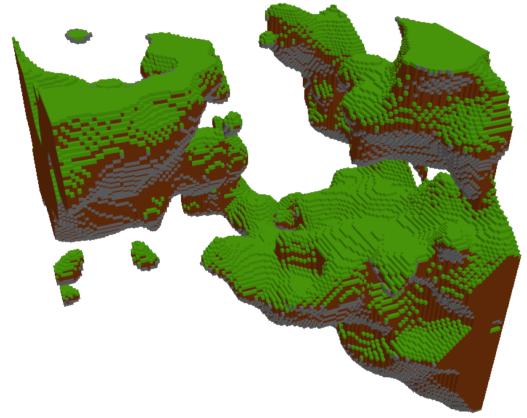


Figure 2.7: Volumetric terrain generated with multi-octave 3D Perlin noise

2.1.2 Simplex noise

Simplex noise was created by Ken Perlin [21] as a *better* alternative to Perlin noise. The improvements stated by Perlin are for example isotropy (the noise is the same in all directions), better performance and scaling to higher dimensions. While in Perlin noise the number of vector operations grows exponentially, the growth is linear in Simplex noise. Differences of the algorithms principles are summed up in an article by Stefan Gustavson [13].

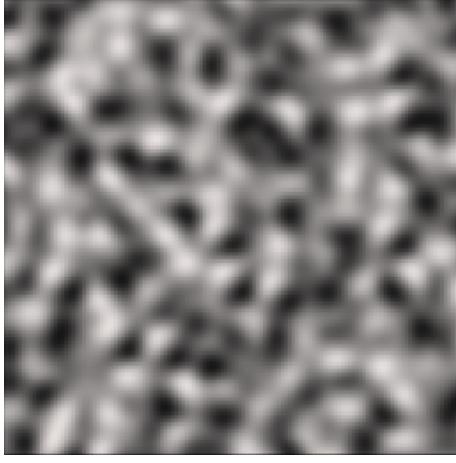


Figure 2.8: 2D Simplex noise

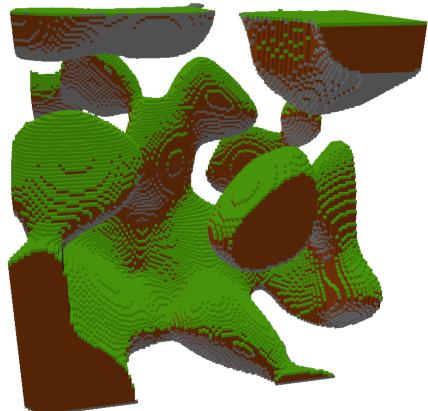


Figure 2.9: Volumetric terrain generated by 3D Simplex noise

The primary difference between the two noises is the grid, which in Simplex noise does not divide the space into D -dimensional cubes, but to so called *simplexes* that have only $D + 1$ vertices (compared to 2^D vertices of cubes).

For determining the simplex vertices, a coordinate transformation is used:

$$skew(\vec{v}) = \vec{v} + \frac{\sqrt{D+1}-1}{D} \cdot \sum_{d=1}^D \vec{v}[d], \quad (2.11)$$

where D is the dimensionality of the noise. The reverse transformation is calculated as

$$unskew(\vec{v}) = \vec{v} - \frac{1}{D} \left(1 - \frac{1}{\sqrt{D+1}} \right) \cdot \sum_{d=1}^D \vec{v}[d]. \quad (2.12)$$

Position of the first simplex vertex (the nearest one to the coordinate system origin) is determined as

$$\overrightarrow{node}_1 = unskew(\lfloor skew(\overrightarrow{pos}) \rfloor). \quad (2.13)$$

The remaining D vertices are calculated in a sequential manner where we add $unskew(\vec{k})$ to the position of the previous vertex $\overrightarrow{node}_{i-1}$, where \vec{k} is zero in all dimensions except for one ($\vec{k}[s_i] = 1$). The order of the sequence of s_i is determined by sorting the offset vector $offset = \overrightarrow{vec} - \overrightarrow{node}_1$ in ascending order. Then $\overrightarrow{node}_i = \overrightarrow{node}_{i-1} + \overrightarrow{vec}[s_i]$. Similarly to Perlin noise, gradients are pseudorandomly determined for the vertices. The resulting value of the noise is then calculated as

$$simplex(\overrightarrow{pos}, seed) = \sum_{i=1}^{D+1} max(0, 0.6 - |\vec{o}_i|^2) \cdot \left(\vec{o}_i \cdot \overrightarrow{grad}(\overrightarrow{node}_i, seed) \right), \quad (2.14)$$

where $\vec{o}_i = \overrightarrow{pos} - \overrightarrow{node}_i$.

2.1.3 Voronoi diagrams

Let $P \subset \mathcal{R}^D$ be a set of points randomly distributed in a D -dimensional space. Then the space is divided into regions based on P : each point in the space $s \in \mathcal{R}^D$ belongs to region of the closest point $p \in P$. This structure is called a Voronoi diagram [5].

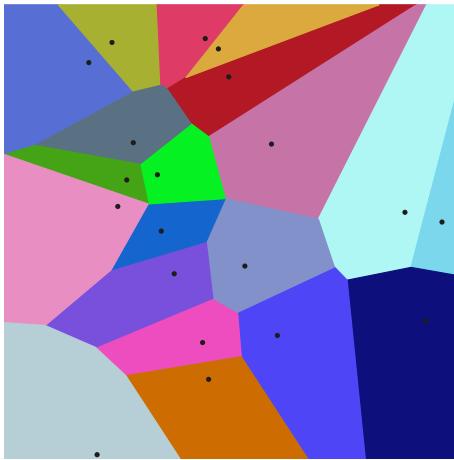


Figure 2.10: Euclidean 2D Voronoi diagram

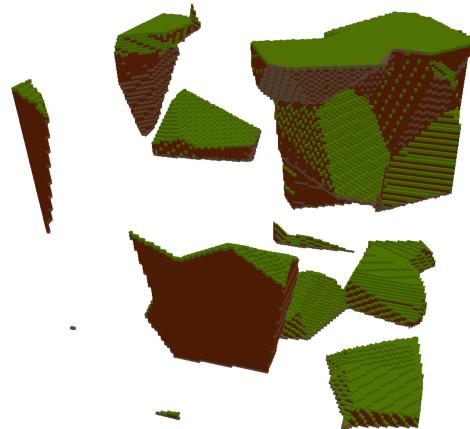


Figure 2.11: Volumetric terrain generated by a 3D Voronoi diagram

Because of its cellular structure, Voronoi diagrams can be a good foundation for generating organic material textures, rocky materials or mosaics. They can be used for procedurally generating borders of parcels or various regions. The nature of the diagram itself can be an inspiration of an art style – there are many possible usages. Choosing a non-euclidean metric can expand the utilization even more, as demonstrated in Image 2.12 generated with the metric

$$\rho(\vec{a}, \vec{b}) = \sqrt[4]{|\vec{a}_x - \vec{b}_x|^4 + |\vec{a}_y - \vec{b}_y|^4 + |\vec{a}_z - \vec{b}_z|^4}. \quad (2.15)$$

It is also possible to assign a numerical value to each point in space – for example nearest to the n-th nearest point $p \in P$ – this approach is used in a noise function designed by Steven Worley [27].

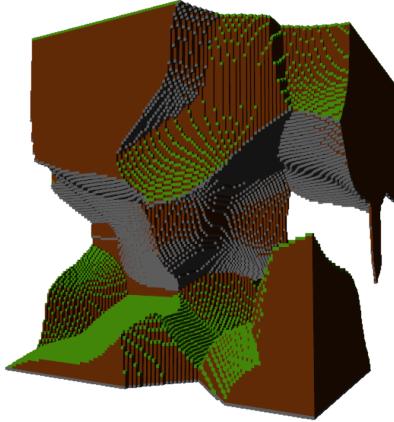


Figure 2.12: Volumetric terrain generated by a Voronoi diagram with non-euclidean metric

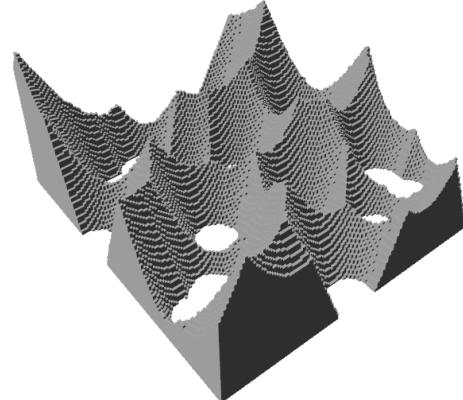


Figure 2.13: Volumetric terrain generated from a height map based on a 2D Voronoi diagram; the height is calculated as distance to the nearest point $p \in P$

To fit Voronoi diagram to the formalisms defined in this chapter, value $v_p \in \langle 0, 1 \rangle$ is assigned to each $p \in P$. Then

$$\text{voronoi}(\vec{pos}, \text{seed}) = v_p, \text{ kde } \vec{p} \in P \text{ a } \rho(\vec{pos}, \vec{p}) \text{ is minimal for all } \vec{p} \in P. \quad (2.16)$$

2.1.4 L-systems

The topic Lindenmayer systems (L-systems) is too complex for including it into this thesis. It is methodically elaborated for example by Prusinkiewicz [18]. In short, L-systems are defined as triplets $H = (V, P, \omega)$, where V is a set of symbols, P is a set of rules in form $a \rightarrow x$, where $a \in V$, $x \in V^*$ and $\omega \in V^+$ is the starting string. Based on the rules in P , the original string ω is iteratively rewritten, resulting in a new string α . The string α is then graphically interpreted by an agent who executes the symbols in the string as instructions.

There are multiple types of L-systems; for example L-systems with a stochastic rule application, with branching or with parametric symbols. For the purposes of this thesis, L-system can find applications in procedural generation of trees, grass or other vegetation.



Figure 2.14: Bush generated by a D0L-system, copied from [24]

2.2 Volumetric terrain representation

If the terrain can be represented by a function $\text{ter}(x, y, z) \rightarrow \{\text{true}, \text{false}\}$ as defined in 2.1, it is theoretically possible to use this function in visualisation and there is no need to store any data of the terrain. The function will however be too computationally demanding for it to be run 60 times per second for each voxel in the scene. A more viable approach is to calculate the function for each voxel just once and store the results in an appropriate data structure. It is also a part of the assignment to be able to edit the terrain, so it is required to store at least the changes.

As a reference scale for memory requirements, let's consider a chunk of $1024 \times 1024 \times 256$ voxels (270 million voxels). Each voxel would require four bytes of information, which corresponds to approximately 1 GB of memory with no overhead and no compression. Considering a potentially infinite procedurally generated terrain, it is necessary to accommodate for gradual growth of the terrain as further chunks are generated with the movement of the camera. Because of the amount of the data, a system for storing parts of the terrain to a disc will be required.

2.2.1 Array

A possibility is to store voxel data in a three-dimensional array. This approach grants both constant-time reads and writes. However, having the entire terrain stored in a single array

would require expensive reallocations with the growth of the terrain. Also, there is no way to swap parts of the terrain to a disc.

It would be appropriate then to combine the array with another data structure, such as *hash table*. The terrain would be divided into fixed-size chunks; voxels in each chunk would be stored in a static array. Regions would be organized in the hash table, with key being position of the chunk. This hybrid approach still keeps constant-time random reads and writes, where iteration over voxels in a single chunk would also be very fast. Implementation of swapping chunks to disc is also possible and very easy.

This approach introduces no compression, so large homogenous areas are not optimized in any way. A different method might be more suitable for cases where for example 50 % of the terrain is empty.

2.2.2 Sparse voxel octree

Sparse voxel octree (SVO) is a tree data structure, where each node is either a leaf or has exactly eight children. Every node represents a volume region, typically a cube; children of a node then divide the region into eight smaller parts, typically cubes with half the edge length compared to the parent. Sections of a terrain with the same value of $ter(x, y, z)$ can be aggregated into a single node.

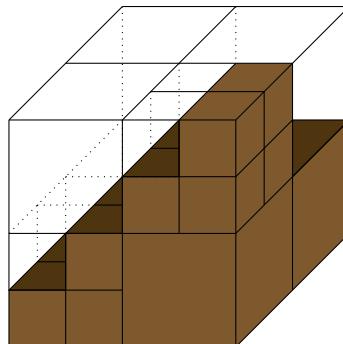


Figure 2.15: Visualisation of a *octree* volumetric terrain representation

In random access, the tree structure is traversed from the root node; random reads have a time complexity of $O(\log n)$, where $n = w \cdot h \cdot l$ is the volume of the scene. Reads will require $O(\log n)$ nonsequential memory accesses that can be slowed down by *cache misses*. Writes have the same complexity and disadvantages and they can result in dynamic memory allocation/deallocation during node splitting/joining.

SVO can dramatically reduce both memory and computational requirements in cases where the terrain contains large homogenous regions; in an ideal case such regions can be represented by a single tree node and rendered as a single cube. On the other hand, in the worst-case scenario, where neighbouring voxels always differ, both the memory and computational complexity can be several orders higher than with the static array.

Swapping parts of the tree to disc can be easily implemented. Similarly to the static arrays, it might be suitable to combine SVOs with a *hash table* to reduce speed reductions with the terrain growth.

2.2.3 3D texture on GPU

When storing the terrain on a GPU, it is possible to utilize 3D textures that provide memory-effective data structure and access functions for it. The structure implementation is not exactly defined and can vary with different graphic cards and configurations. Giesen [11] describes methods for storing textures on the GPU that are based on a linear memory mode, but with [swizzling] introduced that permutes bits of a texel address. By choosing a good permutation function it is possible to achieve decrease of neighbouring (in 2D/3D) texels distance in the linear memory – for example by creating a so called Z-order. Swizzling does not introduce any overhead, GPU textures should use the same amount of space as linear arrays on CPU (textures are more prone to the 2^N size alignment).

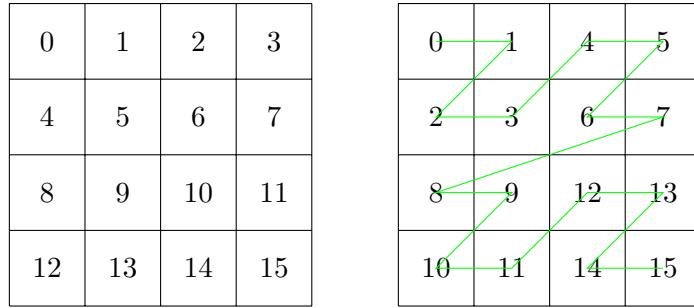


Figure 2.16: Demonstration of *Z-order swizzling* when storing 2D texture data on GPU

2.3 Volumetric terrain visualisation

Elvins in his overview [1] categorizes methods of volumetric terrain visualisation to *direct volume rendering* (DVR) and *surface-fitting* (SF) methods.

2.3.1 Direct volume rendering methods

Direct volume rendering methods work directly with the voxel data and do not create any custom representation for it. A typical method in this category is *ray casting*, where for each pixel of the screen a ray is casted, the color of the pixel is then determined by first intersection of the ray with the terrain.

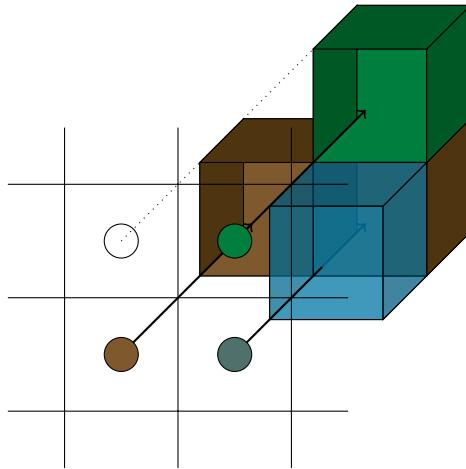


Figure 2.17: Demonstration of *ray casting*

The ray does not have to be stopped by a first non-empty voxel: a opacity can be implemented. The ray is then cast further and the resulting pixel color is determined by blending. Ray casting is not a lighting method as it does not consider any lighting and shading. There are however lighting methods based on a similar principle – for example *ray tracing* or *voxel cone tracing* (with more information in Section 3.1) – that achieve high photorealism.

2.3.2 Surface-fitting methods

Surface-fitting methods translate the volumetric data to a surface representation (triangles) which is then rendered using the standard rendering pipeline. The most naive translation is simply rendering faces of non-empty voxels. This method always produces a clearly noticeable perpendicular structure. The structure can be reduced for example using the *marching cubes* method, as demonstrated in Image 2.18. There are other methods, working with curves and normal vectors, that attempt to surface-fit even partially transparent voxel data; those are mentioned for example in Modern Computer Graphics [14].

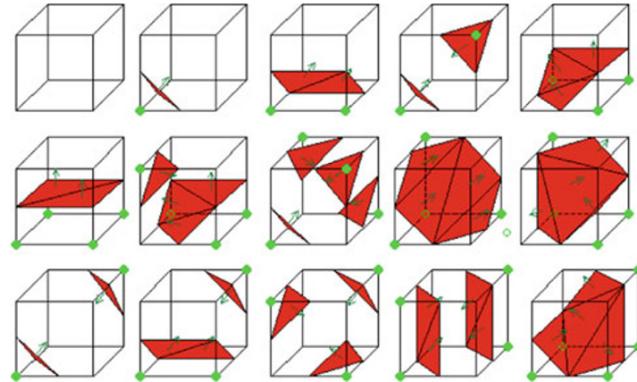


Figure 2.18: Possible surface representations in the *marching cubes* methods, copied from [6]

Chapter 3

Graphic effects

In the real world, the image we see is mediated by photons hitting retina in our eyes. Although the human eye is able to detect a hit of a single photon [25], tens of trillions of photons hit our eye every second during daytime.¹ These photons are during their way to human's eyes reflected, bent, absorbed and re-emitted. It is a goal of computer graphics to reproduce these phenomena. In this chapter, a selection of phenomena and techniques concerning this topic is summarized.

3.1 Lightinh

The visualisation methods alone, without any lighting model, assume uniform ambient light. The first step to photorealism is introducing situated light sources. Basic two types of light-object real world interaction are diffuse and specular reflections. In specular reflection, the light ray is reflected in the same plane – this reflection is associated with glossy and reflective surfaces. In diffuse reflection the ray is scattered randomly in all directions – this reflection is associated with matte surfaces and is most contributing to the perceived object color.

Žára in his book [14] distinguishes local and global illumination methods. Local illumination methods are generally faster, because during the calculations they only consider a light source, an object and the observer – the ray reflects maximally once during its way from the light source to the observer. Global illumination methods consider the scene as a whole, the ray can reflect multiple times and from different objects.

3.1.1 Global illumination methods

A typical representative of global illumination methods is *ray tracing*. The method casts rays for each pixel of a screen and for each ray hit it calculates illumination in the hit location; the ray reflection count is limited for performance reasons. There are multiple methods for casting the rays, they can differ in the calculations performed on ray hits or in the direction the rays are traced (from observer to light sources or the other way around). Because it is not possible to analytically compute the scene lighting from the light behaviour equations on current-day hardware, a Monte Carlo sampling is performed. The results quality correlates with the sample count – when the sample count is unsufficient, the image appears grainy. Even when using Monte Carlo methods, computational abilities of an average consumer hardware are not enough to perform real-time ray tracing calculations,

¹Very approximate estimation, based on <https://physics.stackexchange.com/questions/329971>

and so these methods are usually used for video or image rendering. This status is however slowly changing with development of graphic cards optimized for these applications, such as the new nVidia RTX™[3].

There are more global illumination methods – for example *voxel cone tracing* [7] does not represent the rays as lines, but as cones. As a complement to ray tracing, which is good for glossy surfaces, the *radiosity* method is used, which performs well for indirect diffuse lighting.

3.1.2 Local illumination methods

As mentioned above, local illumination methods consider only a light source, an observer and a single object during each computation. There is no indirect illumination nor reflections. No obstacles, not even the object itself, are considered between the object and the light source. Models describing this simplified situations are used as a basis for global illumination methods, too.

In 1997, Bui-Tuong Phong [22] proposed an empirical (not based on real-world physics) model for local illumination calculations, that eventually became a standard for real-time rendering applications. The model considers three components for the shading:

- **Ambient component** is not affected by position of light or object to the observer. It describes “omnipreset light” which is constant for the entire scene. Its purpose is to prevent non-illuminated parts of objects to be completely black (in real world there usually isn’t absolute darkness, too).
- **Diffuse component** has usually the largest impact on the perceived object color. Its intensity depends on the angle between the illuminated surface and the light source, but not on the position of the observer.
- The intensity of the **specular component** depends both on the angle between the light source and the illuminated surface and on the angle between the surface and the observer. Its intensity peaks when the angle between the light source and the surface is the same as the angle between the surface and the observer.

Individual components can have different influence for different materials, for example the specular component is reduced in matte materials. The resulting illumination of an object is then sum of these three components for all light sources in the scene. The calculation is performed for each rendered scene pixel, or alternatively for each screen pixel in *deferred shading*.

3.1.3 Shadow mapping

Because local illumination methods themselves do not contain mechanics for determining if a given point is in shadow or not, the problem must be solved separately. It generally applies that if there is an obstacle between an observed point and a light (the obstacle can be the object itself), the point is in shadow and the light does not influence it. Semi-transparent objects are not considered as they make the issue way more complex. One of methods for determining if a point is in shadow or not is *shadow mapping* created by Lance Williams [26]. The method renders the scene from each light source perspective, storing the depth data in a Z-buffer. When testing if a point is in shadow or not a line between the point and a given light is considered. If the length of the line is higher than the value in

the Z-buffer (there is something in front of the point from the light perspective), the point is in shadow.

3.2 Additional graphic techniques

Although more complex existing rendering/shading models cover a wide variety of optical phenomena, they are usually not suitable for real-time applications due to their high computational costs. Because of that, more simple models are opted for that are less photorealistic; approximations of various phenomena are then implemented as additional effects.

3.2.1 Depth of field

Depth of field (DoF) is a phenomenon caused by optics in a human eye (or a camera) that is able to focus only on a single depth plane; an object is more blurry as the distance between the object and the focus plane increases. Various implementations of the DoF effect are listed for example in the GPU Gems book [10]. One of the techniques is *Reverse-Mapped Z-Buffer Depth of Field*, which works in screen space and applies blur on each pixel with the amount depending on its distance from the focus plane (the distance is calculated from the depth buffer).

3.2.2 Screen Space Ambient Occlusion

Ambient Occlusion is a global illumination phenomenon where narrow, concave areas receive less ambient light (because there are surfaces that absorb the light) and are darker. This effect is observable typically in room corners, tree hollows etc. One of techniques that approximate this phenomenon in screen space is *Screen Space Ambient Occlusion* (SSAO). The method takes several samples from the depth buffer from each pixel p and calculates how many of these samples are inside a semi-sphere defined by the point p and its normal. Every sample located in the semi-sphere contributes to darkening the pixel p (see Image 3.1). This technique was first introduced in CryEngine 2 [17].

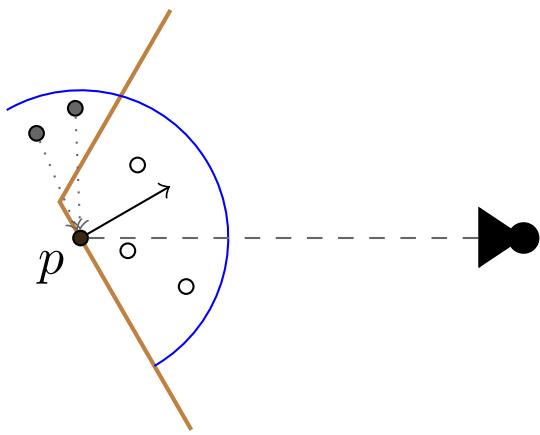


Figure 3.1: Principle of *Screen Space Ambient Occlusion*

3.2.3 God rays

Crepuscular rays, also called sun rays or god rays, is a phenomenon caused by light being scattered in a not completely transparent air. The light can be scattered and bent by dust or semi-opaque gas particles, aerosol etc. Because of this, the air itself seems to be glowing. If the light source is partially covered, the air behind the obstacle is in shadow. A pattern of occluded and non-occluded areas then creates an impression of individual light rays being visible.

The graphic effect representing this phenomenon is called *volumetric light scattering*. There are multiple possible ways of implementation. Kenny Mitchell [2] describes a method that samples depth buffer between a screen pixel p and a light source in screen space; the scattering effect then depends on how many of the samples are behind the light source and are not occluding it. Benjamin Glatzel in his presentation on the Digital Dragons 2014 conference [12] describes a method which for each screen pixel samples points on a ray cast to the scene and tests if the given sample is in the shadow or not.



Figure 3.2: Photo with visible crepuscular rays, author Les Chatfield

3.2.4 Order independent transparency

Rendering of semi-transparent primitives requires solving of additional problems. The operation of semi-transparent pixel blending is not commutative, so it is necessary to know order of the pixels from the camera view. That traditionally required sorting the primitives by their distance from camera and sequential rendering from back to front (painter's algorithm). The sorting can however easily become a bottleneck when rendering large scenes. Therefore, techniques that do not require the ordering were researched – so called *order independent transparency* methods. These techniques handle ordering on the pixel level, not on the primitives level. There are multiple approaches, for example stochastic transparency [8], where fragments are rendered with a probability corresponding to their transparency. This method introduces a noise that is suppressed by multiple rendering and averaging. Yang [28] presents a method that operates with a linear list of transparent fragments for each pixel of the screen that requires only a single pass of rendering. McGuire [16] also introduces an approach that requires a single pass, where the non-commutative blending operation is replaced by a commutative approximation.

An another approach is so called *depth peeling* [9], which introduces multiple rendering passes for transparent objects; in every pass an additional depth test is performed, discarding fragments with equal or lower depth compared to the previous pass. With each rendering pass, a new semi-transparent layer is created; the resulting image is then blended from these layers. Further publications introduce variants of this technique, for example dual depth peeling [4].

Chapter 4

Design

In this chapter, individual aspects of volumetric terrain generation, representation and visualisation design created in this thesis are described. The subject of this chapter is only a general conception, implementation details are then documented in Chapter 5.

4.1 World representation

For the memory world representation, a following hierarchical structure was chosen:

- The world is split into *chunks*, which are areas of $16 \times 16 \times 256$ voxels each (the voxels can also be referred to as *blocks* in this document). Chunks are organized by a hash table, with the key being chunk position in the world.
- In chunks, individual voxels are stored in a linear array.

Hash table enables having only a small area round a player loaded in the memory (while the world itself is potentially infinite), individual chunks can be loaded and unloaded according to the player's movement. Memory access to any loaded block is very fast, with constant complexity. The world is considered to be heterogeneous with high voxel data diversity, so tree structures would probably be more memory demanding and multiple indirections would severely reduce the memory throughput. Storing blocks in a linear arrays also allows to direct-copy the data to and from GPU, which is highly utilized in this project. Storing and loading chunks to/from a disc is also very simple.

For each voxel, two bytes determining the voxel type (*block ID*) are stored; for the purposes of this thesis it would be enough to have only one byte per block, however for fully-developed game scenario 255 block types would not be enough. Additionally, although this mechanism is not fully demonstrated in this thesis, another two bytes per block are dedicated for storing additional block data (for example plant growth level, door open/close state, ...). If the block would need to store more data (for example when the block is a chest with an inventory and it needs to store information about its contents), the two-byte data field would store index for a dynamically allocated array (each chunk has its own array) where a pointer to the data structure would be stored (*large data*).

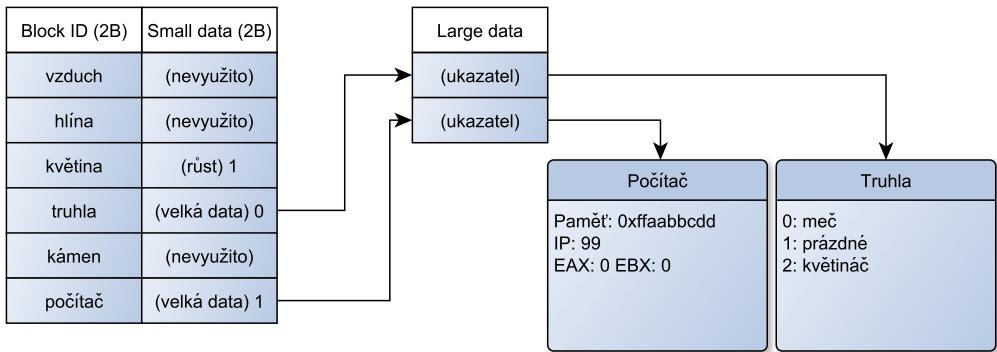


Figure 4.1: Voxel data structure

The height of the world is limited to a single chunk: because of the daylight calculations, all the blocks on the vertical axis (z) would have to be loaded anyway (the calculations are described in Section 4.3); limiting the vertical view distance would also not make much sense and could be confusing for the player. Satisfactory results can be achieved with a relatively small height maximum (compared to the horizontal view distance) and the resources required for enabling larger chunk heights can better be utilized for increasing the horizontal view distance.

The selection of the chunk height is influenced by many factors:

- A chunk should not be too small, so its overhead can be better divided between the aggregated voxels.
- The dimensions should be power of two because of various optimizations.
- Chunk dimensions in the horizontal plane (width and length) should be the same for simplicity.
- Height of a chunk determines height of the world: there should be enough space for underground, ocean and mountains.
- For the implementation simplicity, all mechanics should have a maximum influence radius of one chunk (8-neighbourhood). For example the light should be able to propagate only to neighbouring chunks, but not further. So the chunk should be large enough so the maximum light radius is not overly limited.
- Volume of a chunk should correspond with a range of values that can be stored in voxel *small data*. That way it is ensured that each voxel in the chunk can have *large data*.

Based on the criteria mentioned above, the size of $16 \times 16 \times 256$ voxels was selected for a chunk. The 256 voxel height reasonably covers demands of the terrain generation; some mountains can have an elevation of over 100 blocks, so a smaller chunk height (128) would be a considerable limitation (because it is also necessary to dedicate some height layers to underground, ocean water mass and the base ground terrain). The horizontal size 16×16 is small enough for a reasonable terrain granulation, with 16 light spread radius being not too restrictive. Horizontal voxel position in a chunk can be represented by 4 bits, which can be aligned well into bytes. And the volume of a chunk is 2^{16} , which exactly corresponds to the two bytes used for storing *small data* (or *large data* indexes).

4.2 Storing terrain on a disc

As a format for storing the terrain to a disc, SQLite was selected. Chunks then correspond to rows in the `chunks` table, the rows are indexed by the chunk coordinates. *Block IDs* are then stored in binary format in a dedicated column, the stored data being zlib-compressed copy of the memory linear array.



Figure 4.2: Data structure for storing terrain in SQLite database

SQLite handles all the organisation of the disc space fragmentation for individual chunks, making the whole solution very simple and universal, everything is stored in a single file. Because of the zlib compression, the saved world is relatively small.

4.3 Lighting model

The core idea of the lighting model introduced in this thesis is same as in the Minecraft game. The model has a constant complexity with the light count increasing. It can be understood as the *Light Propagation Volumes* method with first order spherical harmonic function [15]. It can also be described by a cellular automaton.

A regular cubic grid is introduced in the world, with the same density as the voxel grid, where nodes of the grid are situated in block midpoints. A light value is assigned to each grid node; the light value simply represents “the amount of light” in each voxel, it does not contain any directional information. The light can propagate through the grid, each step (8-neighbourhood or more precisely 26-neighbourhood in 3D) reducing the light value by 1 (plus additional inhibition determined by the block type). Value of multiple light sources is not combined additively, but using the *max* function.

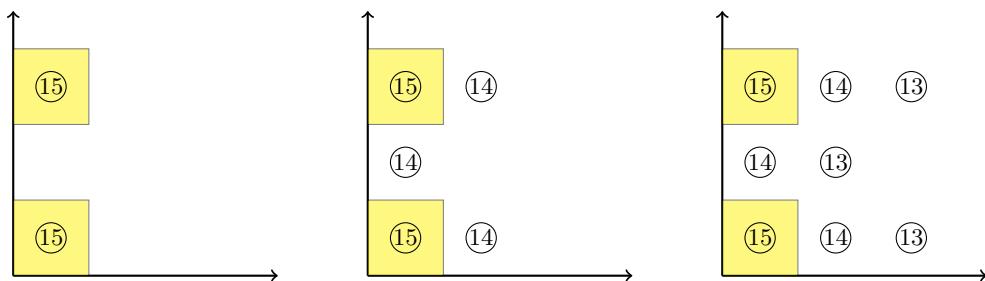


Figure 4.3: Light propagation demonstration according to the model used in this thesis (in 2D, individual diagrams correspond with the calculation steps).

The light is composed of four components: red, green, blue and daylight. Value of each component is stored in 4 bits, so each node requires 2 bytes in total. Color of the

daylight component changes based on daytime; the daylight components can also propagate vertically without intensity reduction.

The lighting calculation proceeds as follows:

1. Set values of all nodes to zero; if a voxel corresponding to a node is a light source, set the node value to the intensity of the light source.
2. Iterate from top to bottom and in every vertical column sequentially set value of the daylight component. The value to be set is maximum on the top of the column and decreases based on opacity of blocks in the column.
3. Iterate until the values stop changing: update value of each one according to this formula: $v_x \leftarrow \max(v_x, n_{0,x} - d, \dots, n_{5,x} - d)$, where $x \in \{R, G, B, D\}$ are individual light components, $n_{i,x}$ are light intensity values of surrounding nodes (in 6-neighbourhood) and $d = 1 + d_v$ is the light reduction (d_v is reduction determined by a block on the position, 0 = no reduction).

For any arbitrary position in the world (not aligned to the grid), it is possible to calculate the lighting value using a linear interpolation between the eight closest grid nodes. When rendering block faces, the sampled point in the lighting grid is offset by half unit in direction of the face normal; without this offset, overly dark values would be sampled, because block faces are exactly in half distance between an outer node (which is bright) and the node in the center of the block (which is completely dark) – so without the offset, the interpolation would return a value affected by the complete darkness in the center of the block.

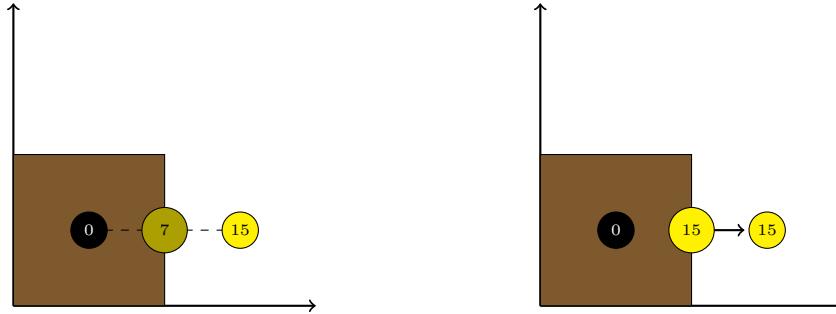


Figure 4.4: Demonstration of a sampling point offset by half unit in the face normal direction for lighting calculations (in 2D): without the offset (left) and with the offset (right)

As a consequence of this model, the lighting natively exhibits the *ambient occlusion* effect, where concave corners are darkened:

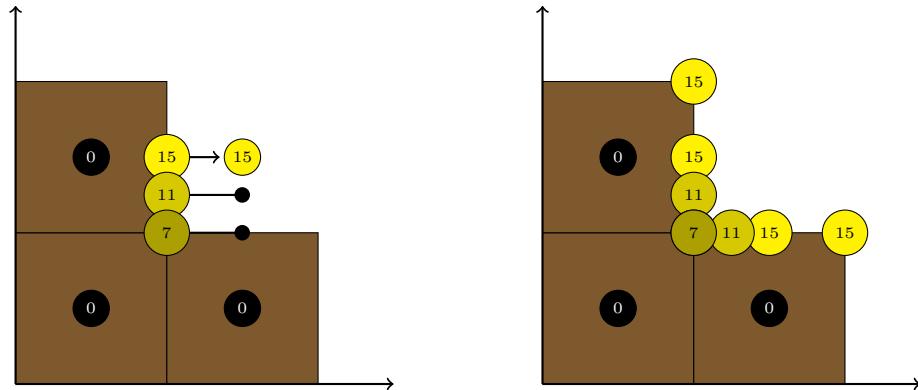


Figure 4.5: Demonstration of a native *ambient occlusion* in the lighting model (in 2D)

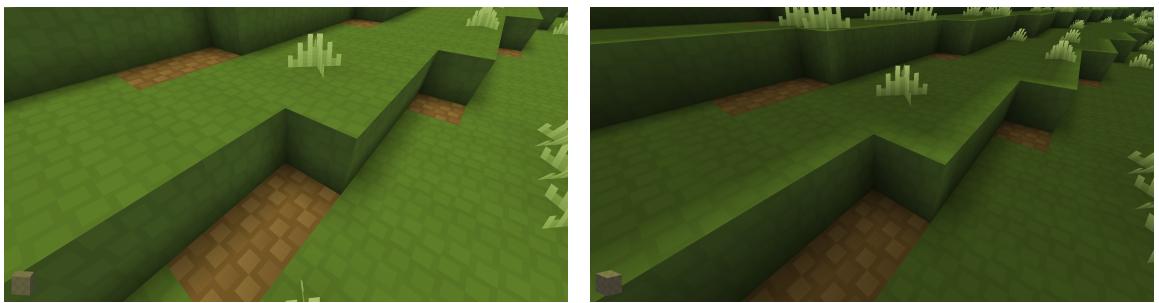


Figure 4.6: The lighting model with the offset-by-normal (left) and without (right)

The lighting model also produces artefacts where the light “spills” through borders that separate areas in 4-neighbourhood (6-neighbourhood in 3D), but not in 8-neighbourhood (26-neighbourhood in 3D):

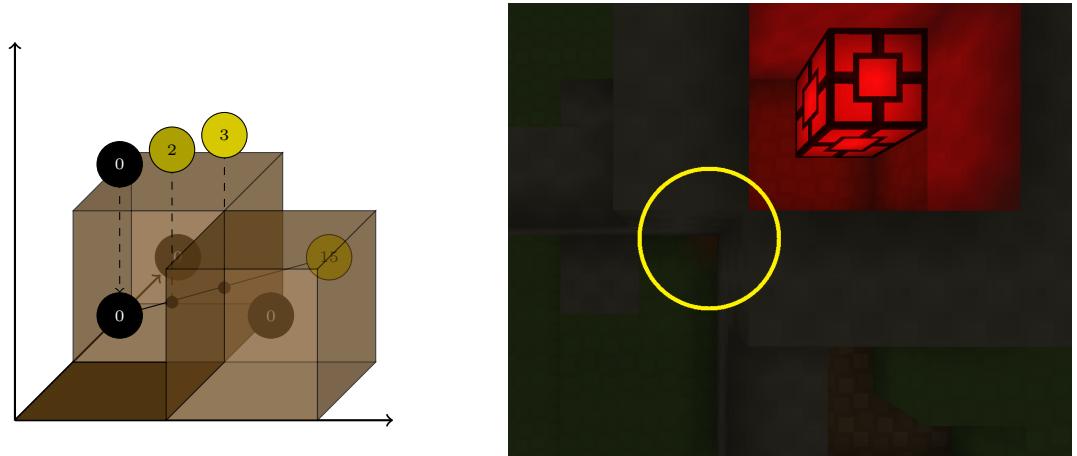


Figure 4.7: Artefact caused by the lighting model: the light “spills” through “sharp” edges

The daylight component is further separated into two subcomponents: ambient and direction. The final pixel color when the lighting is applied is then calculated as

$$\vec{c} = \vec{c}_{albedo} * \left((\vec{l}_{rgb} + g) \cdot (1 - u \cdot l_{day}) + l_{day} \cdot \left(\vec{d}_{amb} + \max(0, \vec{n}_{face} \cdot \vec{n}_{sun}) \cdot \vec{d}_{dir} \right) \right), \quad (4.1)$$

where \vec{c}_{albedo} is the color pixel before lighting is applied, \vec{l}_{rgb} is vector of the RGB light components for the given point, g is glow of the given point (can be passed in alpha channel of a texture), $u \in \langle 0, 1 \rangle$ is reduction of artificial (non-daylight) light sources, l_{day} is daylight component for the given point, \vec{d}_{amb} is color of ambient daylight component, \vec{n}_{face} is a normal for the given point, \vec{n}_{sun} is a normal of the sunlight and \vec{d}_{dir} is color of the directional component of daylight. During day, the artificial light source reduction can be up to $u = 0.8$; this mechanism is supposed to reflect that the daylight is much brighter than artificial light sources, so the artificial sources appear weaker in daylight. Color and direction of the daylight can change without requiring any additional light propagation calculations/updates.

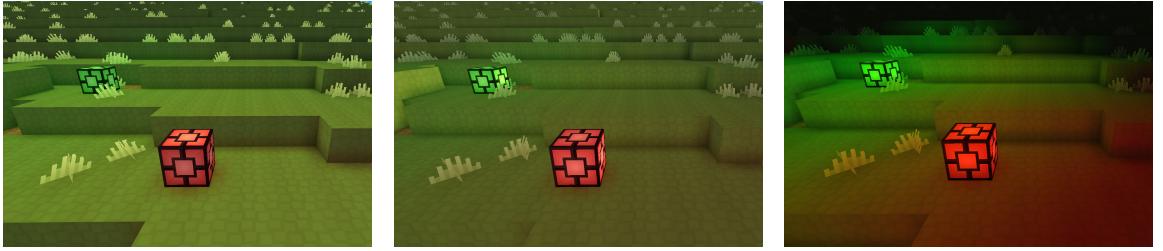


Figure 4.8: Demonstration of the directional component of daylight and reduction of artificial light sources

Alongside with the lighting system, standard *shadow mapping* is also applied; *shadow mapping* is however calculated only for the sunlight (so artificial light sources cast no shadows).

Design of the lighting model is very similar to the lighting model used in the Minecercraft game: they both utilize cellular light propagation, the propagation formula is also the same and both models have a separate component for daylight. More detailed information about the model used in Minecraft is not known. Contrary to Minecraft, the lighting model proposed in this thesis introduces RGB lighting, directional component of daylight and the normal-offset sampling system which produces native *ambient occlusion*.

4.4 Terrain generation

Procedural terrain generation in this thesis utilizes 2D and 3D Perlin noise and Voronoi diagrams; principles of those are detailed in Section 2.1. Some calculations are realized in 2D, so they are the same for each vertical voxel column (the x, y coordinates are the same, the z coordinate is used for altitude). First, values representing the terrain properties are calculated for each point in 2D (a high octave size 2D Perlin noise is used); these values represent properties such as *tree rate*, *desertness*, *mountainess* and so on. They will be represented by variables in format $e_{treeRate}$.

The terrain is generated based on a height map, which is calculated using this formula:

$$z = \begin{cases} & z_{ocean} \text{ pro } e_{ocean} > 0 \\ z_{base} + \min(z_{rivers}, \max(z_{desert}, z_{hills}, z_{mountains}, z_{plains})) \text{ pro } e_{ocean} <= 0 & \end{cases} \quad (4.2)$$

z_{base} represents base terrain altitude to which further landscape elements are added, and is determined by a Perlin noise with relatively high octave size. z_{plains} and z_{hills} are

also Perlin noises. Other parameters will (informally) elaborated in the following sections. Majority of these values is adjusted (by multiplying with a value derived from e_{ocean}) so that they converge to zero near seashore. Voxel types on the ground level are determined by the e_{XX} coefficients and based on the heightmap gradient.

4.4.1 Mountains

Mountains are generated by compositing two 2D Voronoi diagrams and one 2D Perlin noise. The principle of mountain terrain generation was demonstrated on Image 2.13 in Section 2.1.3: the terrain height increases with the distance from the nearest point in the Voronoi diagram, creating sharp edges resembling the mountain range. This principle is applied twice in two scales: the larger scale creates mountain peaks and ridges, the smaller scale adds secondary rock folding. Perlin noise is then added for adding more irregularities.

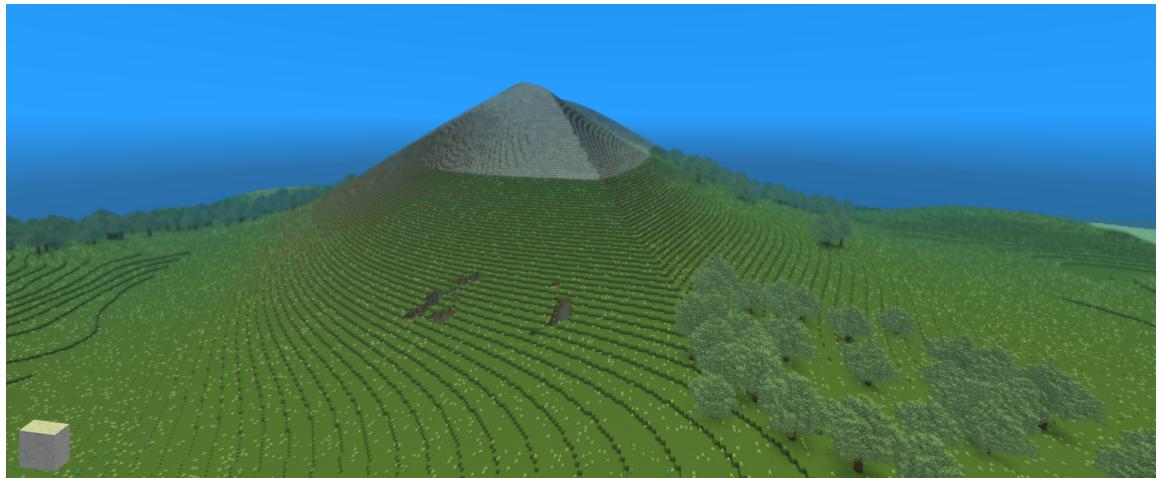


Figure 4.9: Three components used in mountain generation: Voronoi diagram for mountain peaks and ridges, Voronoi diagram for secondary, smaller folding and Perlin noise for adding irregularities (sequentially applied from top to bottom)

4.4.2 Deserts

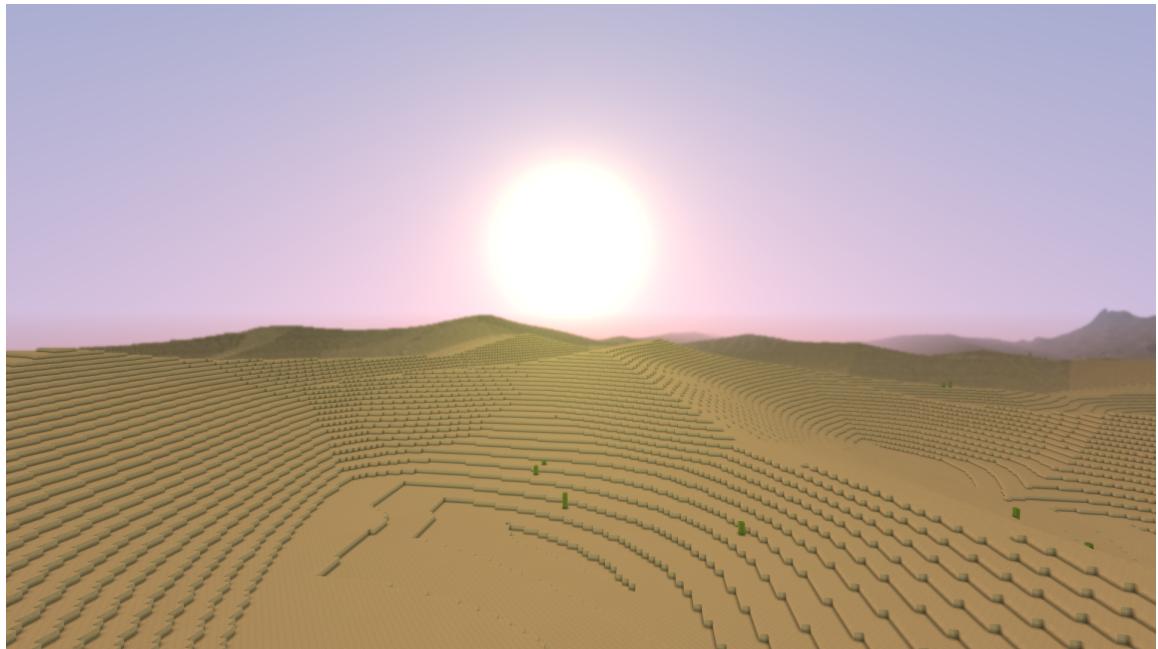


Figure 4.10: Desert

Deserts are also constructed by combining Perlin noise and Voronoi diagrams; in this case the Voronoi diagrams are used to modelling dunes. A maximum from two Voronoi diagrams with the same octave size is calculated to reduce visibility of pattern a single Voronoi diagrams would create.

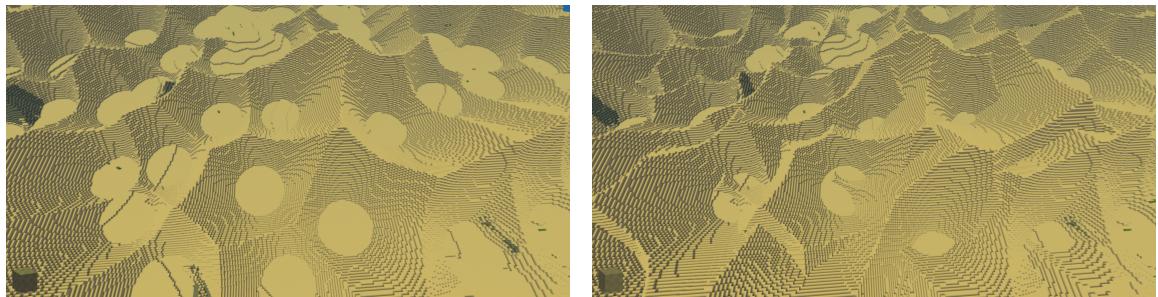


Figure 4.11: Dunes generated using single (left) and two (right) Voronoi diagrams (reduced scale)

4.4.3 Rivers

Rivers are generated using Perlin noise. The absolute value of a Perlin noise generates a pattern with sharp minima around zero. Rivers are generated in areas of these minima.

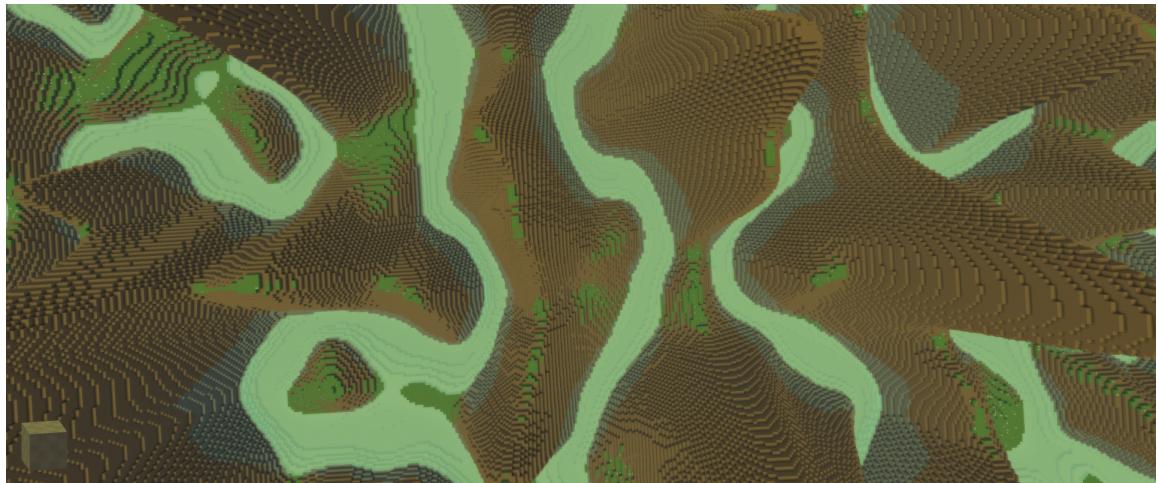


Figure 4.12: Absolute value of Perlin noise visualisation. Rivers are generated by thresholding this value.

All rivers are generated on the sea level. The height of the surrounding landscape is sequentially reduced around riverbeds (Based on extended values of the noise the rivers were generated from).



Figure 4.13: Reducing the altitude around rivers, so their surface can be on the sea level.

4.4.4 Trees

Trees are relatively simple from the mathematical description point of view. Every tree has height randomly assigned from a defined interval. The treetop size is then derived from the tree height, leaves are generated with a probability decreasing with the distance from the tree (*manhattan distance*). Tree distribution is based on Voronoi diagrams; the terrain is split into a grid (2D), every grid area can contain 0–N points, each point represents one tree.



Figure 4.14: Forest

4.4.5 Caves

In cave generation, two 3D Perlin noises are used: the first noise defines areas where the caves can occur (high octave size), the second noise generates the very caves (smaller octave size). Both noises have a threshold defined, a cave is generated only when both noise values are above their respective thresholds. In order to not make cave entrances too extensive, caves near the surface are reduced in some areas using a 2D Perlin noise.

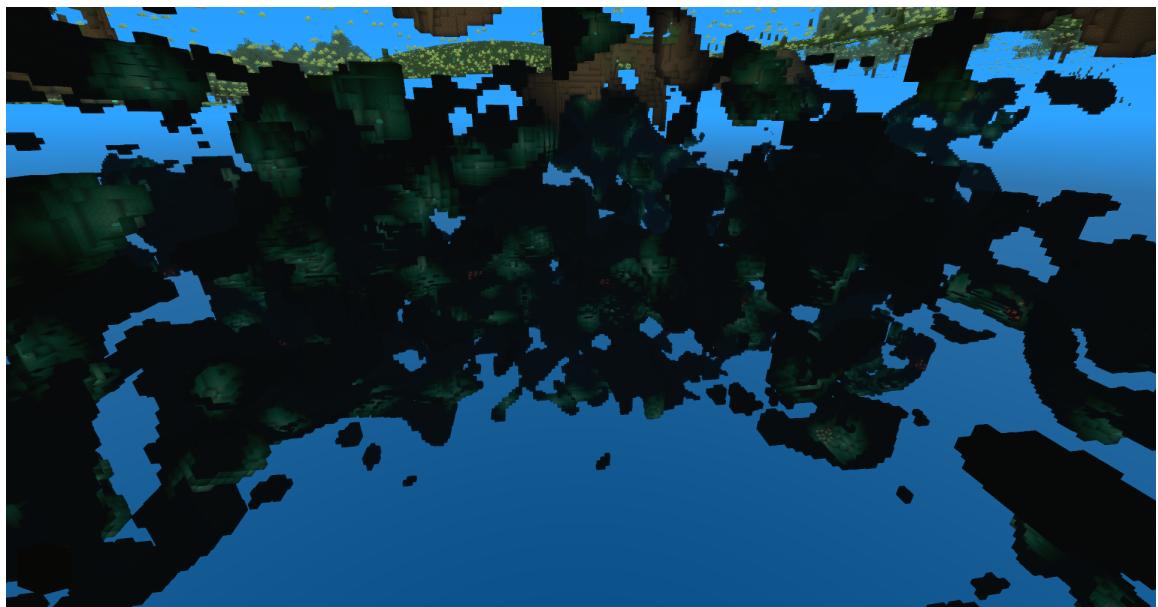


Figure 4.15: Procedurally generated cave complex

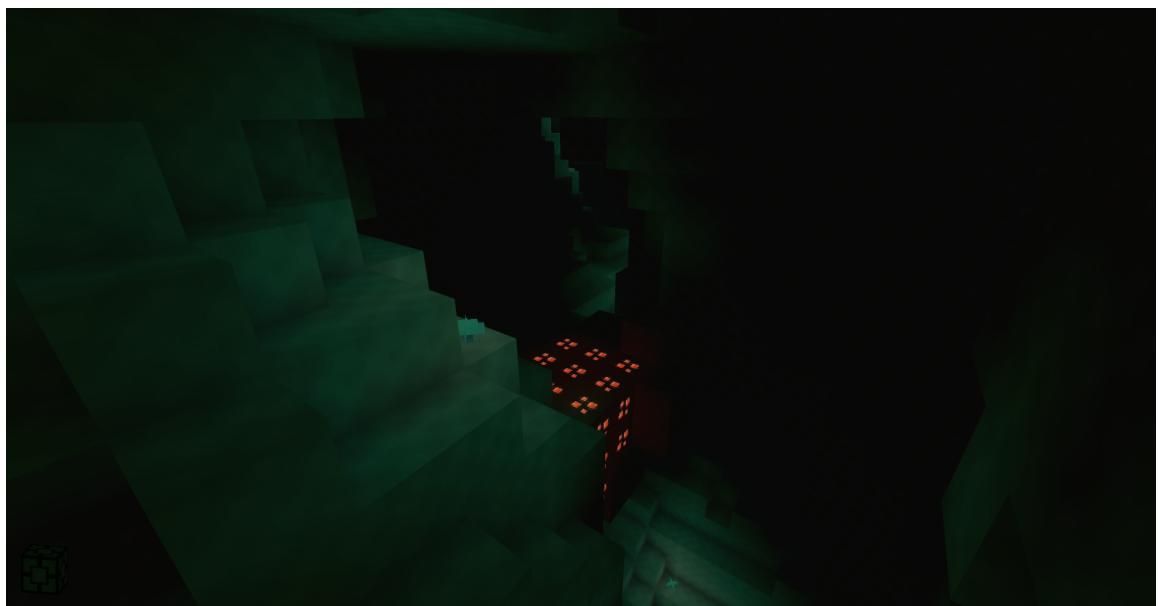


Figure 4.16: View from inside of a cave

Chapter 5

Implementation

As a part of this thesis, an application was created that demonstrates methods mentioned in this document. The source code was written in the D programming language. OpenGL in version 4.6 was selected as a graphics API (library `bindbc-opengl` was used for binding). From the features introduced in this version, anisotropic filtering (`ARB_texture_filter_anisotropic`) and vertex shader `gl_DrawID` (`(ARB_shader_draw_parameters)`) features were utilized. From modern OpenGL features used in the application, *compute shaders* (GL 4.3) *bindless textures* (not a *core feature*) or *direct state access* (GL 4.5) can be mentioned.

For image loading, window and user event management and text rendering, the library `derelict-sfml2` was used. `d2sqlite3` was selected for SQLite interface. Graphic resources were taken from the “C-tetra texture pack”¹ published under CC BY-NC 4.0² license (free for non-commercial purposes, original author must be mentioned).

5.1 World representation

Classes related to world representation are listed in the following diagram:

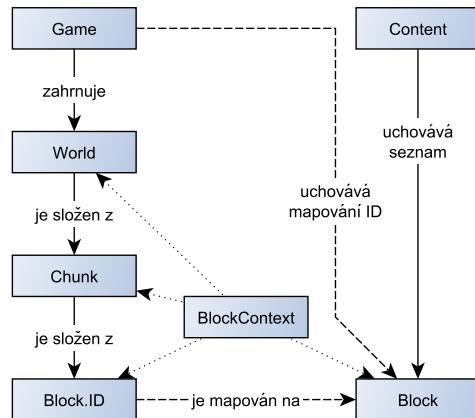


Figure 5.1: Relationship diagram of classes related to world representation

¹https://www.planetminecraft.com/texture_pack/16x-c-tetra-1-13/

²<https://creativecommons.org/licenses/by-nc/4.0/>

Class `Block` represents various voxel types. Its methods are used to define voxel properties – appearance, opacity, light emission, collision model and so on. New voxel types can be defined by subclassing the `Block` class or alternatively they can be constructed from various components (there are components defined for various voxel shapes, collision models and so on) using the classes `ComponentBlock` and `BlockComponent_XXX`. The class `Content` then contains a database of all registered voxel types.

The class `Game` represents the game backend. The design considers a possibility to have multiple worlds in a single game, however the current implementation only supports single world. `Game` also manages a database file for storing the world on disc and keeps mapping to the two-byte *block ID* value (see Section 4.1) to the `Block` class instances.

The class `World` represents a single world. That includes loading a swapping individual chunks to/from memory and providing access to them.

The world `Chunk` represents a single chunk (the term *chunk* is described in Section 4.1). It contains a linear array of blocks the chunk consists of.

The class `BlockContext` is used to represent any single voxel in a world. It works as a “pointer” to the block; it contains information of the location of the block (world, position), a reference to the `Block` class and so on. Instance of this class is passed as a parameter for all methods in the `Block` class, for example the method for rendering a block is defined as

```
1| void b_staticRender(BlockContext ctx, BlockRenderer rr);
```

An example code for creating a stone block on the position (0, 10, 0) is

```
1| scope BlockContext ctx = new BlockContext(world, WorldVec(0,10,0));
2| content.block.stone.b_construct(ctx);
```

For representing position of a voxel in a world, the structure `WorldVec` is used; it is a 3D vector with `int` component type (four bytes on any platform).

5.2 World generation and storage

The world generation and storage subsystem is based on these classes:

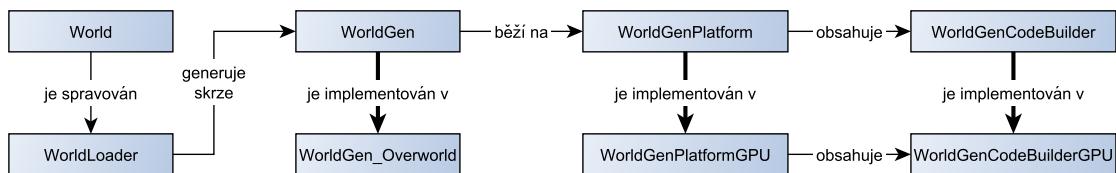


Figure 5.2: Relationship diagram of classes related to world generation and storage

The class `World` manages only chunks stored in memory. If a chunk not present in the memory is requested, the request is passed to the `WorldLoader` class, running on a dedicated thread. The class then either loads the chunk from the SQLite database (if it has been already generated) or manages its generation. Chunks no longer required to be kept in the memory are also passed to this class and the class manages storing them to the database.

The system is designed in a way so that chunks that are to be kept loaded in the memory have to be periodically requested for. Chunks that haven't been requested for a defined number of seconds (5 seconds) are passed for unloading. For the purpose of requesting

chunks, functions `World.maybeChunkAt(WorldVec pos)`, `World.maybeLoadChunkAt(WorldVec pos)` and `World.chunkAt(WorldVec pos)` are used. The first function mentioned returns the chunk only in the case that it is already loaded into memory. It does nothing otherwise and returns `null`. The second function (`maybeLoadChunkAt`) also returns `null` if the chunk is not already loaded, but it also requests the chunk to be loaded asynchronously. It also resets the chunk unload timer. The third function (`chunkAt`) synchronously loads the chunk to the memory if it is not present (so it can block), so it always returns a loaded chunk. Chunks loaded to the memory are called *active*.

Chunk generation requests are passed to the `WorldGen` class. The class is intended to be subclassed where individual subclasses are meant to define various terrain generation algorithms. In this paper only one generator, `WorldGen_Overworld`, is implemented, generating an Earth-resembling terrain. The world generation system is designed to support having multiple platform-dependent implementations. For this thesis a GPU-accelerated generation platform was implemented in classes `WorldGenPlatformGPU` and `WorldGenCodeBuilderGPU`. The possibility for multiple platforms was designed so that eventual multiplayer servers could run on platforms not supporting GPU acceleration.

The world generation is realized by multiple 2D and 3D passes that can be combined in arbitrary order. The passes have a functional character – the calculations are done separately for each pixel (or voxel) and in each pass it is only possible to read data from previous passes. Individual passes are represented by the class `WorldGenCodeBuilder` that also provides an interface for defining their behaviour. Terrain generators are defined using this interface, so although the world generation is realized on GPU using *compute shaders*, the world generator programming is done in the D language. However there is a disadvantage for this approach – because the D language does not provide enough flexibility for overloading of some operators (namely comparison, assign and logical `&&` or `||`), the generator code writing is not as comfortable. Compared to compute shaders, the interface approach can automatically generate structures for passing data between generation passes, making the multi-pass code writing relatively simple.

```

1 with (platform.add2DPass()) {
2     auto seaLevel = c(seaLevelVal);
3     auto mountainess = clamp01(perlin2D(256, [c(0.25), c(0.5), c(1)].x + 0.2);
4     auto elevationZ = 64 * clamp01(0.2 + perlin2D(baseOctave, coefs).x);
5
6     // Big mountain peaks
7     auto bigPeakVoronoi = voronoi2D(1024, 8);
8     auto bigPeakVal = pow((bigPeakVoronoi.x - 0.05) * 5, c(2)) * mountainess;
9
10    // Small mountain peaks
11    auto smallPeakVal = pow(voronoi2D(64, 4).x * 3, c(4));
12
13    // Lil' perlin to smooth things out
14    auto smoothPerlin = perlin2D(16, [c(0.3), c(0.2), c(0.1), c(0.1)].x);
15
16    auto mountainsZ = max(c(0), bigPeakVal * 128 + smallPeakVal * 32 +
17                           smoothPerlin * 32 - 16) * mountainess;
18
19    set2DDData("groundZ", seaLevel + elevationZ + mountainsZ);
20    finish();
}

```

Kód 5.1: Example of world generator programming; the code is an excerpt from the first (2D) `WorldGen_Overworld` generator pass.

The Perlin noise and Voronoi diagram GPU calculation functions utilize thread cooperation. However more detailed algorithm description does not fit into this thesis in terms of space.

The `WorldGen_Overworld` generator uses five passes ($2 \times 2D$ and $3 \times 2D$). Their function is described in the following diagram.

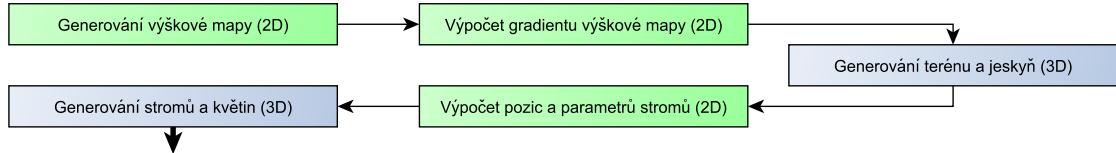


Figure 5.3: `WorldGen_Overworld` generator pass diagram

The vegetation is generated in a separate 3D pass, because it is necessary to prevent the plants being generated above cave entrances where the terrain does not correspond with the height map. For that, the first 3D pass provides information to the further passes of whether there is a cave entrance in the height map level. 3D passes can provide a 2D output, but it can be written to only by a single invocation in a column; otherwise the behaviour is not defined. In the case of cave generation, only the invocation in the height map level writes to the 2D output.

During each chunk generation a 3×3 area around the chunk is generated: this redundancy is required for example for cases where leaves are to be generated for a tree in a neighbouring chunk and it is necessary to determine if the tree is actually generated or not

because of cave entrances. In the chunk generation, the locality principle determined by chunk size is manifested, as mentioned in Section 4.1.

5.3 World rendering

The following classes are related to world rendering:

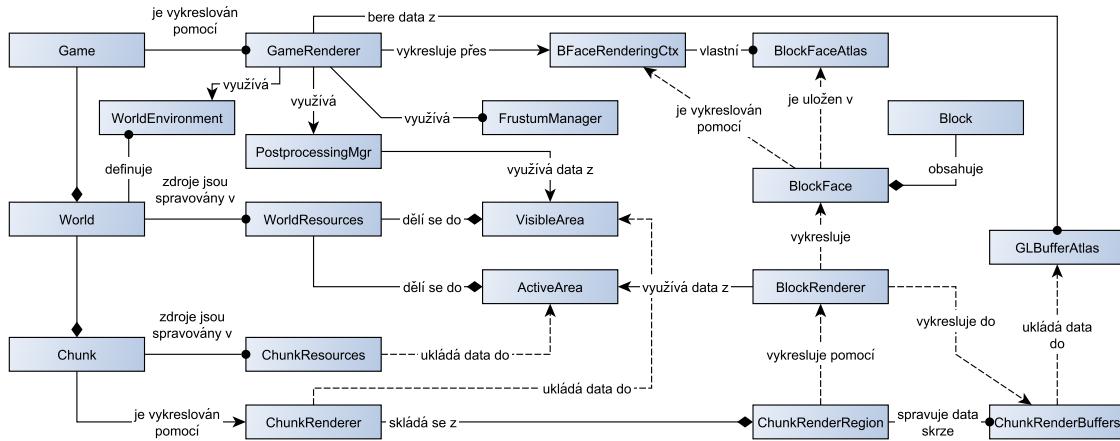


Figure 5.4: Relationship diagram of classes related to world rendering

5.3.1 *Block faces*

Faces of blocks are rendered using a standard rendering pipeline as triangles. Blocks in the application do not have to be strictly cubes, they can have any shape.



Figure 5.5: Various block shapes

The entire rendering is however based on textured rectangles (further referred to as *block faces*). Individual block face appearances are represented by instances of the `BlockFace` class. The appearance is not determined only by the texture, but also by other rendering-related properties:

- Alpha channel behaviour:
 - The alpha channel can determine thresholded opacity (*alpha testing*). This method is fast for rendering but it prohibits using *mipmapping*. It is used to render trees and vegetation in the application.
 - The alpha channel can determine opacity (*alpha blending*). In that case the block face is rendered using slower methods that support transparency (*depth peeling*).

- The alpha channel can determine level of the pixel light emission (*glow map*).
- *Cull facing*: some faces (for example in vegetation blocks) have to be rendered from both sides whereas the faces of opaque cubes can be only rendered in the outwards direction from the cube center.
- *Wrapping* denotes if the texture is to be approached as a seamless repeating pattern (this can be noticeable on higher mipmap aggregation levels).
- Vertex animation. The face can be animated in several ways; it can wave in the wind, either on all eight vertices (tree leaves) or only on the top part (flowers) or it can be waving as a liquid surface. These animations are implemented on the *vertex shader* level.
- Texture resolution.

Block faces sharing the same configuration are rendered the same way using OpenGL. The OpenGL configuration and shaders used for such rendering are represented by class `BlockFaceRenderingContext` (further referred as *face context*). All shaders are compiled from the same source code located in `res/shader/render/blockRender.(vs|fs).glsl`, the actual configuration being passed by a series of `#defines` to the compiler. Every *face context* contains multiple shader program variants: for standard rendering, for rendering to a shadow map (no fragment color data needed) and for rendering for *depth peeling* (with additional *near depth test* implemented).

Individual *block faces* are stored in a texture atlas (class `BlockFaceAtlas`); every *face context* has its own atlas. Internally the faces are stored in `GL_TEXTURE_2D_ARRAY`.

5.3.2 Rendering data representation

Every chunk is vertically divided into multiple regions, represented by class `ChunkRenderRegion`. These regions are then the smallest rendering unit (either the region is rendered as a whole or it is not rendered at all). For every *face context* the region keeps a set of buffers containing the rendering data. The data is updated only before the first show or if the chunk changes.

On the GPU the data is stored in a single big buffer managed by class `GLBufferAtlas`. A custom memory management system is implemented for the buffer atlas. The system does not support reallocation (it is not even needed as the data is always prepared from scratch on the CPU and then uploaded to the GPU) and the allocation always tries to use the smallest free memory region that is simultaneously large enough for the allocation.

For every vertex, position, uv texture coordinates and normal is stored. Because the normal is the same for all three vertices of a triangle, an optimization is introduced that allows storing the normal only once per triangle: the normal buffer is connected three times, once for every normal dimension (N_x , N_y , N_z). The *stride* is set to zero for all three connections, so each connection sequentially accesses data of all three dimensions of a normal. *Offset* is however set for each connection in a way so that the last vertex of a triangle (resp. *provoking vertex*, that is implicitly set to `GL_LAST_VERTEX_CONVENTION`) always receives the correct data for all three components. Attributes of the components are marked as `flat` in the shader, so fragment shader always receives data from a provoking vertex. The disadvantage of this approach is that the vertex shader only receives valid normal data for a single vertex per triangle, so it is impossible to perform per-vertex normal operations

(for example rotating the normals based on block animations that are realized in the vertex shader).³

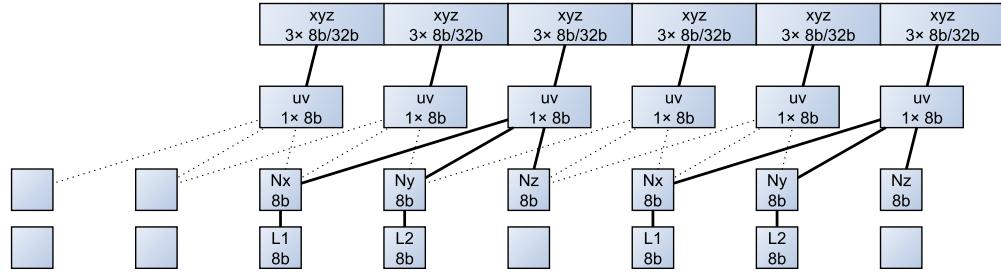


Figure 5.6: *Vertex arrays* structure passed to *vertex shader*

The application keeps data for a lot of vertices (in a testing scene there were 24 millions vertices measured with 32-chunk view distance), so a good memory utilization is desired. Normals are stored as a $3 \times 8b$ vector. Because block faces are stored in a texture array, uv coordinates for a vertex will always be either 1 or 0, so it can be fit into a single byte. The texture layer index is the same for all three vertices in a triangle so it can be spread in the same way as the normal (one byte for each vertex, data is then composed together and passed to the fragment shader as a single *flat* parameter).

Vertex position components are also stored in a single byte; position of a region in a world is passed additionally as a constant for the entire buffer. Because of that, the chunk has to be vertically divided into at least two regions, because it would not be possible to represent coordinates of top faces of the upmost voxels in the chunk ($z = 255$ would represent the bottom face vertices, the byte would overflow for the top face vertices).

Because not all block types have the faces aligned with the voxel grid (for example cactus or wheat; although flowers have an atypical shape, their coordinates are actually aligned with the grid, only the triangles go diagonally across a voxel as demonstrated in Image 6.5), two sets of buffers were implemented: in the first set the coordinates are stored as $3 \times 1B$, in the latter set the coordinates are stored as 3×32 -bit floating point numbers. Occurrence of these non-aligned blocks is much lower however, so the capacity of the latter buffer set can be much lower.

³After finishing the thesis, it was discovered that using a separate OpenGL buffer for normal data (and texture layer data, as mentioned below) and then accessing the buffer using `gl_VertexID % 3` as index could be a better option.



Figure 5.7: Cactus and weah blocks, with *faces* not aligned with the voxel grid

Memory requirements per vertex are then $3 \times 1 + 2 \times 1 + 1 = 6$ bytes with 8b coordinates and $3 \times 4 + 2 \times 1 + 1 = 15$ bytes with 32b floating point coordinates. In a testing scene containing 24 million 6B vertices and 100 thousand 15B vertices, the vertex data consume 145 MB of VRAM. If all the coordinates were stored in the 32b floating point format, the memory requirements would be 360 MB. Those are not large numbers for modern GPUS, however the buffers are not the only memory-demanding resources used by the application. Additionally, having smaller data decreases the memory bandwidth requirements, that is the most common bottleneck in GPU rendering.

5.3.3 Sestavování dat pro vykreslování

Základem vykreslování je projít všechny bloky v regionu a nad každým blokem zavolat funkci `b_staticRender(context, renderer)`, která zajistí vložení dat reprezentujících příslušné *faces* do příslušných *bufferů*. Tento přístup je dále optimalizován:

1. Sousedící stěny krychlových bloků, které jsou těsně vedle sebe, nelze vidět, tedy se ani nemusí vykreslovat. Vynechání těchto stěn přináší extrémní zrychlení, protože namísto všech bloků lze vykreslovat pouze tenkou “skořápku” povrchu terénu. Počet vykreslovaných primitiv takto klesne o několik řádů.



Figure 5.8: Pohled “zevnitř” terénu; aplikace vykresluje pouze stěny, které jsou viditelné z volného prostoru

2. Sousedící stěny stejného typu, které jsou v jedné rovině, lze agregovat do jednoho vykreslovacího primitiva.
3. Iterování přes všechny bloky v chunku a volání funkce `b_staticRender` pro každý z nich je pomalé. Pokud víme, že voxel je ze všech stran obklopen neprůhlednými krychlemi (nebo pokud jsou všechny jeho viditelné stěny agregovány, takže jejich vykreslení zajišťuje jiný blok), můžeme ho zcela přeskočit.

Při vykreslování chunku je třeba mít v paměti načteny také jeho sousední chunky (aby šly počítat optimalizace sousedících stěn i pro bloky na okraji chunku, dále také kvůli výpočtu osvětlení), proto se klasifikace chunků dále rozvádí na *neaktivní* (nenačtené v paměti), *aktivní* (načtené v paměti, ale nevykreslované) a *viditelné*. Aby mohly proběhnout výpočty pro vykreslování, musí být chunk *viditelný* a jeho sousedi v 8-okolí musí být *aktivní*. Viditelnost chunku musí být periodicky žádána, podobně jako musí být žádána aktivnost. Všechny chunky v určité vzdálenosti od hráče (*manhattan distance*, dle aktuálního nastavení dohledové vzdálenosti) jsou v každém snímku obnovovány jako *viditelné*.

Všechny výše zmíněné optimalizace jsou akcelerovány na GPU. Z toho důvodu je v GPU uchovávána textura 3D textura s *block IDs* (jedná se o 1 : 1 kopii dat z CPU) a buffer s informacemi o vlastnostech jednotlivých typů bloků. Data *block IDs* jsou agregována do 3D textur o velikosti $32 \times 32 \times 256$ bloků (tedy 2×2 chunků). Tyto oblasti jsou v CPU reprezentovány třídou `ActiveArea` a jsou centrálně spravovány třídou `WorldResources`.

Block IDs na GPU jsou uchovávány pro všechny *aktivní chunky*. Ke každému typu bloku je na GPU uloženo, které z jeho šesti stran jsou zcela zakryté neprůhlednou stěnou. Při sestavování dat pro vykreslování je pak spuštěn *compute shader*, kde je na základě těchto informací v jednom paralelním kroku vypočteno, které stěny každého bloku v chunku jsou viditelné, a které ne (viditelné jsou ty stěny, jejichž odpovídající sousedi nemají na stejnou stranu neprůhlednou stěnu). Do výstupního *storage bufferu* je pak pomocí atomického čítače serializován seznam souřadnic těch voxelů, které nejsou prázdnné a které mají alespoň jednu stěnu viditelnou. Data tohoto *bufferu* jsou předána CPU, který pak navrácené bloky projde a zavolá pro ně `b_staticRender`. Jedním z parametrů této funkce je třída `BlockRenderer`, která poskytuje metody typu `drawFace` a `drawBlock` zajišťující sestavení korektních dat pro vykreslování.

Stejný *compute shader* zajišťuje i agregaci sousedících stěn. Pro každou stěnu každého voxelu je určena velikost obdélníku, ve kterém jsou všechny stěny viditelné a stejného typu. Tato data jsou pak předána CPU, na kterém třída `BlockRednerer` zajistí konstrukci primitiva správných rozměrů. Informace je předána pouze bloku v jednom z okrajů obdélníku; ostatním blokům je předán rozměr 0×0 , který `BlockRednerer` informuje, že vykreslení dané stěny zajišťuje jiný blok. Maximální aggregace je 8×8 stěn, což je dáno velikostí pracovní skupiny *compute shaderu* ($8 \times 8 \times 8$) a tím, že uv souřadnice jsou uloženy ve 2×4 bitech (rozsah přípustných hodnot je 0–15, pro $16 \times$ agregaci by byl třeba rozsah 0–16). Vizualizaci aggregace lze v aplikaci aktivovat nastavením vizualizovaných dat (první *combobox* shora) na “Aggregation”. Byly implementovány tři metody aggregace (lze mezi nimi přepínat 3. *comboboxem* zespod v menu):

- **Lines** aggreguje nejprve do maximální míry v jednom směru (2 směry aggregace v rovině stěny) a poté spojuje obdélníky stejné délky ve směru druhém.
- **Squares** aggreguje střídavě v jednom a ve druhém směru v mřížce, jejíž velikost se v každém kroku zdvojnásobí.
- **Squares ext** přidává k algoritmu `squares` dodatečný krok, který následně spojuje i obdélníky nezarovnané do mřížky.

Porovnání efektivity jednotlivých metod je realizováno v oddílu 6.2.3. Problémem aggregace je, že vytváří tzv. *T-junctions* (tedy situace, kdy na sebe trojúhelníky nenavazují hranami, ale vrchol jednoho trojúhelníku se nachází na hrani druhého trojúhelníku), které způsobují artefakty v podobě průhledných pixelů (protože OpenGL garantuje přesnou návaznost trojúhelníků pouze v případě, když na sebe trojúhelníky navazují vrcholy). Pro omezení těchto artefaktů byl vytvořen efekt *postprocessingu*, který detekuje bodové díry v obraze a vyplňuje je z okolních pixelů.

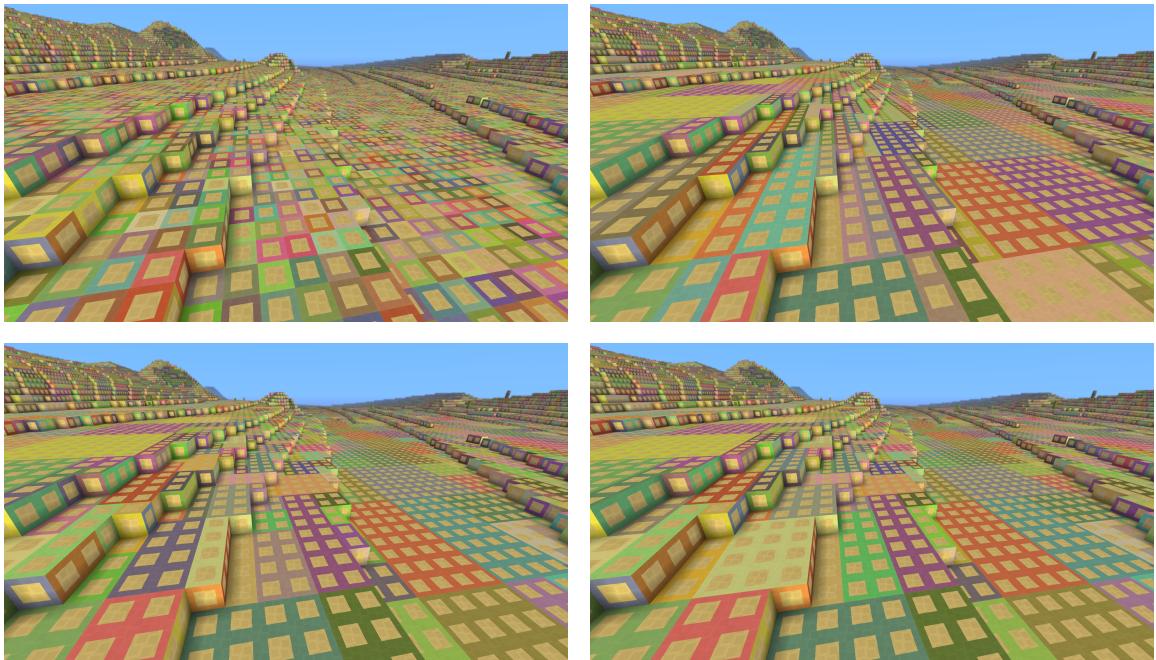


Figure 5.9: Metody agregace stěn bloků: bez agregace, `lines`, `squares`, `squares ext.` Stěny se stejnou barvou označení jsou agregovány do jednoho vykreslovacího primitiva. Vizualizaci agregace lze v aplikaci aktivovat zvolením položky “Aggregation” v prvním *comboboxu* shora.

Výpočty na GPU probíhají asynchronně, na CPU je uchováván kruhový buffer se synchronizačními bariérami jednotlivých výpočtů. Prevence zahlcení GPU těmito výpočty je realizována zavedením maximálního počtu položek v kruhovém bufferu.

Pravidla optimalizace je dále třeba vhodně rozšířit pro průhledné bloky. Proto byla přidána další verze pravidla: stěny neprůhledných bloků jsou směrem k průhledným blokům vždy vykresleny. Stěny mezi průhlednými bloky nejsou vykresleny pouze v případě, že se jedná o stejný typ bloku.



Figure 5.10: Průhledné bloky a jejich interakce s bloky neprůhlednými

5.3.4 Vykreslování, *frustum culling, depth peeling*

V předchozích kapitolách byly popsány metody, jakými se sestavují data pro vykreslování: pro každý region v chunku (`ChunkRenderRegion`) a pro každý *face context* (`BlockFaceRenderingContext`) je alokován prostor v atlasu bufferů, ve kterém jsou uloženy informace o vykreslovaných vrcholech. Atlasy jsou celkem dva, jeden pro vrcholy s 8bitovými souřadnicemi (`GL_UNSIGNED_BYTE`) a jeden pro vrcholy s 32bitovými souřadnicemi v plovoucí řádové čárce (`GL_FLOAT`).

Pomocí *compute shaderu* je vypočten seznam regionů, které jsou pro kameru viditelné. Tuto funkcionalitu realizuje třída `FrustumManager`. Každé invokaci odpovídá jeden chunk, invokace iterativně projde všechny regiony v chunku a seznam viditelných regionů serializuje pomocí atomického čítače do *storage bufferu*. Region je považován za viditelný, pokud alespoň jeden z jeho osmi hraničních vrcholů je před *near* rovinou, alespoň jeden vrchol je vpravo od levého okraje obrazovky, alespoň jeden z vrcholů je vlevo od pravého okraje obrazovky a obdobně pro horní a spodní okraj. Detekce těchto podmínek je realizována projekcí souřadnic do prostoru obrazovky (pomocí projekční matic) a následným porovnáním vůči 1 nebo -1 ; pakliže je komponenta w při projekci menší než nula, jsou výsledky porovnání bočních okrajů invertovány.



Figure 5.11: Demonstrační *frustum cullingu*; červené čáry v horním obrázku označují hrany a střed projekčního jehlanu. Zelený čtyřúhelník označuje oblast pokrytu *shadow mapou* (pouze orientační). V aplikaci lze do pohledu shora přepnout klávesou F3.

Výpočty jsou současně provedeny pro projekční matici z pohledu hráče a pro matici používanou při *shadow mappingu*. Veškeré vykreslování je realizováno $c \cdot n$ voláními funkce `glMultiDrawArrays`, kde c je počet *face contexts* v aplikaci (8) a n je počet buffer atlasů (2). Jelikož se využívá jeden velký buffer, je možné vykreslit všechny chunk regiony jedním voláním, stav se musí měnit pouze pro různé *face contexts* (jak již bylo uvedeno, různé kontexty používají různé shadery, mají různé atlasy textur a mohou se lišit v nastavení jako

např. `GL_CULL_FACE`). Výstupem vykreslování jsou hloubková textura, *albedo* a normálová textura. Aplikace zavádí *deferred shading*, tedy stínování je realizováno v *postprocessingu*.

Vykreslování průhledných objektů je řešeno *depth peelingem* s až třemi vrstvami. Všechny průhledné objekty jsou tedy vykresleny několikrát; míchání barev (`GL_BLEND`) je při vykreslování vypnuto, takže výstupem je vždy průhledný objekt nejblíž kameře. Druhý a třetí průchod však provádí dodatečný *near depth test* (implementovaný ve *fragment shaderu*), který zahazuje ty fragmenty, které jsou blíže ke kameře (nebo ve stejné vzdálenosti) oproti fragmentům z předchozího průchodu.

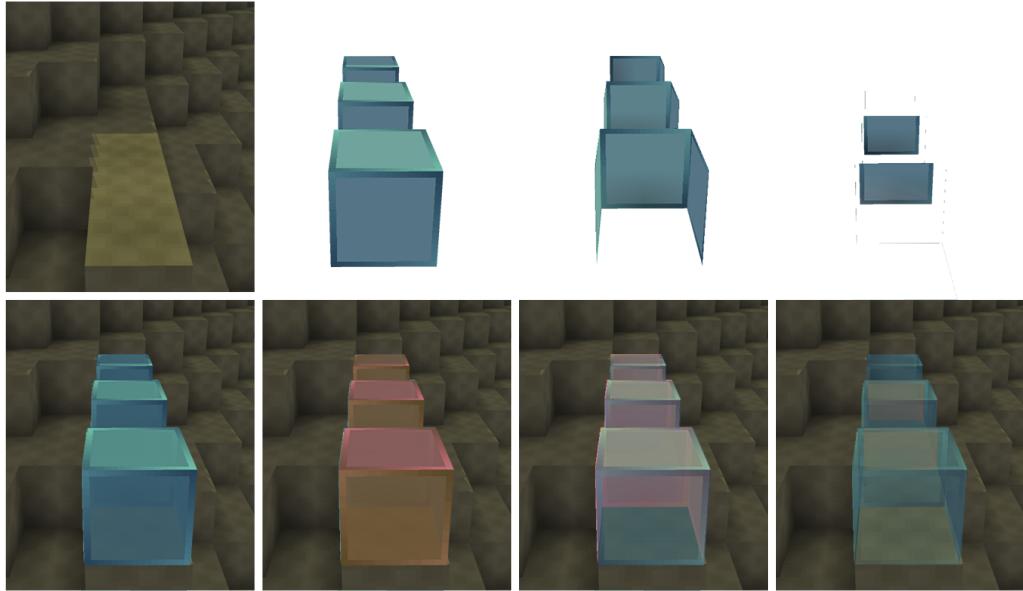


Figure 5.12: Třívrstvý *depth peeling*, vizualizace jednotlivých vrstev

5.3.5 Osvětlení

Data osvětlení jsou uchovávána v 3D textuře na GPU. Svět je rozdělen do oblastí o velikosti $128 \times 128 \times 256$ voxelů, každá tato oblast odpovídá jedné 3D textuře a je reprezentována tří-dou `VisibleArea`. Každá ze čtyř komponent (RGB + denní světlo) je uložena ve čtyřech bitech. Osvětlení je aplikováno jako průchod v *postprocessingu*. Protože OpenGL nativně nepodporuje texturu, kde jsou složky uloženy ve čtyřech bitech (a interně texturu `GL_RGBA4` ukládá jako `GL_RGBA8`, což zbytečně plýtvá místem), jsou všechny složky uloženy v jedné komponentě `GL_R16UI` a korektní interpolace jednotlivých složek při vzorkování je implementována ručně. Aby se při vzorkování dala využít funkce `textureGather`, která není podporována pro 3D textury, je textura reprezentována jako pole 2D textur, kde vrstvy odpovídají souřadnici z (výška).

Do 4 bitů lze uložit celkem 16 různých úrovní osvětlení (0–15), což souhlasí s omezením dosahu jevů danou velikostí chunku, který má v horizontální rovině rozměr 16 voxelů. Zvýšení maximálního možného dosahu světla by tedy vyžadovalo zvětšení velikosti chunku, a kromě toho také přechod z `GL_R16UI` na `GL_R32UI`, což by zdvojnásobilo paměťové nároky.

Celkově je tedy na GPU kromě dat pro vykreslování trojúhelníků (145 MB pro dohl. vzd. 32 chunků) třeba ukládat dva bajty s *block ID* a dva bajty pro údaje o osvětlení na každý voxel. Při dohledové vzdálenosti 32 chunků ($(2 \cdot 32 + 1)^2 = 4\,225$ chunků, což odpovídá 276 889 600 voxelů) je tedy potřeba přibližně 1,2 GB grafické paměti; v praxi

to může být kolem 2 GB kvůli dalším použitým zdrojům, nedokonalému zarovnání (kdy není využita celá alokovaná textura) a kvůli tomu, že *block IDs* se uchovávají pro všechny *aktivní* chunky, tedy pro oblast s průměrem o dva chunky větší, než je dohledová vzdálenost (protože všichni sousedi *viditelných* chunků musí být *aktivní*).

Výpočet dat osvětlení také probíhá na GPU; celulární algoritmus popsáný v sekci 4.3 je vhodný pro akceleraci na grafickém koprocesoru. Pro výpočet osvětlení se využívají i *block IDs* sousedících chunků (8-okolí). Výpočet probíhá v několika krocích (jeden krok odpovídá jednomu spuštění *kernelu*):

- V prvním kroku se nastaví iniciální hodnoty osvětlení na místa zdrojů světla a propaguje se denní světlo shora dolů. Tento běh probíhá ve 2D (každá invokace odpovídá jednomu sloupci).
- V dalších krocích se výpočty provádějí v diskrétních oblastech $8 \times 8 \times 8$ s využitím vláknové kooperace. Mřížka dělení do diskrétních oblastí je v každém běhu posunuta o 4 jednotky ve všech třech směrech, takže mřížka alteruje podobně jako okolí typu *margolus* u celulárních automatů. Takto je celkem provedeno pět kroků, což zajistí korektní propagaci i přes dělení výpočtů do diskrétních buněk.
- Jako výsledek se vždy použijí pouze data prostředního chunku. Data okolních chunků se zahodí, protože nejsou kompletní – nezohledňují zdroje světla od části jejich sousedů.

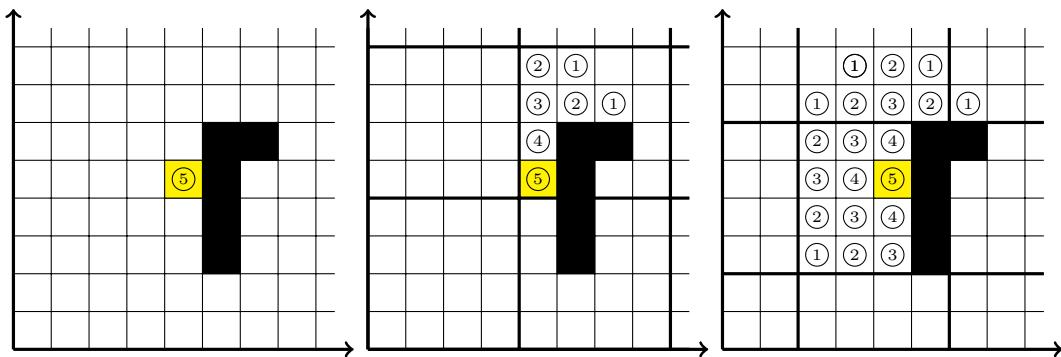


Figure 5.13: Šíření světla v alternujících diskrétních oblastech (2D, zmenšené měřítko)

Souhrnně jsou tedy k vykreslování chunků třeba tyto kroky:

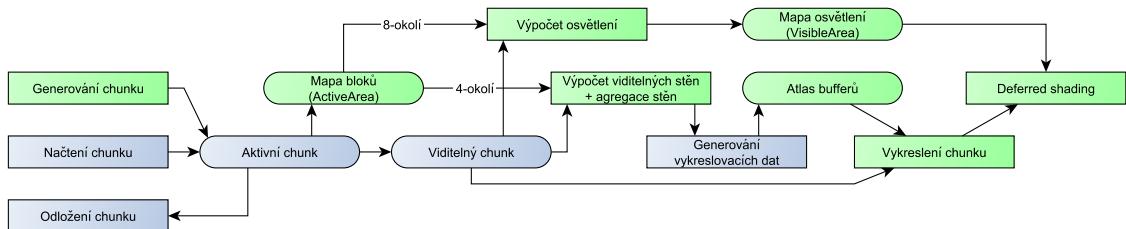


Figure 5.14: Souhrnný proces nutných k vykreslování. Zeleně jsou označeny prostředky a procesy vázané na GPU, modře na CPU.

5.4 Postprocessing

Postprocessing se skládá z následujících průchodů:

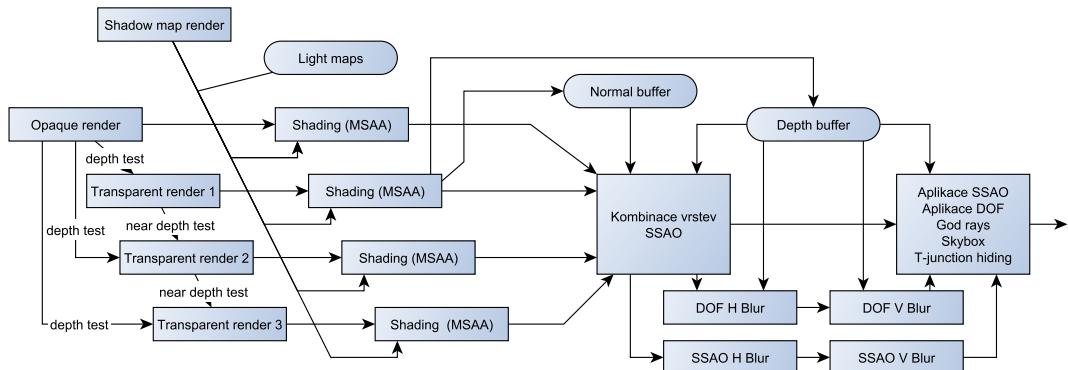


Figure 5.15: Průchody při *postprocessingu*

Postprocessing tedy obsahuje jeden lokální průchod pro každou vrstvu (neprůhledná + 3 vrstvy *depth peelingu*), dva globální průchody plus výpočty rozostření. V lokálních průchodech se počítá osvětlení (na základě inverzní projekční matice a hloubkové textury se pixely obrazovky promítou do světa a vzorkuje se 3D textura s osvětlením, jak bylo popsáno v oddílu 5.3.5) a *shadow mapping* pro sluneční světlo. Světelné mapy jsou *shaderu* předány formou *bindless* textur v uniformní paměti. Pokud by každému chunku odpovídala jedna textura, při dohledové vzdálenosti 32 chunků ($(32 \cdot 2 + 1)^2 = 4\,225$ chunků celkem) a velikosti ukazatele na texturu 8 B by bylo třeba 34 kB uniformní paměti, což překračuje minimální garantovanou velikost 16 kB. Proto byla data agregována do větších textur o velikosti $128 \times 128 \times 256$ bloků (8×8 chunků, spravované třídou *VisibleArea*, jak bylo uvedeno v oddílu 5.3.5), což redukuje paměťové nároky 64krát. Data pro vlastní emisi pixelů (*glow map*) jsou uložena v alfa kanálu normálové textury.

V aplikaci je implicitně aktivován $2 \times$ *multisampling*; lokální průchody v důsledku toho pracují s *multisample* texturami. Výstupem lokálních průchodů jsou již jednovrstvé textury se stínovaným obrazem. Z prvního *depth peeling průchodu* (tedy pro objekty nejblíž ke kameře) je pro další *postprocessing* exportována hloubková a normálová mapa. V případě vypnutého *depth peelingu* se mapy exportují z neprůhledného průchodu.

5.4.1 Shadow mapping

Shadow mapping využívá *percentage closer filtering* [23], vzorkuje se oblast 2×2 texelů pomocí jednoho volání funkce `textureGather` a výsledek se interpoluje (textura má nastaveno `GL_TEXTURE_COMPARE_FUNC` na `GL_LEQUAL`, takže normalizaci souřadnic a porovnávání vůči hloubkové mapě zajišťuje OpenGL).



Figure 5.16: Stín vrhaný sluncem, rozlišení *shadow mapy* 2048^2 , aplikován *percentage closer filtering*

Vzorkované souřadnice jsou mírně posunuty ve směru normály a porovnávaná hloubka je mírně snížena, aby se předešlo artefaktům na stěnách téměř rovnoběžných se slunečním světlem.

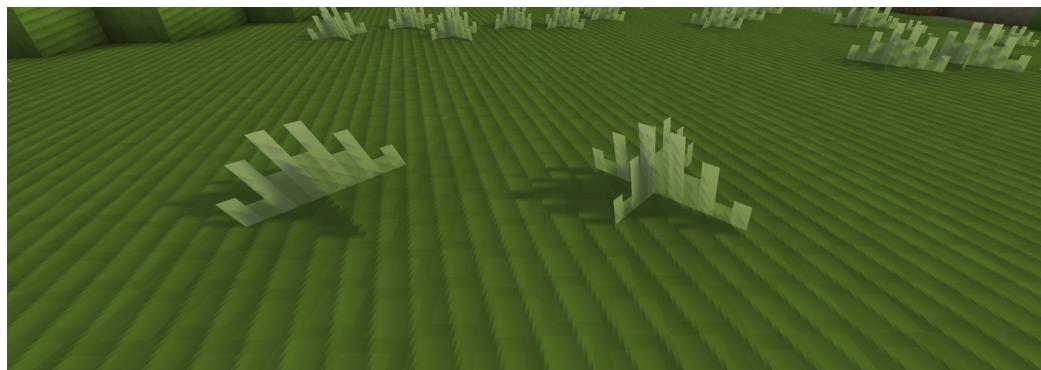


Figure 5.17: Artefakty v *shadow mappingu* při velkých úhlech mezi normálou povrchu a dopadajícího světla

Výpočet osvětlení vzniká aditivní kombinací 4 složek (*ambientní* složka, složka umělého osvětlení, *ambientní* složka slunečního světla a směrová složka slunečního světla), jak bylo uvedeno v rovnici 4.1. Pakliže je aktivovaný *shadow mapping*, směrová složka denního osvětlení se dále násobí s jeho výstupem.

Shadow mapping vyžaduje dodatečné vykreslení scény ortogonální maticí “z pohledu slunce”. Při vykreslování do stínové mapy je *face culling* zapnutý pro všechny kontexty, aby se předešlo např. prosvítání slunečního světla skrze stěny jeskyní. Pro kontexty, které

to vyžadují, se provádí *alpha testing*. Průhlednost stínů lineárně klesá se vzdáleností od kamery, ve vzdálenosti 50 bloků je už aplikován pouze standardní osvětlovací model. Je zavedena optimalizace upravující projekční matici tak, aby byla vykreslována pouze oblast před kamerou: do neoptimalizované matice je promítnuto všech osm vrcholů pohledového jehlanu kamery a matice je posunuta a škálována tak, aby vykreslovaná oblast odpovídala obalovému obdélníku promítnutých bodů. Do stínové mapy nejsou vykresleny průhledné bloky, v důsledku čehož nevrhají žádný stín.

5.4.2 Screen space ambient occlusion (SSAO)

V prvním globálním průchodu dochází ke kombinaci jednotlivých vrstev a k výpočtu *SSAO*. *SSAO* byla původně zamýšlena jako suplementární technika k „přirozenému“ *ambient occlusion*, které je důsledkem osvětlovacího modelu – ten dobře funguje pro plné krychle, ale nepokrývá detaily vznikající např. při vykreslování vegetace. Pro neuspokojivé výsledky a vysokou výpočetní náročnost je však *SSAO* ve výchozím nastavení deaktivované.

SSAO pro každý pixel sbírá 2 až 8 vzorků hloubky (dle vzdálenosti pixelu od kamery) v polokouli dané vzdálenosti a normálou pixelu. Vzorky jsou umístěny do 2D spirály včetně normále, hloubka je pak daná uniformním rozložením. Na efekt *SSAO* je aplikováno 9×9 Gaussovo rozostření se $\sigma = 3$.

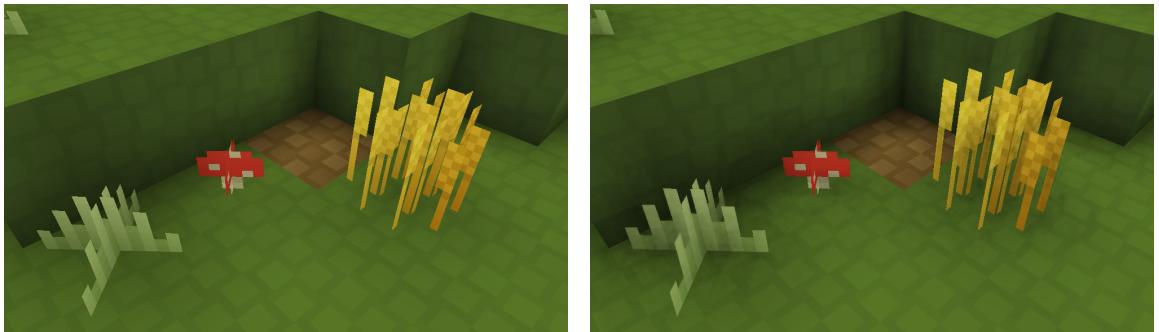


Figure 5.18: Scéna s deaktivovaným (vlevo) a aktivovaným (vpravo) *screen space ambient occlusion*

5.4.3 Depth of Field (DOF)

Aplikace neimplementuje efekt *depth of field* v takovém smyslu, jak byl popsán v oddílu 3.2.1, spíše se jedná o efekt rozostření při průchodu atmosférou. Rozostřeny jsou tedy vždy objekty nejdále od kamery. Implementace provádí rozostření výstupu prvního globálního průchodu a v druhém průchodu interpoluje tento rozostřený obraz s původním na základě hloubkové informace. Rozostření je realizováno dvěma průchody (horizontální + vertikální) bilaterálním (zachovávající ostré hrany, počítá se rozdíl mezi vzorky v hloubkové textuře) Gaussovým filtrem s jádrem o velikosti 9 pixelů ($4 + 1 + 4$) a $\sigma = 2$. V prvním globálním průchodu je obraz navíc na základě hloubky smíšen s barvou pozadí (*skybox* bez složek souvisejících se sluncem) a zprůhledněn na základě vzdálenosti od kamery, čímž vzniká atmosférický efekt.



Figure 5.19: Scéna s deaktivovaným *depth of field* a atmosférickým efektem



Figure 5.20: Scéna s aktivovaným *depth of field* a atmosférickým efektem

5.4.4 God rays

Pro krepuskulární paprsky byla zvolena metoda popsaná panem Wesleym, která efekt počítá ve *screen space*. Originální metoda pracuje s “emisní texturou”, která pro každý pixel ukládá “emisivity” od zdroje světla. Pokud je pozadí překryté popředím, je toto reflektováno i v emisní textuře a odpovídající pixely jsou černé. Pro každý pixel ve scéně se pak sbírá určitý počet vzorků mezi výchozím pixelem a zdrojem světla a vzorkované hodnoty se sčítají. Výsledný součet je přičten k výstupu.

Místo vzorkování po celé přímce je v této práci vzorkována pouze úzká oblast pixelů v kotouči slunce (protože zbytek scény má nulovou emisi). Vzorkuje se pouze *depth buffer*, barva emise se počítá přímo v *shaderu* a je pro všechny vzorky uvnitř slunce stejná.

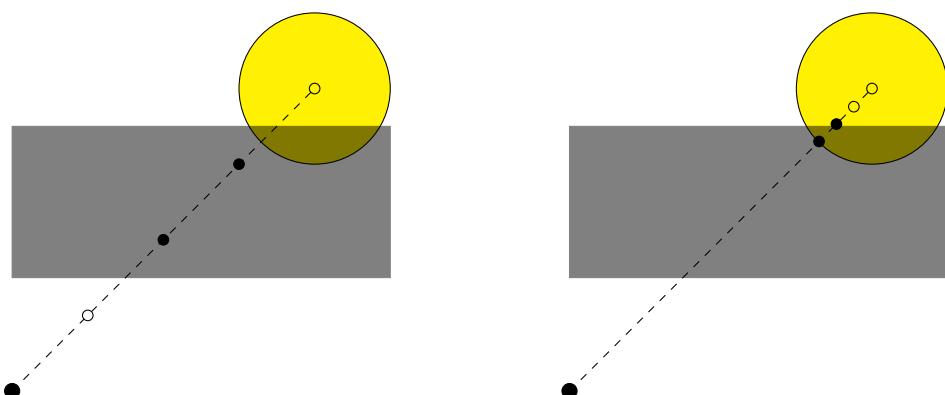


Figure 5.21: Původní navržené vzorkování (vlevo) vs. implementované vzorkování (vpravo) v efektu *god rays*



Figure 5.22: *God rays* v aplikaci

Tato implementace ke slunci částečně přistupuje jako k bodovému zdroji světla (nikoli jako ke kruhovému). Důsledkem toho je, že pokud je zakryta spodní část slunce (od středu dolů), slunce vrhá paprsky pouze nahoru, i když by vrchní polovina slunce měla také vrhat paprsky. Pro odstranění tohoto jevu by bylo třeba vzorkovat celou plochu slunečního kotouče, nikoli pouze přímku mezi výchozím pixelem a středem slunce.

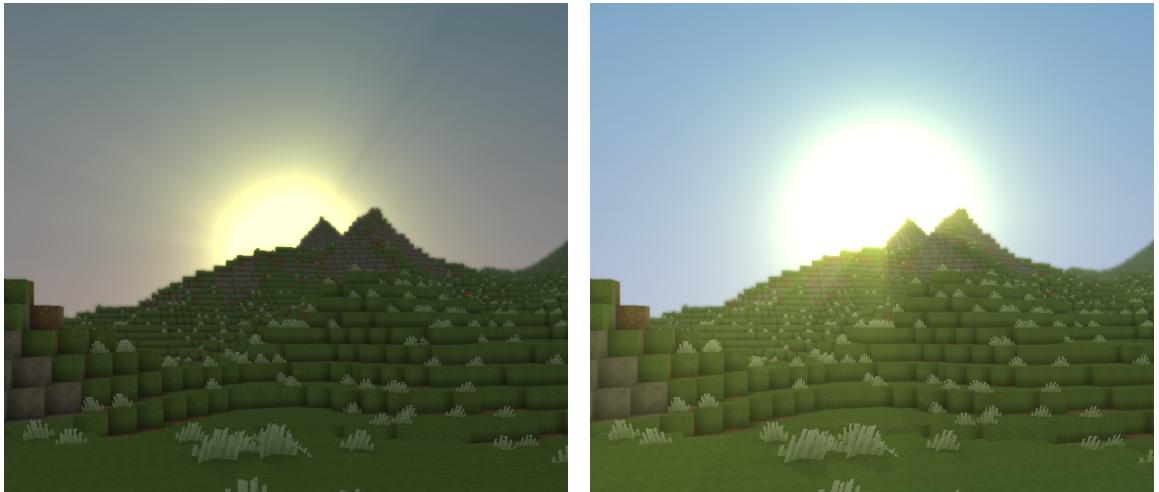


Figure 5.23: Zakrytí slunce od středu k některému okraji zamezuje vrhání paprsků do daného směru ve zvolené implementaci *god rays*

5.5 *Skybox* a střídání dne a noci

Skybox je dynamicky generován v druhém globálním průchodu *postprocessingu*; současně je také jeho část počítána v prvním průchodu pro barvu atmosféry v DOF. Většina výpočtů pracuje s normálou kamery (a vše je tedy založené na úhlech), slunce se ale počítá ve *screen space* kvůli deformacím, které by vznikaly při promítání sférického *skyboxu* na rovinu.

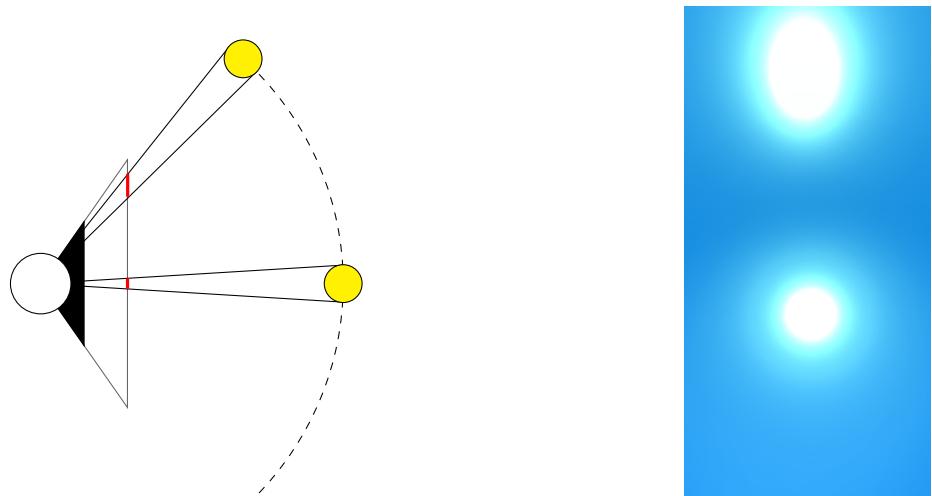


Figure 5.24: Demonstrační deformace slunce při promítání na rovinu

Výpočet pak probíhá pouze pro pixely s průhledností menší než jedna (*skybox* se tedy nepočítá pro pixely zcela překryté popředím). Barva každého pixelu se počítá dle následující rovnice:

$$\begin{aligned}
 k_1 &= \text{clamp}_{\langle 0,1 \rangle}((\vec{n}_z + 0.1) \cdot 5)^2 && (\text{výška nad horizontem}) \\
 d_{sun} &= |\vec{n}_{light} - \vec{n}| \\
 \vec{c} &= \vec{c}_{base} \cdot (0.4 + 0.6 \cdot k_1) && (\text{základní barva}) \\
 &\quad + \vec{c}_{horizon} \cdot (1 - |\vec{n}_z|) && (\text{horizont}) \\
 &\quad + \vec{c}_{halo} \cdot \max_0(1 - d_{sun} \cdot 0.5), && (\text{záře od slunce}) \\
 &\quad + \vec{c}_{sun} \cdot \min(1, s_{size}/d_{sunPx})^{s_{pow}} \cdot k_{12} && (\text{slunce})
 \end{aligned} \tag{5.1}$$

kde \vec{n} je normála kamery, \vec{n}_{light} je normála denního světla, \vec{c}_{XX} jsou barvy jednotlivých složek, s_{size} je parametr udávající velikost slunce, d_{sunPx} je vzdálenost od středu slunce v pixelech a s_{pow} je parametr ovlivňující velikost prstence slunce.

Skybox se tedy aditivně skládá ze čtyř vrstev:

1. Základní složka (která je pro spodní polovinu *skyboxu* vynásobena koeficientem 0.6)
2. Složka horizontu
3. Složka horizontu v okolí slunce
4. Slunce

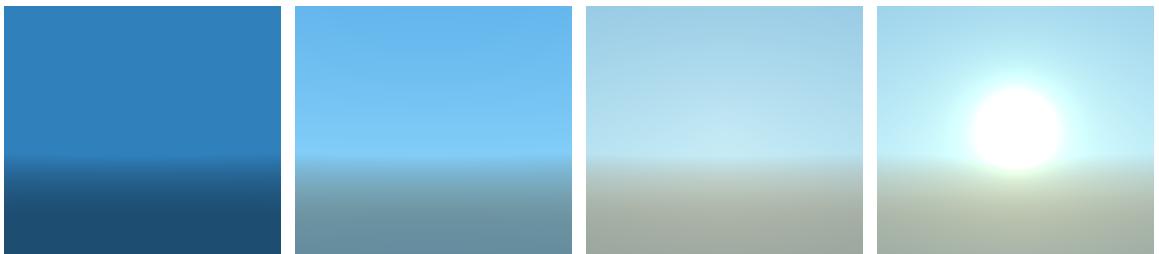


Figure 5.25: Postupná aplikace jednotlivých složek *skyboxu*

Parametry ovlivňující vzhled *skyboxu* jsou *shaderu* předávány přes uniformní paměť. Hodnoty nejsou konstantní, počítají se interpolací z přednastavených hodnot na základě denní doby. Hodnoty parametrů pro *skybox* určuje třída `WorldEnvironment` (implementace pro výchozí svět je v třídě `WorldEnvironment_Overworld`). V noci je měsíc vykreslován stejným způsobem, jakým je ve dne vykreslováno slunce – to zahrnuje i *shadow mapping*, *god rays* a složku denního světla v osvětlovacím modelu. Návrh tohoto *skyboxu* se nezakládá na žádných publikacích.



Figure 5.26: Denní doba v aplikaci: ráno



Figure 5.27: Denní doba v aplikaci: dopoledne

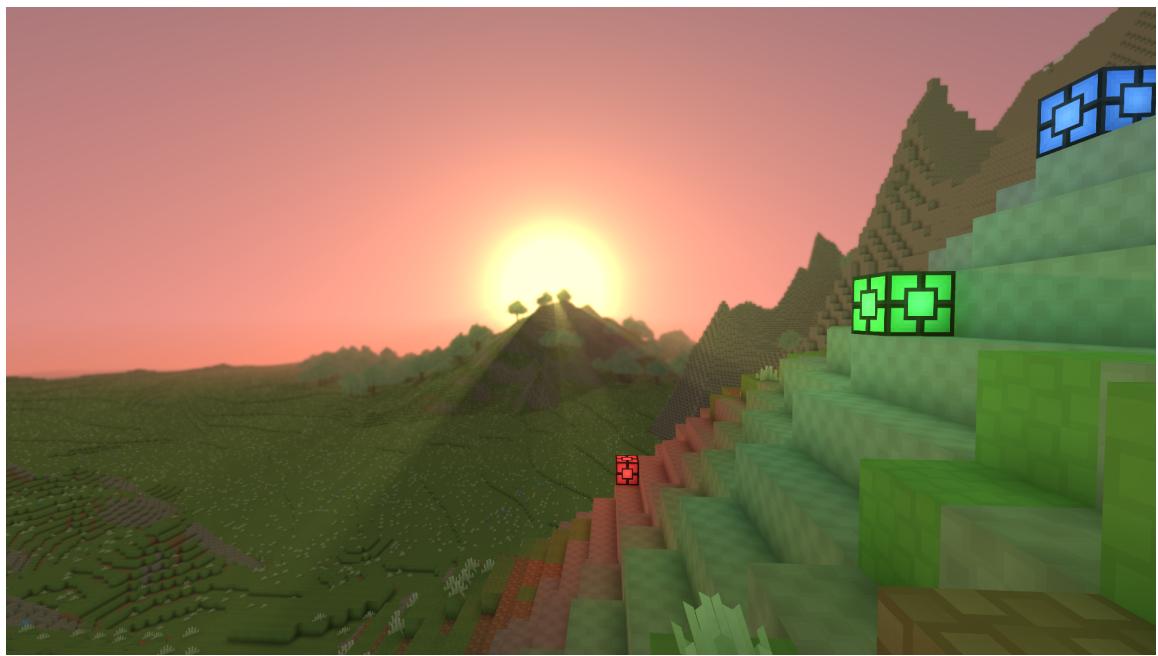


Figure 5.28: Denní doba v aplikaci: večer

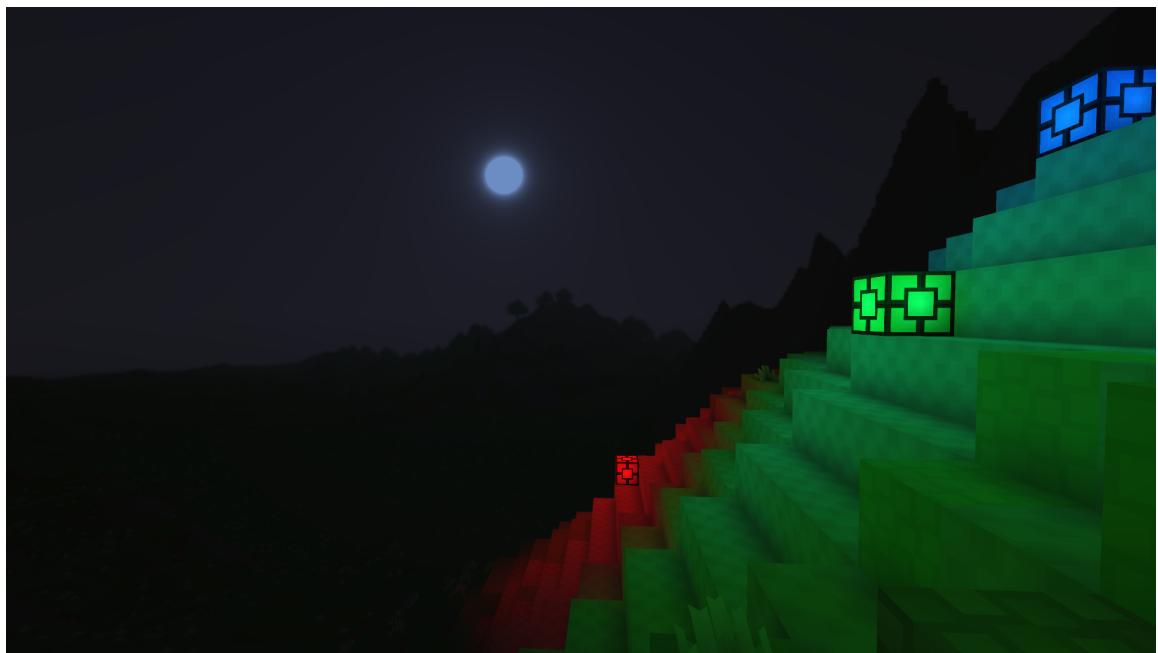


Figure 5.29: Denní doba v aplikaci: noc

Chapter 6

Aplikace

Na přiloženém disku lze nalézt jak předkompilovanou verzi aplikace (pro Windows x64), tak její zdrojové kódy. K sestavení aplikace je třeba mít nainstalován kompilátor D. Ten lze stáhnout z <https://dlang.org/>, na linuxových systémech bývá k dispozici jako balíček `dmd`. Aplikace se sestaví příkazem `dub build --build=release --arch=x86_64`, binární soubor se vytvoří ve složce `bin_x86_64` (resp. jiný prefix pro jinou architekturu). Pro spuštění je vyžadovaná grafická karta s podporou OpenGL 4.6+ a 4 GB grafické paměti. Předkompilované binární soubory lze spustit přímo z CD; v tom případě je ale SQLite databáze se světem vedená v paměti RAM a svět se neukládá na disk.

Aplikaci lze spustit s následujícími parametry:

- `--help`: Vypíše návod
- `--saveGame=name`: Nastavuje název souboru se hrou
- `--recreate`: Při spuštění zajistí nové vygenerování světa se stejným semínkem (zmizí veškeré zásahy hráče)
- `--fullScreen`: Spustí aplikaci v režimu celé obrazovky
- `--debuGL`: Aktivuje ladící režim OpenGL s vypisováním chyb do konzole
- `--collectPerfData`: Aktivuje ukládání metrik do csv souboru.
- `--position=X`: Umístí kamery do pozice X , která je uložená v souboru `positions.txt`. Pozice se do tohoto souboru dají ukládat stiskem klávesy F5.

Soubory uložených her se ukládají do složky `save`.

Vstup	Akce
Prostřední tlačítko myši	Zapnutí/vypnutí ovládání kamery myší
Levé tlačítko myši	Zničení bloku
Pravé tlačítko myši	Postavení bloku
Kolečko myši	Výběr typu bloku (náhled vlevo dole)
W, A, S, D	Pohyb kamery po horizontální rovině
Shift/Ctrl nebo E/Q	Pohyb kamery nahoru/dolů
Mezerník	Skok (pouze v režimu gravitace)
Esc	Ukončení aplikace
F2	Zobrazit/skrýt GUI
F3	Pohled shora (náhled <i>frustum cullingu</i>)
F4	Zapnutí/vypnutí vizualizace načítání chunků
F5	Uložit aktuální pozici kamery do <i>positions.txt</i>
F6	Načíst pozici kamery z <i>positions.txt</i> (cyklicky prochází všechny záznamy)
O/P	Posun denní doby
I	Zapnutí/vypnutí automatického posunu denní doby

Tabulka 6.1: Ovládání aplikace



Figure 6.1: Náhled aplikace

V pravé části obrazovky je jednoduché GUI, kde lze měnit různá nastavení:

1. **Zobrazovaná data;** Pro ladící účely lze místo textur zobrazovat např. normálu, hloubkové informace apod. Jednou z možností je “Aggregation”, která vizualizuje agregaci stěn.
2. **Shading.** Druhým *checkboxem* lze zapnout/vypnout stínování bloků (které je realizované v *postprocessingu*). Protože aplikace podporuje *multisampling*, lze zvolit, zda se má stínování provádět pro každý vzorek zvlášť (**MSAA Shading**), nebo zda se má pracovat pouze s jedním vzorkem na pixel (**Final pixel shading**). V druhém případě se pracuje s nejnižší hodnotou hloubky v pixelu (pro výpočet souřadnic pixelu ve světě). Hodnoty normál se průměrují, což do jisté míry nahrazuje vyhlazování hran. Agregované stínování i tak způsobuje artefakty, když jsou v jednom pixelu dva vzorky, pro které by osvětlení správně mělo mít razantně rozdílné hodnoty.

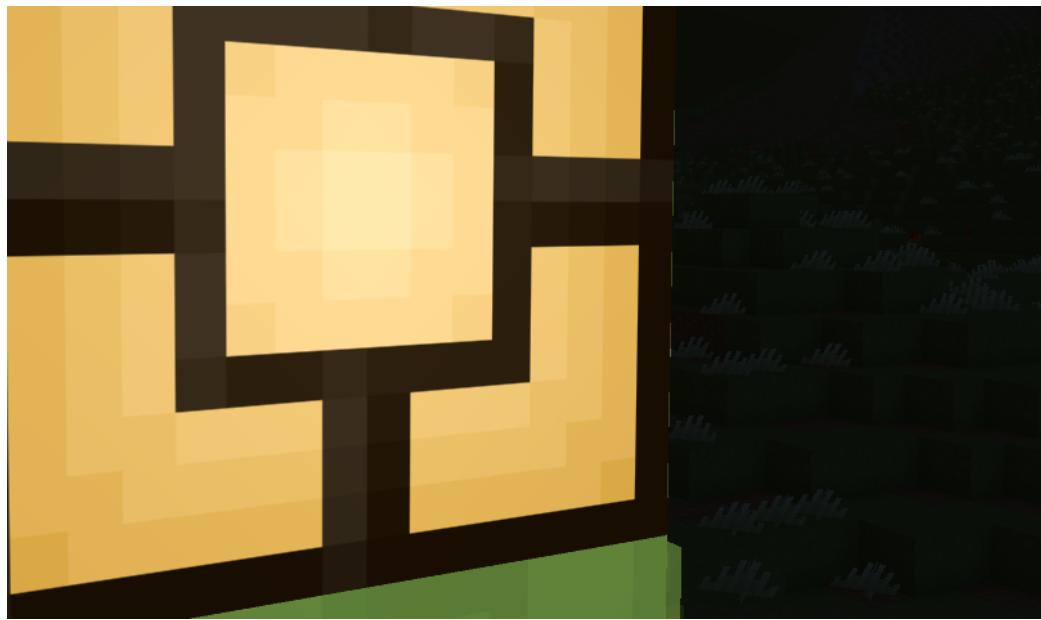


Figure 6.2: Artefakty vznikající na hranách, kde je velký rozdíl osvětlení popředí a pozadí, při *final sample shadingu*

3. **Screen space ambient occlusion.** V tomto menu lze zapnout *SSAO*, případně zobrazit jeho nerozostřený výstup.
4. **Multisampling**
5. **Shadow mapping** a velikost *shadow mapy*.
6. **Depth of field**
7. **Atmosférický efekt** prolnutí s barvou *skyboxu* a zprůhlednění na základě vzdálenosti od kamery
8. **God rays**
9. **Dohledová vzdálenost.** Celkově je zobrazeno $(x + 1 + x)^2$ chunků, kde x je číslo udané v GUI.

10. Zobrazení jednotlivých vrstev *depth peelingu*

11. Počet vrstev *depth peelingu*. V tomto menu lze transparentní vykreslování úplně vypnout nebo změnit počet průhledných vrstev. Vizualizace vypadá plnohodnotně už při jedné vrstvě a protože je tato technika výpočetně náročná, pro slabší počítače se doporučuje počet vrstev omezit.

12. Lepší texturování. Tato volba zapíná vlastní *mag* filtrování u textur (`GL_TEXTURE_MAG_FILTER`). Textury použité v aplikaci nejsou vhodné pro lineární *mag* filtrování, protože pak vypadají rozmařaně; `GL_NEAREST` je vhodnější, nicméně ten způsobuje *aliasing* na přechodech mezi texely. Proto bylo vytvořeno vlastní filtrování (implementované ve *fragment shaderu*), které sice provádí lineární interpolaci, ale pouze na okrajích texelů (míra závisí na *level of detail*). Toto filtrování je o něco dražší, a proto se aplikuje pouze na vybrané *block faces*, na kterých byly artefakty patrné.

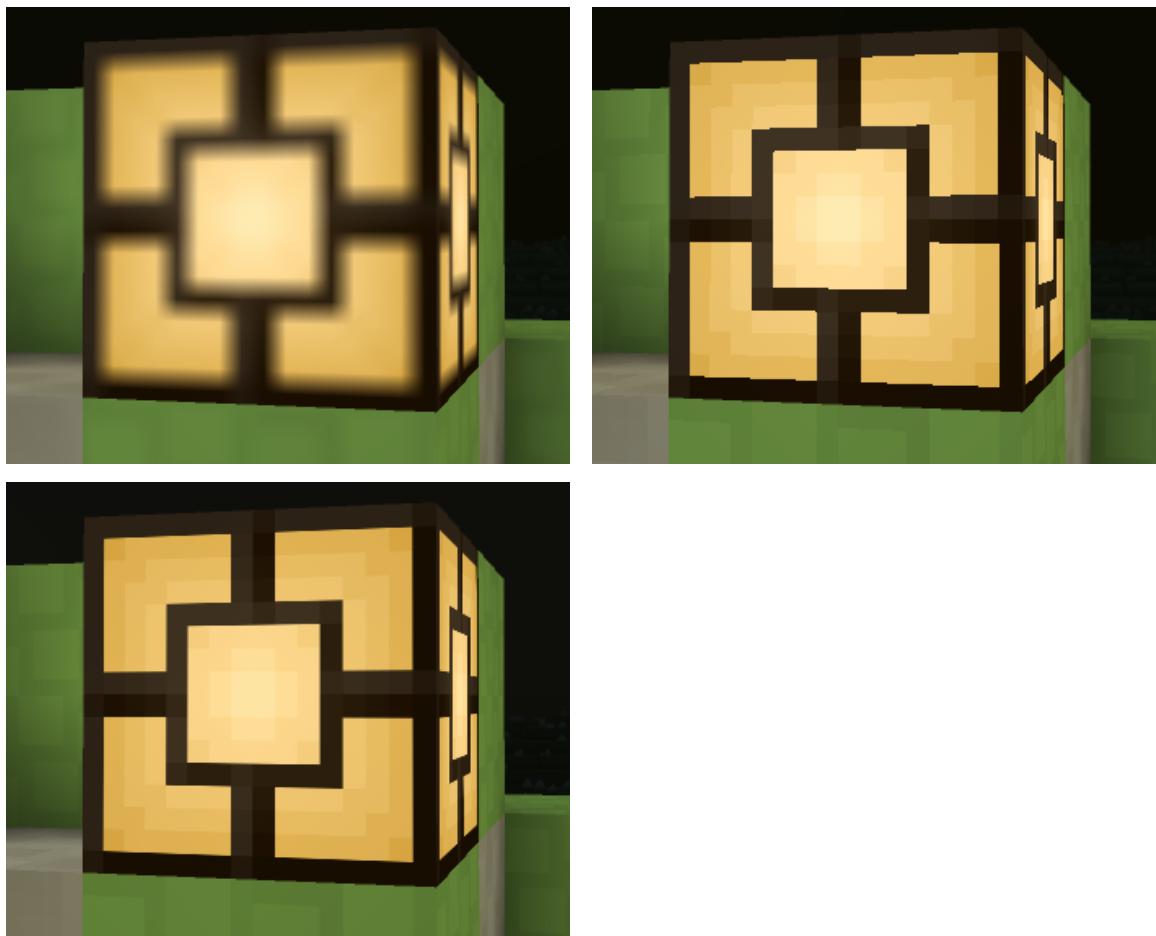


Figure 6.3: Filtrování textur: `GL_LINEAR`, `GL_NEAREST`, vlastní implementace

13. **Multisample alpha testing.** V případě aktivování této možnosti se *alpha testing* provádí pro každý vzorek *multisamplingu*, u vybraných *face contexts* se tedy *fragment shader* invokuje pro každý vzorek (namísto výchozí jedné invokace na pixel).

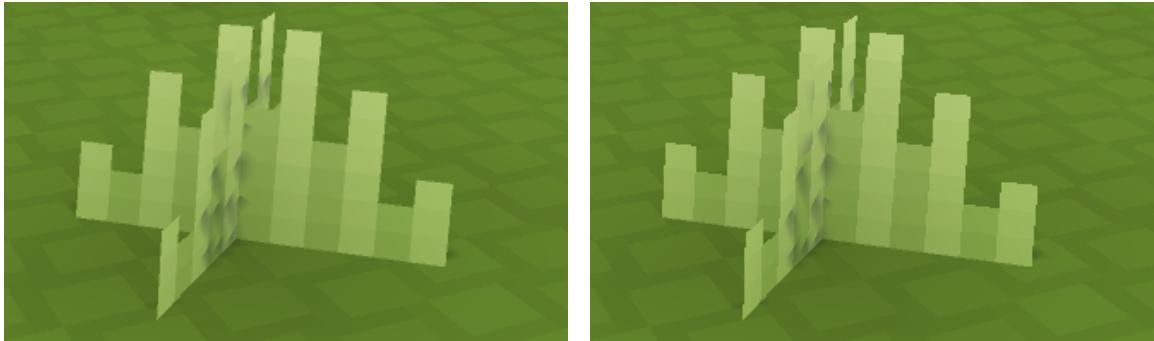


Figure 6.4: Aktivní (vlevo) a neaktivní (vpravo) *multisample alpha testing*

14. **Agregace.** Tato položka umožňuje zvolit metodu agregace primitiv, jak bylo popsáno v sekci 5.3.3.
15. **T-junction hiding** aktivuje efekt v *postprocessingu*, který potlačuje artefakty způsobené *t-junctions*, které jsou při aggregaci vytvářeny, viz sekce 5.3.3.
16. **Animace bloků.** Tato možnost umožňuje vypnout animaci bloků (vlání ve větru, vlnění hladiny).
17. **Módy pohybu.** V tomto menu lze zvolit mód pohybu: rychlý bez kolizí, středně rychlý bez kolizí, pomalý s kolizemi, pomalý s kolizemi a gravitací. Kolize jsou implementovány jednoduchým AABB.
18. **Denní doba.** Posunem jezdce lze měnit denní dobu světa.

6.1 Typy bloků

Náhled	Název	Zvláštní efekty
	Kámen, hlína, tráva, sníh, písek	
	Ruda	Ruda je vidět i bez zdroje světla (textura má vlastní emisi), vydává slabé světlo, generuje se pod zemí
	Kmen stromu	Různé textury na bocích a na vrchu/ves-pod
	Květiny, tráva	Vršek vlaje ve větru, blok ve tvaru X
	Obilí	Vršek vlaje ve větru, blok ve tvaru #
	Houby	Blok ve tvaru X
	Svítící houba	Blok ve tvaru X, modře svítí, generované v jeskyních
	Listí	Celý blok vlaje ve větru
	Kaktus	Blok ve tvaru # (stěny jsou oproti obilí blíže k okrajům)
	Lampy	Vydávají světlo (různé barvy)
	Sklo	Průhledné
	Voda	Průhledné, hladina se vlní

Tabulka 6.2: Přehled typů bloků v aplikaci

Všechny uvedené bloky lze stavět a ničit (pro stavění se vybere blok pomocí kolečka myši a postaví se stiskem pravého tlačítka; ničí se levým tlačítkem). Všechny bloky kromě lamp a skla jsou generovány ve světě. Všechny bloky kromě květin, obilí, trávy a hub kolidují s hráčem a nelze jimi procházet (pakliže jsou zapnuté kolize).

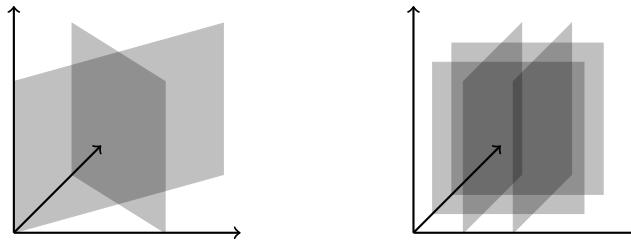


Figure 6.5: Zvláštní tvary bloků: X (vlevo) a # (vpravo)

6.2 Výkon

Tento oddíl se věnuje zhodnocení a optimalizaci výkonu aplikace. Všechna zde uvedená měření se vztahují na referenční prostředí:

- Notebook Acer Aspire VN7-592G
- GPU nVidia GeForce GTX 960M, 4 GB GDDR5
- CPU Intel i7-6700HQ, 4× 2,6 – 3,5 GHz + HyperThreading
- RAM 2× 8 GB DDR4, 2133 MHz
- HDD 1 TB, 5400 RPM
- OS Windows 10 x64
- Aplikace 1920×1080 v režimu celé obrazovky
- Sestavení aplikace pro architekturu `x86_64` v režimu `release` kompilátorem `dmd` (`dub build --build=release --arch=x86_64`)

Pokud není uvedeno jinak, měření probíhá při těchto podmínkách:

- 2× MSAA, *MSAA shading, MSAA alpha testing*
- *Shadow mapping* 2048×2048
- Deaktivované SSAO
- Aktivní DOF, *god rays*, animace bloků, lepší texturování
- 3 vrstvy *depth peeling*
- *Squares ext aggregace, T-junction hiding*
- Dohledová vzdálenost 32 chunků

Měření budou prováděna na níže uvedených scénách. Scéna definuje přesnou pozici a nařízení kamery ve světě a navíc i denní dobu, protože s pohybem slunce se mění i data vykreslovaná do *shadow mapy*. Svět s referenčními scénami je přítomen na přiloženém CD. Z CD nicméně nelze světy načítat, takže je třeba aplikaci zkopirovat na disk a soubor terénu `/referenceScenes.sqlite` přesunout do složky `save` (umístěný na stejném úrovni se složkou `bin`). Poté je možné jednotlivé scény zobrazit příkazem `./ac --saveName=referenceScenes --position=x`, kde `x` je číslo scény míinus jedna.

#	Název	Vlastnosti
1	Oceán	Velké množství průhledných bloků.
2	Poušť	Většinou pouze krychlové bloky, zcela bez průhledných bloků.
3	Krajina	Velké množství vegetace a listů – <i>alpha testing</i> , animace bloků. Listí není neprůhledné, takže se stěny listí vykreslují i uvnitř koruny. Vegetace se nedá agregovat.
4	Jeskyně	Pohled dolů, malé množství viditelných chunků i při velkých vykreslovacích vzdálenostech. Zcela bez průhledných bloků.

Tabulka 6.3: Přehled referenčních scén pro měření výkonu

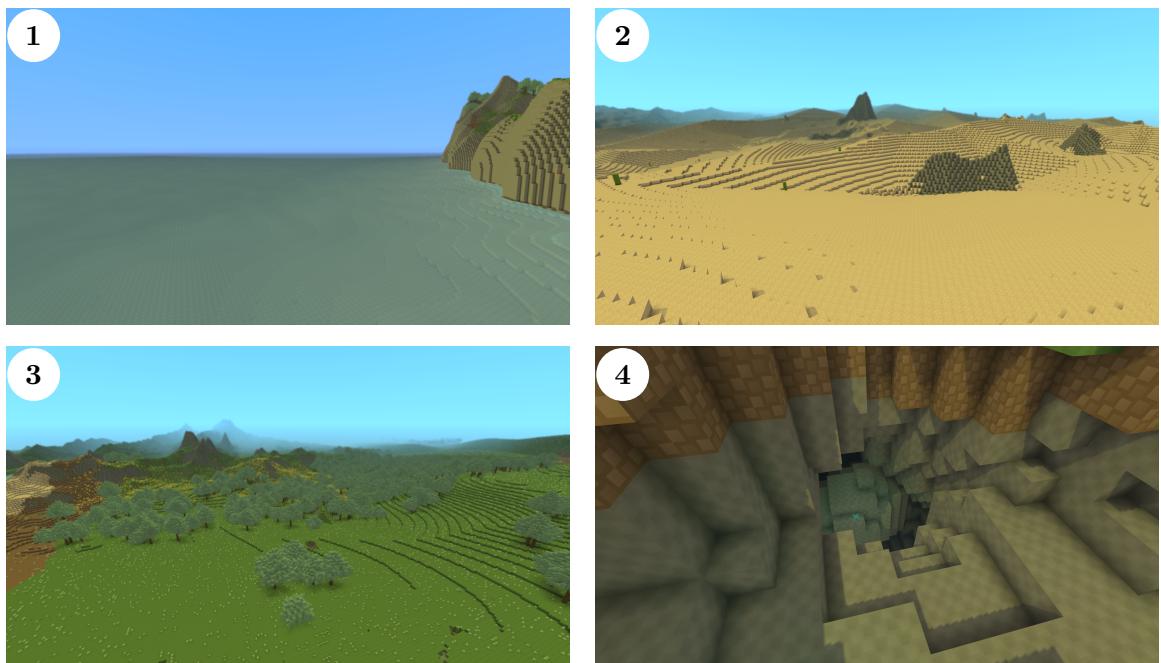


Figure 6.6: Referenční scény pro měření výkonu

Při měření se budou používat tyto metriky:

Metrika	Jednotka	Popis
FPS	–	Počet vykreslených snímků za vteřinu. Měří se modus mezi počtem snímků v každé vteřině. Měření probíhá po načtení všech chunků.
T_{load}	s	Čas mezi spuštěním aplikace a načtením (načtení z disku, sestavení vykreslovacích dat, výpočet osvětlení) všech chunků v dohledové vzdálenosti.
V_{dist}	chunky	Dohledová vzdálenost v chuncích. Dohledová vzdálenost d odpovídá $(2d + 1)^2$ viditelným chunkům.
N_{Δ}	–	Počet vykreslovaných trojúhelníků.
N_{Δ}^+	$O/T/S$	Počet vykreslovaných trojúhelníků. Údaj je rozdelen na trojúhelníky vykreslované jako neprůhledné (O), trojúhelníky vykreslované v jedné vrstvě <i>depth peelingu</i> (T) a trojúhelníky vykreslované do <i>shadow mapy</i> (S). Celkový počet vykreslovaných trojúhelníků je $O + T \cdot d + S$, kde d je počet vrstev <i>depth peelingu</i> .

Tabulka 6.4: Metriky používané při měření výkonu

Následující oddíly se věnují měření vlivu jednotlivých faktorů na výkon aplikace. Hlavní faktory ovlivňující výkon jsou:

1. Počet vykreslovaných trojúhelníků
2. Počet *aktivních* chunků: každý snímek se iteruje přes všechny *aktivní* chunky a volá se funkce `step`
3. Vykreslovací vzdálenost: vykreslovací vzdálenost zvyšuje nároky na výpočet *frustum cullingu*, s rostoucím počtem viditelných *chunk regionů* roste režie CPU při sesťavování seznamu bufferů k vykreslení.
4. *Postprocessing* efekty
5. Procedurální generování chunků: procedurální generování je akcelerované na GPU, takže grafická karta musí dělit výkon mezi generováním a vykreslováním. Generování je také pomalejší než načítání.

6.2.1 Konstanty

Aplikace obsahuje několik pro výkon relevantních konstant:

1. Rozměry chunku
2. Výška *chunk regionu*
3. Velikosti pracovních skupin shaderů
4. Rozměry *light mapy* (třída `VisibleArea`) – 3D textury, ve které jsou na GPU uložena data osvětlení

Velikost chunku nelze jednoduše měnit, protože je na ní založeno mnoho systémů, které na ní zakládají např. velikosti pracovních skupin shaderů, zarovnání souřadnic na bajty apod. Nicméně lze měnit výšku *chunk regionu*: *regiony* dělí vertikálně chunk na několik částí, každý region má vlastní buffery pro vykreslování a pro každý region je zvlášť počítán *frustum culling*. Z implementačních důvodů je minimální výška regionu 8 bloků a maximální 128 bloků.

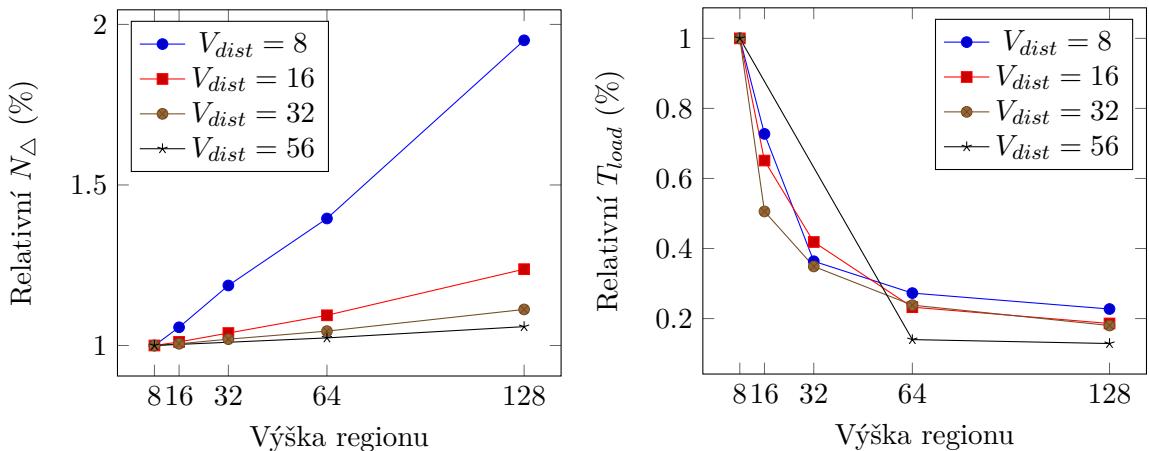
Zvýšení velikosti regionu může znamenat:

1. Zvýšení počtu vykreslovaných trojúhelníků, protože probíhá hrubší *frustum culling*. Rozdíl bude méně patrný při větších dohledových vzdálenostech, kde je většina chunků stejně v pohledovém jehlanu celá.
2. Méně *draw calls*. Větší regiony znamená větší oblasti pokryté jedním *draw call* a uložené v jednom bufferu.
3. Menší režii CPU, protože seznam bufferů pro vykreslení sestavuje CPU.
4. Menší režii GPU. Pro každý region jsou samostatně sestavována vykreslovací data. Sestavování dat je také akcelerováno na GPU a pro každé sestavování je spuštěn dedikovaný *kernel*. Menší počet spuštění *kernelů* může mít vliv zejména při načítání světa, kdy se sestavují data pro velké množství chunků. Řízení výkonu GPU je navíc implementováno omezením počtu procesů na pozadí, které mohou být zadány během jednoho snímku, takže zvětšením regionu se sníží počet procesů nutných k načtení chunku.
5. Pomalejší úpravy terénu. Při změně jednoho voxelu je třeba znova sestavit vykreslovací data pro celý region. Sestavování jednoho regionu je ale příliš rychlé na to, aby tento efekt mohl být znát.

Bylo provedeno měření závislosti snímkové frekvence a doby načítání na výšce regionu v referenčním prostředí 3:

Dohl. vzd.		Výška regionu				
		8	16	32	64	128
$V_{dist} = 8$	FPS	79	79	79	79	79
	N_Δ	139 594	147 518	165 684	194 812	272 270
	T_{load}	11 s	8 s	4 s	3 s	2,5 s
$V_{dist} = 16$	FPS	59	60	55	61	59
	N_Δ	739 286	747 504	767 902	808 836	914 982
	T_{load}	43 s	28 s	18 s	10 s	8 s
$V_{dist} = 32$	FPS	40	44	41	41	44
	N_Δ	1 734 210	1 743 498	1 768 282	1 811 754	1 928 928
	T_{load}	172	87	60	41	31
$V_{dist} = 56$	FPS	26			28	29
	N_Δ	4 163 242			4 261 558	4 407 838
	T_{load}	893			125	115

Tabulka 6.5: Výsledky měření závislosti výkonu na výšce *chunk regionu* ve scéně 3



Graf 6.1: Závislost N_Δ na výšce *chunk regionu* ve scéně 3

Graf 6.2: Závislost T_{load} na výšce *chunk regionu* ve scéně 3

Z měření vyplývá, že je výhodnější mít větší velikost regionu. Vliv velikosti regionu na snímkovou frekvenci je minimální, ale zvýšením velikosti regionu se drasticky snižuje doba načítání. Finální výška regionu byla tedy zvolena 64 bloků, protože při 128 blocích se doba načítání již příliš nezkratí, ale zato drasticky stoupá počet vykreslovaných trojúhelníků při menších dohledových vzdálenostech, což by mohlo ovlivnit výkon na slabších počítačích. Výsledky naznačují, že by mohlo být výhodné mít dodatečné dělení regionů, kde by se vykreslovací data generovala najednou pro celý region, ale *frustum culling* by se granuloval na subregiony.

Další konstantou, která může být předmětem optimalizace, je velikost pracovních skupin shaderů. V *postprocessingu* většina shaderů nevyužívá vláknové kooperace, proto by teoreticky jejich rychlosť měla růst, dokud velikost pracovní skupiny neodpovídá velikosti *warpu*.

(většinou 32/64 vláken) na grafické kartě, a pak by měla být víceméně stejná, případně klesat kvůli obtížnější koordinaci. Shadery s vláknovou kooperací by mohly větší pracovní skupiny využít, takže by stagnace mohla nastat až na vyšších rozměrech.

Scéna	Velikost pracovní skupiny				
	2×2	4×4	8×8	16×16	32×32
1 Oceán	17	42	56	55	53
2 Poušt	20	42	55	55	52
3 Krajina	17	36	45	45	43
4 Jeskyně	18	49	66	67	60

Tabulka 6.6: Závislost snímkové frekvence na velikosti pracovní skupiny shaderů *postprocessingu* (měří se pouze pro shadery bez vláknové kooperace)

Scéna	Velikost prac. sk.		
	8×8	16×16	32×32
1 Oceán	56	56	55
2 Poušt	55	55	54
3 Krajina	45	46	45
4 Jeskyně	66	65	64

Tabulka 6.7: Závislost snímkové frekvence na velikosti pracovní skupiny *blur* shaderů (s vláknovou kooperací)

Měření potvrzují teorii ohledně shaderů bez vláknové kooperace. Aplikace nevykazuje žádné zrychlení ani u větších velikostí pracovní skupiny u shaderů rozostření. Pakliže nějaké zrychlení je, není nijak patrné na celkovém výkonu aplikace. Proto byla zvolena velikost pracovní skupiny u všech shaderů jako 8×8.

Aplikace obsahuje i další shadery, nicméně u těch je velikost pracovní skupiny limitována dalšími okolnostmi:

1. Při výpočtech osvětlení je velikost pracovní skupiny nastavena na maximální možnou hodnotu 8×8×8; menší velikost skupiny by výpočty dělila do menších diskrétních oblastí a tím pádem by muselo být spuštěno více kernelů, aby se světlo mohlo propagovat do maximální vzdálenosti.
2. Sestavování vykreslovacích dat taktéž využívá maximální přípustnou velikost pracovní skupiny, protože velikost pracovní skupiny limituje maximální míru agregace stěn, která je prováděna v rámci vláknové kooperace.
3. Procedurální generování světa taktéž využívá vláknovou kooperaci do velké míry. Zde by mohlo být měření velikosti pracovní skupiny smysluplné, ale je vynecháno z časových důvodů.

Poslední konstantou zkoumanou v tomto oddíle je velikost 3D textur pro ukládání osvětlovacích dat na GPU. Velikost textur může mít vliv na využití VRAM (protože alokovaná paměť se zaokrouhluje na objem textury) a na snímkovou frekvenci, kde se při *shadingu* v *postprocessingu* musí shaderu předávat pole *bindless* textur (menší velikost regionu = více textur k předání) a může docházet k masivnější serializaci při divergenci vláken, kdy vlákna

v jedné pracovní skupině přistupují do více textur. Na GPU jsou v textuře (`ActiveArea`) ukládány také *block IDs*, ty ale nejsou předávány jako *bindless* textury a má je smysl sdružovat jen pro 2×2 chunky, díky čemuž lze oblast 3×3 chunků (což se používá pro výpočty osvětlení) připojit k shaderu pomocí pouze 4 textur. Textur musí být dostatečně malý počet, aby se jejich *handles* vešly do konstantní paměti shaderů.

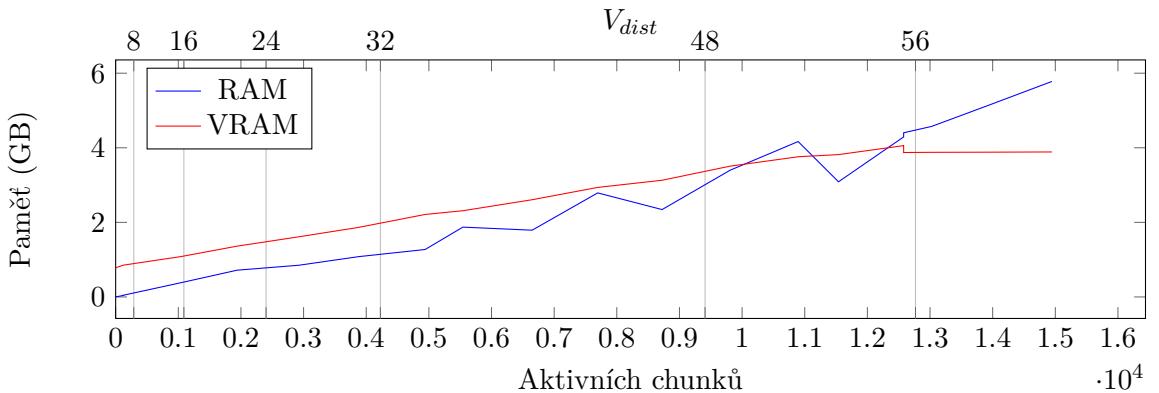
Scéna		Velikost <i>light map</i> textur (v chuncích)				
		2×2	4×4	8×8	16×16	32×32
1 Oceán	VRAM	1 931 MB	1 967 MB	2 043 MB	2 219 MB	2 665 MB
	FPS	51	56	57	57	57
2 Poušt	VRAM	1 931 MB	1 965 MB	2 041 MB	2 217 MB	2 667 MB
	FPS	52	54	55	56	56
3 Krajina	VRAM	1 931 MB	1 967 MB	2 043 MB	2 219 MB	2 665 MB
	FPS	41	45	46	45	45
4 Jeskyně	VRAM	1 931 MB	1 967 MB	2 043 MB	2 217 MB	2 667 MB
	FPS	60	65	65	67	60

Tabulka 6.8: Závislost FPS a využití VRAM paměti na velikosti 3D textury s osvětlovacími daty

Z dat je patrné, že snímková frekvence neroste od velikosti textury 8×8 . Protože ale dále rostou nároky na grafickou paměť, byla zvolena tato velikost (8×8 chunků = $128 \times 128 \times 256$ voxelů).

6.2.2 Spotřeba RAM a VRAM

Nároky na RAM a VRAM by měly přibližně odpovídat 4 B na jeden blok, tedy 0,262 MB na chunk.



Graf 6.3: Závislost spotřeby paměti aplikace na počtu *aktivních* chunků

Spojnice trendu pro naměřené hodnoty RAM je $y = 0,3129x + 24,127$ a VRAM $y = 0,2586x + 853,24$. Spotřeba grafické paměti téměř přesně odpovídá očekávaným hodnotám s tím, že jsou poměrně vysoké počáteční nároky. Protože má referenční stroj k dispozici 4 GB grafické paměti, paměť přesahující tuto hranici se sdílí s procesorem, což už v měření není reflektováno. Operační paměť potom vykazuje mírně zvýšené nároky na režii voxelů, každý voxel tedy ve skutečnosti odpovídá přibližně 4,8 B operační paměti.

6.2.3 Agregace

Byla provedena měření vlivu volby aggregační metody na snímkovou frekvenci a počet vykreslovaných trojúhelníků:

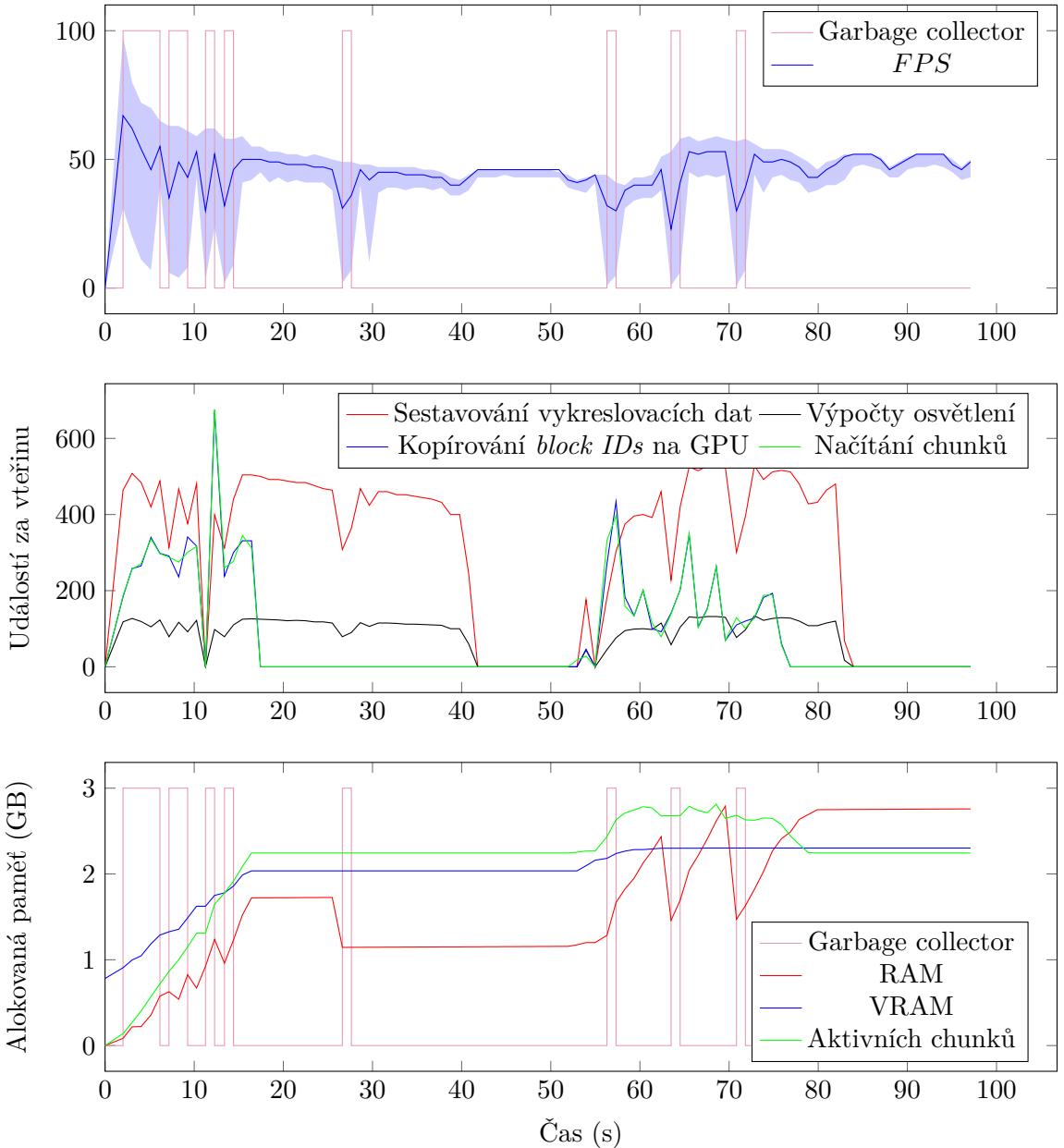
Scéna			Bez agregace	Lines	Squares	Squares ext
1 Oceán	N_{Δ}^O	1611 k (100 %)	639 k (40 %)	553 k (34 %)	480 k (30 %)	
	N_{Δ}^T	560 k (100 %)	71 k (13 %)	10 k (2 %)	9 k (2 %)	
	N_{Δ}^S	73 k (100 %)	32 k (44 %)	30 k (40 %)	26 k (36 %)	
	FPS	35	54	56	56	
2 Poušt	N_{Δ}^O	2371 k (100 %)	1322 k (56 %)	1318 k (56 %)	1140 k (48 %)	
	N_{Δ}^T	0 k (100 %)	0 k (- %)	0 k (- %)	0 k (- %)	
	N_{Δ}^S	55 k (100 %)	28 k (50 %)	27 k (50 %)	23 k (42 %)	
	FPS	44	53	53	56	
3 Krajina	N_{Δ}^O	4811 k (100 %)	2225 k (46 %)	1992 k (41 %)	1743 k (36 %)	
	N_{Δ}^T	10 k (100 %)	2 k (16 %)	0 k (3 %)	0 k (3 %)	
	N_{Δ}^S	156 k (100 %)	81 k (52 %)	78 k (50 %)	68 k (43 %)	
	FPS	29	42	44	46	
4 Jeskyně	N_{Δ}^O	296 k (100 %)	167 k (57 %)	168 k (57 %)	147 k (50 %)	
	N_{Δ}^T	0 k (100 %)	0 k (- %)	0 k (- %)	0 k (- %)	
	N_{Δ}^S	148 k (100 %)	82 k (55 %)	80 k (54 %)	71 k (48 %)	
	FPS	60	60	60	60	

Tabulka 6.9: Závislost snímkové frekvence a N_{Δ}^+ na metodě agregace

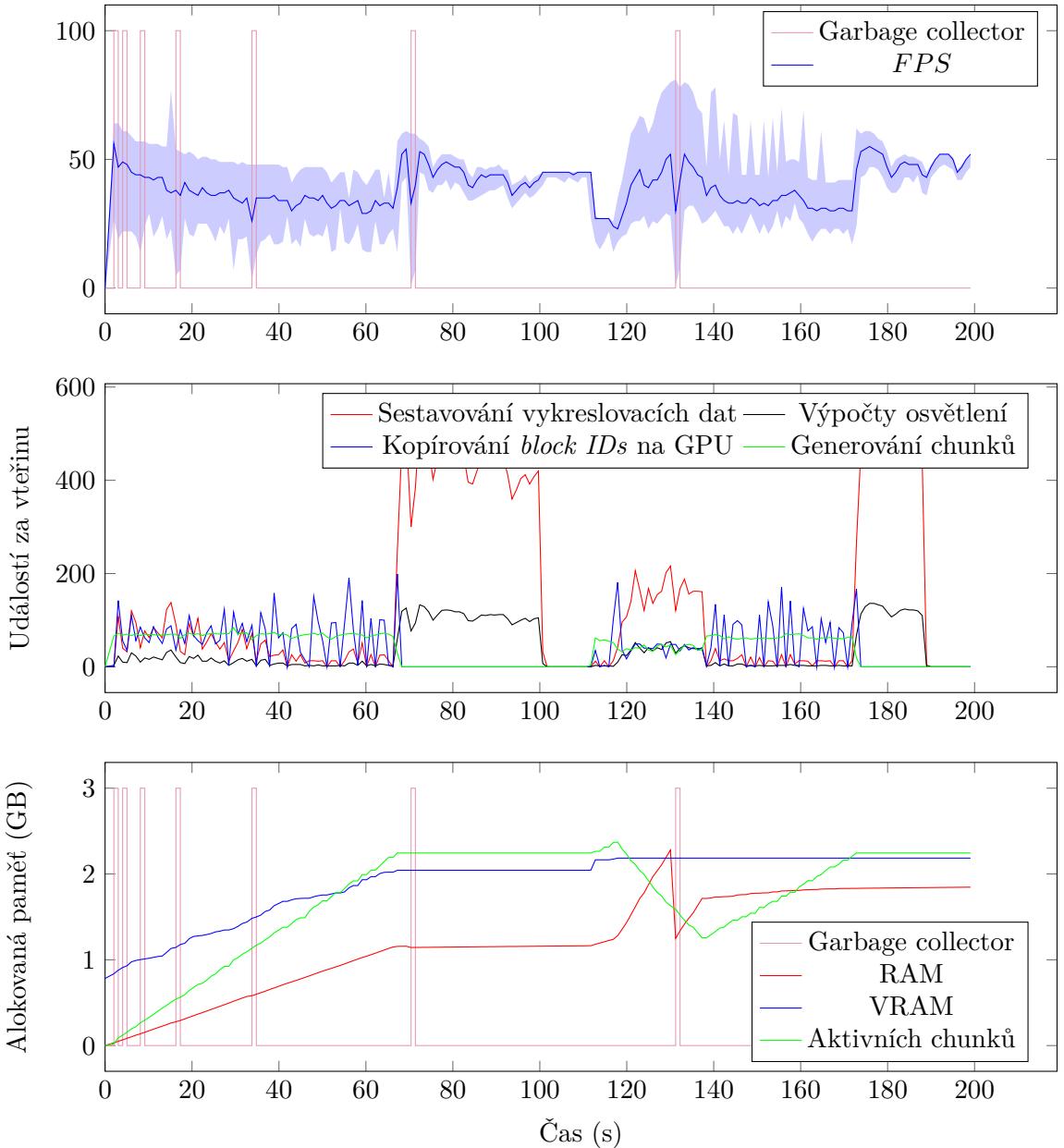
Data ukazují, že agregace je velmi účinnou optimalizační technikou, která dokáže značně zvýšit rychlosť hry. Nejfektivnější z testovaných metod je **Squares ext**, která dokáže omezit počet vykreslovaných trojúhelníků až o 70 % u neprůhledných bloků a až o 98 % u průhledných bloků. Drastické snížení počtu vykreslovaných průhledných bloků ve scéně Oceán je způsobeno tím, že scéna obsahuje převážně jednolitou mořskou hladinu, která jde snadno agregovat až na maximální oblasti 8×8 . Tato agregace je o to úspornější, že průhledné bloky se vykreslují až třikrát kvůli *depth peelingu*.

6.2.4 Načítání a generování chunků, dynamické chování aplikace

Byla provedena měření chování aplikace při statické a pohyblivé kamere. Aplikace byla spuštěna, počkalo se, až se načte terén, poté byl proveden dvacetivteřinový pohyb kamery vpřed, načež se opět počkalo na načtení terénu. Tento test byl proveden dvakrát, jednou na již vygenerovaném terénu, kde se data načítala z disku, podruhé na nevygenerovaném terénu, kde se svět procedurálně generoval na GPU (pomocí přepínače `--recreate`). Pro objem dat nutných k vizualizaci byl test omezen pouze na scénu 3 a dohledovou vzdálenost 32.



Graf 6.4: Dynamické chování aplikace ve scéně 3: nejprve se čeká na načtení terénu, poté se kamera pohybuje 20 s vpřed (`fast noclip`), poté se opět čeká na načtení. Terén je již předvygenerovaný a uložený v SQLite souboru.



Graf 6.5: Dynamické chování aplikace ve scéně 3: nejprve se čeká na vygenerování terénu, poté se kamera pohybuje 20s vpřed (**Fast noclip**), poté se opět čeká na vygenerování. Terén není přednačtený a přímo se generuje na GPU.

Z měření lze odvodit následující:

1. *Garbage collector* (GC) v jazyce D způsobuje znatelné záseky aplikace (až okolo 600 ms). Ještě před tímto měřením byla provedena snaha o minimalizaci využití GC, další omezování by však vyžadovalo nahrazení funkčnosti standardní knihovny (kontejnery, zlib, ...) vlastní implementací, což by bylo velice pracné.

Krátce před odevzdáním této práce byly pole s block IDs a block small data přesunuty mimo paměť spravovanou garbage collectorem (nově jsou alokovány přes malloc), což

omezilo doby běhu *garbage collectoru* ze stovek milisekund na jednotky. Pro nedostatek času už nebyla provedena opravná měření.

2. Při načítání chunků z disku si aplikace drží stabilní snímkovou frekvenci bez výraznějšího kolísání (až na běhy GC). Systém distribuce práce na pozadí, který je v aplikaci implementován, funguje.
3. Procedurální generování chunků je výrazně pomalejší (cca $2,5\times$) než načítání chunků z disku. Během procedurálního generování má aplikace nižší snímkovou frekvenci, která je navíc nestabilní. Tento jev se dá vysvětlit tím, že procedurální generování probíhá na vlákně a v odděleném OpenGL kontextu, kde dynamické omezování výkonu není implementováno. Jedinou formou omezení výkonu generování světa je fixní prodluha 1 ms mezi generováním jednotlivých chunků. V rámci dalších optimalizací by bylo vhodné se zaměřit právě na subsystém procedurálního generování.
4. Načítání světa z paměti je (pro $V_{dist} = 32$) dostatečně rychlé na to, aby aplikace stíhala načítat svět s pohybem kamery (měřeno při `fast noclip`, což je nejvyšší nastavitelná rychlosť v aplikaci). Toto lze vidět ve spodním grafu, kde počet aktivních chunků nejprve přibude, než se začnou odkládat chunky mimo zorné pole (kterým vyprší časovač aktivity, protože nebyl obnovován). Rychlosť procedurálního generování chunků není pro tuto situaci dostatečná.

6.2.5 Postprocessing

Na závěr provedeme měření vlivu jednotlivých *postprocessing* efektů na výkon hry. Prvním zkoumaným efektem bude osvětlení. Ačkoli je osvětlení počítáno ve *screen space*, lze očekávat jisté zpomalení s přibývající dohledovou vzdáleností, protože neprůhledných pixelů bude pravděpodobně více a souřadnice pixelů ve světě budou dál od sebe, takže bude snížena lokalita přístupu do paměti. Úzké hrdlo efektu bude pravděpodobně přístup k texturám, takže není očekáván větší rozdíl ve výkonu mezi *per sample* a *per pixel shadingem*. Pro větší zřetelnost případného rozdílu budeme ale provádět měření při $4\times$ MSAA oproti výchozím $2\times$.

Scéna	<i>Shading</i>	V_{dist}			
		4	8	16	32
1 Oceán	<i>off</i>	82	77	71	57
	<i>per pixel</i>	77	71	65	52
	<i>per sample</i>	60	53	49	41
2 Poušt	<i>off</i>	88	83	73	51
	<i>per pixel</i>	82	77	68	52
	<i>per sample</i>	66	60	55	44
3 Krajina	<i>off</i>	81	73	55	42
	<i>per pixel</i>	79	68	52	40
	<i>per sample</i>	69	57	43	34
4 Jeskyně	<i>off</i>	72	68	68	62
	<i>per pixel</i>	60	59	58	55
	<i>per sample</i>	45	44	44	42

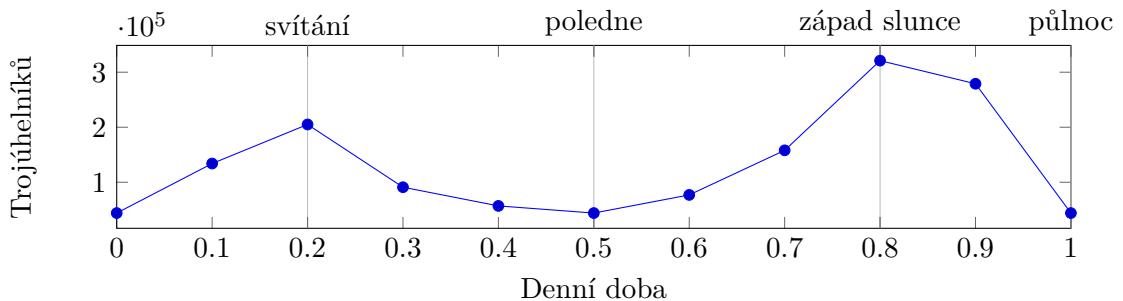
Tabulka 6.10: Závislost snímkové frekvence na dohledové vzdálenosti a metodě *shadingu* při $4\times$ MSAA

Měření zcela neodpovídají očekáváním, značný pokles snímkové frekvence je i při změně z *per pixel shading* na *per sample shading*. Na výkon má tedy znatelný vliv i samotné zpracování dát.

Dalším efektem je *shadow mapping*. Ten by zdánlivě měl být málo závislý na dohledové vzdálenosti, protože se vždy vykresluje oblast před kamerou s fixním poloměrem. Počet vykreslovaných trojúhelníků se nicméně mění v závislosti na denní době: když je slunce nejvýše na obloze, takže jsou paprsky vrhány kolmo dolů na oblast kolem kamery, když je slunce na horizontu, musí být vykresleny všechny chunky ve směru ke slunci. V noci *shadow mapy* využívá měsíc, který putuje po stejném dráze, jako slunce, pouze rychleji a v opačném směru.

$V_{dist}:$	4	8	16	32
N_{Δ}^S v poledne:	39 k	50 k	50 k	50 k
N_{Δ}^S při svítání:	35 k	70 k	221 k	262 k

Tabulka 6.11: Závislost počtu trojúhelníků vykreslovaných do *shadow mapy* na dohledové vzdálenosti ve scéně 3



Graf 6.6: Závislost počtu trojúhelníků vykreslovaných do *shadow mapy* na denní době ve scéně 3

Výkon *shadow mappingu* je pak ovlivněn velikostí *shadow mapy* a rozlišením obrazovky, resp. *multisamplingem*.

<i>MSAA</i>	<i>Shadow mapping</i>			
	<i>off</i>	1024^2	2048^2	4096^2
$1\times$	54	49	39	24
$2\times$	47	41	35	22
$4\times$	35	32	28	20
$8\times$	23	21	19	15

Tabulka 6.12: Závislost snímkové frekvence na *MSAA* a rozměrech *shadow mapy* ve scéně 3 při svítání

Ve výsledcích není nic neočekávaného. Měřením v tabulce 6.12 jsme současně zhodnotili i náročnost *multisamplingu*, která znatelně stoupá s počtem vzorků.

Posledním efektem, který budeme detailněji měřit, je *depth peeling*.

Scéna		Vrstev <i>depth peelingu</i>			
		0	1	2	3
1 Oceán	FPS	81	64	61	57
	N_{Δ}	506 k	515 k	525 k	534 k
2 Poušt	FPS	68	61	59	56
	N_{Δ}	1 163 k	1 163 k	1 163 k	1 163 k
3 Krajina	FPS	54	50	47	45
	N_{Δ}	1 811 k	1 811 k	1 811 k	1 812 k
4 Jeskyně	FPS	100	86	80	75
	N_{Δ}	102 k	102 k	102 k	102 k

Tabulka 6.13: Závislost snímkové frekvence a N_{Δ} na počtu vrstev *depth peelingu*

Z výsledků je patrné, že *depth peeling* zpomaluje aplikaci, i když se nevykreslují žádné průhledné bloky. Snaha o redukci tohoto jevu však už není součástí této práce.

Na závěr uvedeme už jen stručné měření vlivu ostatních *postprocessing* efektů oproti výchozímu nastavení:

	Oceán	Poušt	Krajina	Jeskyně
Výchozí nastavení	57	56	41	59
<i>Depth of field off</i>	66	65	46	71
<i>God rays off</i>	59	56	42	59
<i>Lepší texturování off</i>	57	56	41	59
<i>MSAA alpha test off</i>	57	56	41	59
<i>T-junction hiding off</i>	57	56	41	59
<i>Animace bloků off</i>	57	56	41	59

Tabulka 6.14: Vliv jednotlivých *postprocessing* efektů na snímkovou frekvenci ($V_{dist} = 32$).

Chapter 7

Závěr

V rámci této práce byly prozkoumány metody procedurálního generování, reprezentace a vykreslování volumetrického terénu. Byla vytvořena demonstrační aplikace implementující vybrané techniky. Aplikace provádí uměleckou vizualizaci nekonečného procedurálně generovaného volumetrického terénu, umožňuje jeho editaci a uchovávání na disku. Ze zajímavých implementačních prvků aplikace lze zmínit akceleraci procedurálního generování, *frustum cullingu*, přípravy vykreslovacích dat a výpočtu osvětlení na GPU, systém agregace stěn voxelů do jednoho primitiva (takéž akcelEROVANÝ na GPU) nebo osvětlovací model, který má konstantní složitost v závislosti na počtu světel a jehož přirozeným důsledkem je *ambient occlusion* ve vnitřních rozích (ten vychází z návrhu ve hře Minecraft, takéž akcelEROVANÝ na GPU). Za pozitivní lze také považovat celkový vizuální dojem.

Aplikace je vhodná pro rozšíření na plnohodnotnou hru. V rámci dalších experimentů by mohlo být přínosné implementovat *ray casting* (pro ty bloky, které mají tvar krychle); jelikož se data o blocích již ukládají na GPU, triviální implementace by nebyla příliš obtížná. Dále je relevantní průzkum alternativních způsobů šíření světla, protože aktuální implementace umožňuje šíření světla “za roh” a produkuje kosočtvercový vzor zřetelný při pohledu shora. Existuje prostor pro hledání optimálnější metody agregace stěn a optimalizaci generování terénu. A v neposlední řadě by pro uplatnění aplikace jako hry bylo žádoucí vyřešit občasné několikasetmilisekundové záseky způsobené během *garbage collectoru*.

Byla provedena měření dokumentující využité prostředky a výkon aplikace. V přiloženém CD je kromě zdrojových kódů k dispozici i prezentační video aplikace, které bylo zhotovenou v souladu se zadáním.

Bibliography

- [1] A survey of algorithms for volume visualization. *ACM SIGGRAPH Computer Graphics*, ročník 26, č. 3, 1992: s. 194–201, ISSN 0097-8930.
- [2] *GPU gems 3*. Upper Saddle River: Addison-Wesley, 2008, ISBN 978-0-321-51526-1.
- [3] NVIDIA RTX Technology Realizes Dream of Real-Time Cinematic Rendering. *NASDAQ OMX's News Release Distribution Channel*, 2018.
URL <http://search.proquest.com/docview/2015016661/>
- [4] Bavoil, L.; Myers, K.: Order independent transparency with dual depth peeling. Technická zpráva, 2008.
- [5] Boissonnat, J.-D.; Nielsen, F.; Nock, R.: Bregman Voronoi Diagrams. *Discrete & Computational Geometry*, ročník 44, č. 2, Sep 2010: s. 281–307, ISSN 1432-0444, doi:10.1007/s00454-010-9256-1.
URL <https://doi.org/10.1007/s00454-010-9256-1>
- [6] Cirne, M.; Pedrini, H.: Marching cubes technique for volumetric visualization accelerated with graphics processing units. *Journal of the Brazilian Computer Society*, ročník 19, č. 3, 2013: s. 223–233, ISSN 0104-6500.
- [7] Crassin, C.; Neyret, F.; Sainz, M.; aj.: Interactive Indirect Illumination Using Voxel Cone Tracing. *Computer Graphics Forum*, ročník 30, č. 7, 2011: s. 1921–1930, ISSN 0167-7055.
- [8] Enderton, E.; Sintorn, E.; Shirley, P.; aj.: Stochastic Transparency. *Ieee Transactions On Visualization And Computer Graphics*, ročník 17, č. 8, 2011: s. 1036–1047, ISSN 1077-2626.
- [9] Everitt, C.: Interactive Order-Independent Transparency. 2001.
- [10] Fernando, R.: *GPU gems*. Boston: Addison-Wesley, vyd. 1. vydání, 2004, ISBN 0-321-22832-4.
- [11] Giesen, F.: Texture tiling and swizzling. 2011.
URL
<https://fgiesen.wordpress.com/2011/01/17/texture-tiling-and-swizzling/>
- [12] Glatzel, B.: Volumetric Lighting for Many Lights in Lords of the Fallen. *Digital Dragons conference*, ročník 2014.
- [13] Gustavson, S.: Simplex noise demystified. 2005.

- [14] Žára Jiří: *Moderní počítačová grafika*. Brno: Computer Press, vyd 1. vydání, 2004, ISBN 80-251-0454-0.
- [15] Martin, T. L.: Higher order light propagation volumes. 2012.
URL <http://www.escholarship.org/uc/item/3d36v53h>
- [16] McGuire, M.; Bavoil, L.: Weighted Blended Order-Independent Transparency. *Journal of Computer Graphics Techniques (JCGT)*, ročník 2, č. 2, December 2013: s. 122–141, ISSN 2331-7418.
URL <http://jcgt.org/published/0002/02/09/>
- [17] Mittring, M.: Finding Next Gen: CryEngine 2. In *ACM SIGGRAPH 2007 Courses*, SIGGRAPH '07, New York, NY, USA: ACM, 2007, ISBN 978-1-4503-1823-5, s. 97–121, doi:10.1145/1281500.1281671.
URL <http://doi.acm.org/10.1145/1281500.1281671>
- [18] Ostebee, A.: The Algorithmic Beauty of Plants. *The American Mathematical Monthly*, ročník 104, č. 1, 1997, ISSN 00029890.
URL <http://search.proquest.com/docview/203721669/>
- [19] Perlin, K.: An Image Synthesizer. *SIGGRAPH Comput. Graph.*, ročník 19, č. 3, Červenec 1985: s. 287–296, ISSN 0097-8930, doi:10.1145/325165.325247.
URL <http://doi.acm.org/10.1145/325165.325247>
- [20] Perlin, K.: Improving Noise. *ACM Transactions on Graphics (TOG)*, ročník 21, č. 3, 2002: s. 681–682, ISSN 1557-7368.
- [21] Perlin, K.: Noise Hardware. In *SIGGRAPH 2002 Course 36 Notes*, 2, 2002.
- [22] Phong, B.: Illumination for computer generated pictures. *Communications of the ACM*, ročník 18, č. 6, 1975: s. 311–317, ISSN 1557-7317.
- [23] Reeves, W. T.; Salesin, D. H.; Cook, R. L.: Rendering antialiased shadows with depth maps. *ACM SIGGRAPH Computer Graphics*, ročník 21, č. 4, 1987: s. 283–291, ISSN 00978930.
- [24] Sojma, Z.: L-systémy a jejich aplikace v počítačové grafice. 2009.
- [25] Tinsley, J.; Molodtsov, M.; Prevedel, R.; aj.: Direct detection of a single photon by humans. *Nature Communications*, ročník 7, 2016, ISSN 2041-1723.
URL <http://search.proquest.com/docview/1805466271/>
- [26] Williams, L.: Casting curved shadows on curved surfaces. *ACM SIGGRAPH Computer Graphics*, ročník 12, č. 3, 1978: s. 270–274, ISSN 0097-8930.
- [27] Worley, S.: A cellular texture basis function. In *Proceedings of the 23rd annual conference on computer graphics and interactive techniques*, Acm.org, 1996, ISBN 0-89791-746-4, str. 291–294.
- [28] Yang, J. C.; Hensley, J.; Grün, H.; aj.: Real-Time Concurrent Linked List Construction on the GPU. *Computer Graphics Forum*, ročník 29, č. 4, 2010: s. 1297–1304, ISSN 0167-7055.

Appendix A

Obsah přiloženého CD

Soubor/složka	Popis
ac/	Složka se zdrojovými soubory aplikace
bin_X/	Předkompilované binární soubory aplikace pro platformu X
res/	Dodatečné zdroje využívané aplikací: fonty, textury a zdrojové soubory <i>shaderů</i>
lib_X/	Předkompilované knihovny pro platformu X
README.md	Soubor README s instrukcemi k sestavení aplikace
video.mp4	Prezentační video aplikace
referenceScenes.sqlite	Soubor světa s referenčními scénami použitými při měření výkonu (třeba zkopirovat na disk do složky save , která bude na stejně úrovni se složkou bin)
diplomka.pdf	Dokument diplomové práce ve formátu pdf
master_thesis_CZ/	Složka se zdrojovými soubory textu diplomové práce

A.1 Významné zdrojové soubory

Cesta	Popis
<code>res/shader/frustumCulling.cs.glsl</code>	Shader pro výpočet <i>frustum cullingu</i>
<code>res/shader/lighting/lightPropagation.cs.glsl</code>	Shader pro propagaci světla
<code>res/shader/postprocessing/shading.cs.glsl</code>	Shader pro <i>shading</i> a <i>shadow mapping</i>
<code>ac/client/world/chunkrenderer.d</code>	Třída spravující výpočet osvětlení chunku
<code>ac/common/world/world.d</code>	Třída reprezentující svět
<code>ac/common/world/chunk.d</code>	Třída reprezentující chunk
<code>ac/common/block/block.d</code>	Třída reprezentující typ voxelu
<code>ac/content/block/*</code>	Zdrojové kódy definující jednotlivé typy voxelů
<code>ac/content/worldgen/overworld.d</code>	Třída definující výchozí generátor světa
<code>res/shader/worldgen/*</code>	Shadery pro generování světa (GPU implementace Perlinova šumu, Voroného diagramů)
<code>ac/content/world/env/overworld.d</code>	Třída definující výchozí <i>skybox</i>
<code>res/shader/postprocessing/sky.cs.glsl</code>	Shader pro výpočet <i>skyboxu</i>
<code>res/shader/render/blockRender.*.glsl</code>	Shadery pro vykreslování bloků
<code>res/shader/render/blockRenderList.cs.glsl</code>	Shader akcelerující sestavování vykreslovacích dat a agregující stěny
<code>res/shader/render/aggregation_*.cs.glsl</code>	Shadery pro různé metody agregace
<code>ac/client/game/gamerenderer.d</code>	Třída spravující vykreslování světa
<code>ac/client/block/blockface.d</code>	Třída reprezentující <i>block face</i> a jeho konfiguraci