VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

GENEROVÁNÍ A ZOBRAZOVÁNÍ ROZSÁHLÝCH VOXELOVÝCH SCÉN

GENERATING AND RENDERING OF LARGE VOXEL-BASED SCENES

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. DANIEL ČEJCHAN**
AUTHOR

VEDOUCÍ PRÁCE **Ing. MICHAL MATÝŠEK**
SUPERVISOR

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačové grafiky a multimédií (UPGM) Akademický rok 2018/2019

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

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.

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.

Klíčová slova

voxely, volumetrický terén, procedurální generování, OpenGL

Keywords

voxels, volumetric terrain, procedural generation, OpenGL

Citace

ČEJCHAN, Daniel. *Generování a zobrazování rozsáhlých voxelových scén.* Brno, 2019. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Michal Matýšek

Generování a zobrazování rozsáhlých voxelových scén

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Matýška. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Daniel Čejchan
20. května 2019

Poděkování

Děkuji panu Matýškovi, který mne přijal pod svá křídla a neváhal mi obětovat svůj čas. Děkuji panu Miletovi, který, ač mi jistě bude oponovat, byl vždy ochotný podělit se o své znalosti. Děkuji panu Chlubnovi, který mi poskytl své nápady. Děkuji Jakubu Šolínovi, který mi sloužil místo gumové kačenky.

Obsah

Kapitola 1

Úvod

V roce 2009 byla společností Mojang vyvinuta hra Minecraft[1], která si v průběhu několika let získala širokou hráčskou základnu napříč mnoha věkovými kategoriemi a stala se fenoménem herního průmyslu. Vyznačuje se procedurálně generovaným nekonečným volumetrickým terénem, do kterého jsou hráči ovládající své herní postavy vpuštěni bez nějakého předem zadaného cíle. Hra nabízí zábavu více druhům hráčů: kreativní lidé mohou z jednotlivých voxelů stavět modely dle jejich fantazie, technicky zaměření jedinci mohou vymýšlet různé mechanismy (mechaniky hry umožňují vytvoření i např. jednoduchého CPU), hráči toužící po dobrodružství se mohou ponořit do *survival* módu, kde musí těžit suroviny, stavět si pevnosti a bojovat s monstry o přežití.

Mechanikami uvedenými ve hře Minecraft byla inspirována řada dalších titulů; dalo by se říct, že hra dala zrod celému novému hernímu žánru, pro který jsou typické otevřený procedurálně generovaný svět a *survival* a *crafting* mechaniky. Hra však není prvotním zdrojem těchto mechanik – inspirace pro ně přišla od hry Infiniminer.

Obrázek 1.1: Hra Minecraft

Cílem této práce je prozkoumat techniky použitelné pro vizualizaci volumetrického terénu ve hrách typu Minecraft, vybrané techniky implementovat v demonstrační aplikaci

[1] www.minecraft.net

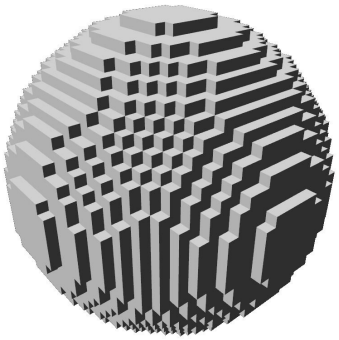
a pokusit se je obohatit o akceleraci na GPU. Návrh a implementace aplikace jsou vedeny tak, aby mohly být v budoucnu rozšířeny na plnohodnotnou hru.

V kapitolách 2 a 3 je proveden průzkum existujících metod pro procedurální generování, reprezentaci a vykreslování grafických modelů a dalších metod relevantních pro tuto práci. Kapitoly 4 a 5 poté dokumentují návrh a implementaci aplikace a kapitola 6 se věnuje zhodnocení výsledného programu.

Kapitola 2

Volumetrický terén

Pojem objemová (volumetrická) reprezentace terénu pouze říká, že veškerá data o terénu jsou vždy vztažena na tělesa o nějakém objemu situovaná v prostoru, který terén definuje. Tvar těles může být obecně libovolný a různý v rámci jedné reprezentace. Tato práce pracuje s krychlemi konstantní velikosti uspořádanými do pravidelné mřížky. Pojem *voxel*, který pro tyto krychle budeme používat, obecně označuje nejmenší jednotku objemu v trojrozměrném diskrétním prostoru [14]. Data, která jsou k voxelům vztažena, mohou být v nejjednodušším případě pouze informace, zda daný voxel reprezentuje hmotu, nebo prázdný prostor. Mohou se ale uchovávat i detailnější informace jako např. barva, materiál nebo jiné optické či fyzické vlastnosti.



Obrázek 2.1: Voxelová reprezentace koule

Práci s terénem lze koncepčně rozdělit na tři části: jeho generování, správu a vykreslování. Správou terénu se rozumí jeho uchovávání v paměti a na disku, v případě nestatického terénu i řízení jeho úprav. Pro tuto kapitolu uvažujme terén jako funkci

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

kde x, y a z jsou souřadnice v kartézském trojrozměrném prostoru a hodnota funkce reprezentuje, zda se na dané souřadnici nachází nebo nenachází hmota. Osy x a y umístíme do horizontální roviny a osa z reprezentuje výškovou souřadnici.

2.1 Procedurální generování volumetrického terénu

Procedurální generování pro účely této práce znamená nalezení takové funkce *ter* (viz rovnice 2.1), pro kterou je možná efektivní implementace pro soudobý hardware. Dále musí mít vygenerovaný terén určitou míru realismu. Očekáváme rysy jako hory, nížiny, koryta řek, jeskyně apod. Všeobecně nehledáme jeden konkrétní terén, ale funkci schopnou generovat různé terény mající tyto rysy. To můžeme formálně zapsat jako

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

kde různé hodnoty parametru *seed* rozlišují různé terény.

Metody níže zmíněné pracují se spojitým prostorem a mají spojitý výstup:

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

Definujeme tedy prahovou hodnotu $t \in \mathcal{R}$ pro výstup a měřítko $\overrightarrow{s} \in \mathcal{R}^3$, které vztahuje velikost voxelu k rozměrům reálného světa:

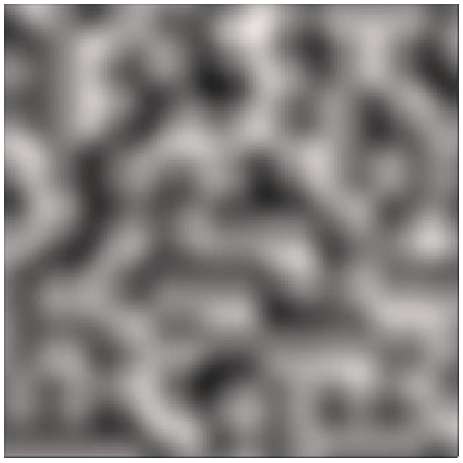
$$proc(\overrightarrow{pos}, seed) = proc_{\mathcal{R}}(\overrightarrow{pos} \otimes \overrightarrow{s}, seed) > t, \tag{2.4}$$

kde $\overrightarrow{a} \otimes \overrightarrow{b}$ je operace skalárního násobení mezi jednotlivými komponentami vektorů. Pro generování trojrozměrného terénu lze využít i funkce pracující se dvěma rozměry. Hodnota funkce v bodu x, y potom udává výšku sloupce terénu na daných souřadnicích:

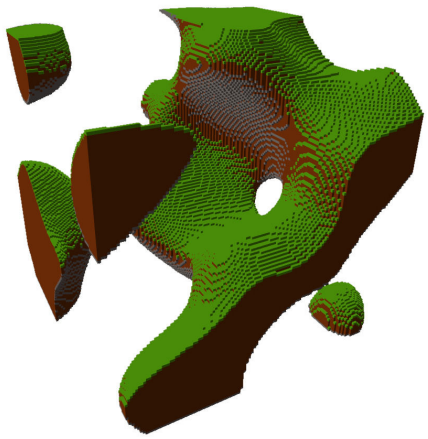
$$proc(\overrightarrow{pos}, seed) = proc_{2\mathcal{R}}(\overrightarrow{pos}_{xy} \otimes \overrightarrow{s}_{xy}, seed) < \overrightarrow{pos}_z \cdot \overrightarrow{s}_z. \tag{2.5}$$

2.1.1 Perlinův šum

Perlinův šum (angl. *Perlin noise*) je hojně využívaná metoda pro generování šumu vytvořená Kenem Perlinem [19, 20]. „Vzhled“ šumu je vhodný pro širokou škálu aplikací, jako například vytváření textur přírodních materiálů, ohně, kouře, mraků. Metoda je aplikovatelná pro libovolný počet dimenzí, trojrozměrná varianta se dá využít pro generování terénů nebo 3D textur. Pro výstupy měnící se s časem (např. mraky) lze přidat čas jako další rozměr.



Obrázek 2.2: 2D Perlinův šum



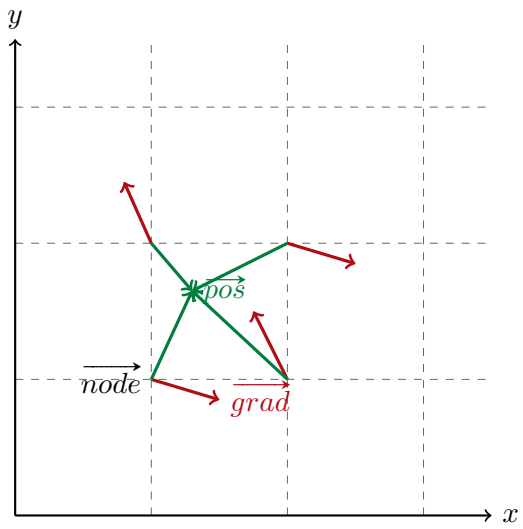
Obrázek 2.3: Volumetrický terén generovaný 3D Perlinovým šumem

Výpočet hodnoty šumu se provede takto: nad prostorem je zavedena jednotková ortogonální mřížka. Pro každý uzel mřížky je pseudonáhodně určena hodnota $\overrightarrow{grad}$ (D je dimenze šumu):

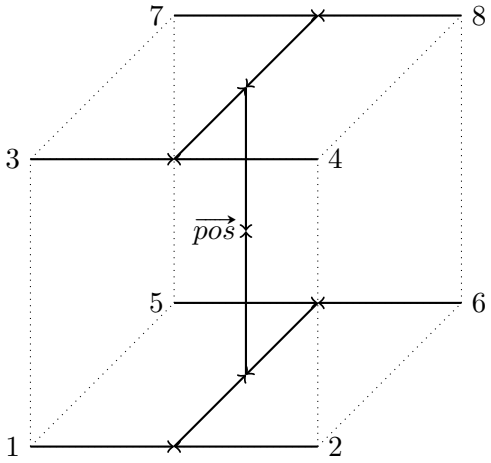
$$\overrightarrow{grad}(\overrightarrow{pos}, seed) : \mathcal{Z}^D \times \mathcal{N} \to \mathcal{Z}^D. \tag{2.6}$$

Pro 2D šum jsou doporučeny vektory jednotkové délky, pro 3D Perlin určuje 12 vektorů (jedná se o vektory do středů hran jednotkové krychle), ze kterých se má pseudonáhodně vybírat. Z mřížky se kolem bodu $\overrightarrow{pos}$, pro který chceme zjistit hodnotu šumu, vybere 2^D uzlů $\overrightarrow{node}_{i \in \langle 1, 2^D \rangle}$ tvořící D-rozměrnou krychli s délkou strany jedna. Pro každý z těchto uzlů je vypočtena hodnota *dot* jako

$$dot_i(\overrightarrow{pos}, seed) = (\overrightarrow{pos} - \overrightarrow{node_i}) \cdot \overrightarrow{grad}(\overrightarrow{node_i}, seed). \tag{2.7}$$



Obrázek 2.4: Vizualizace gradientů a mřížky u 2D Perlinova šumu



Obrázek 2.5: Vizualizace interpolace mezi gradienty uzlů mřížky u 3D Perlinova šumu

Výsledná hodnota šumu je potom získána interpolací mezi hodnotami dot_i:

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

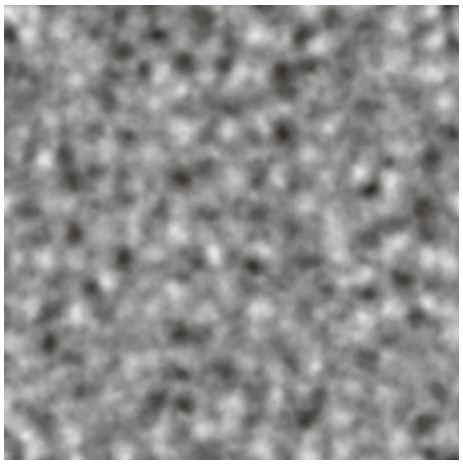
Protože pracujeme s jednotkovou mřížkou, $|\overrightarrow{pos}[j] - \overrightarrow{node_i}[j]|$ bude vždy ležet v intervalu $\langle 0, 1 \rangle$. Jako interpolační funkci *ipol* Perlin [20] definuje

$$ipol(x) = 6x^5 - 15x^4 + 10x^3, \tag{2.9}$$

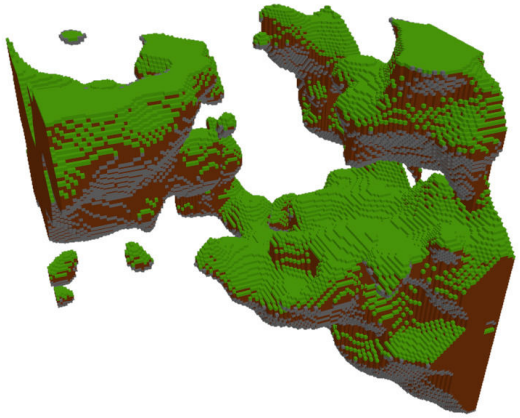
která má pro body $x = 0$ a $x = 1$ nulovou první a druhou derivaci. Výpočet interpolace lze také alternativně chápat jako postupné interpolování mezi dvojicemi hodnot, viz obrázek 2.5. Pro esteticky zajímavější výsledky se šum běžně kombinuje v různých měřítkách a amplitudách:

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

kde O je počet kombinovaných šumů, s je úbytek měřítka a p je úbytek amplitudy. Složky takovéhoto šumu nazýváme také oktávy.



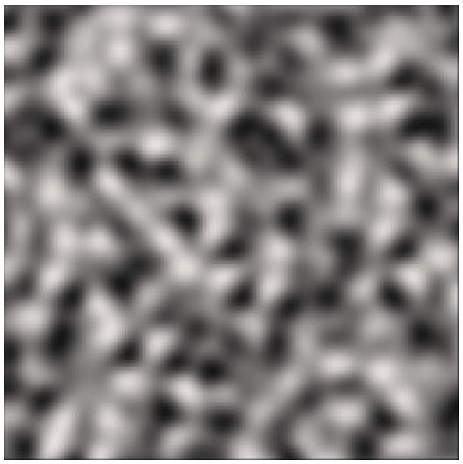
Obrázek 2.6: Výsledek kombinování více 2D Perlinových šumů s různými amplitudami a frekvencemi



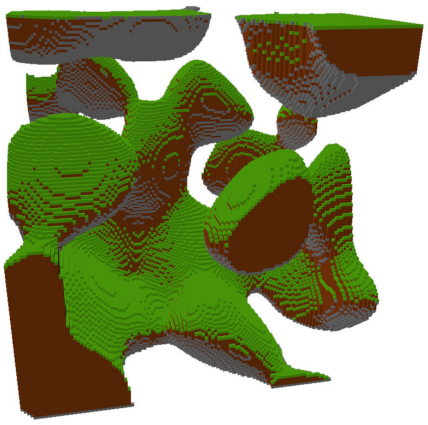
Obrázek 2.7: Volumetrický terén generovaný multioktávovým 3D Perlinovým šumem

2.1.2 Simplex šum

Simplex šum vytvořil Ken Perlin [21] jako *lepší* alternativu k Perlinovu šumu. Jako vylepšení uvádí Perlin mj. izotropii (tj. nezávislost povahy šumu na směru, kterou původní verze nemá), větší výpočetní efektivitu a lepší škálování do více rozměrů. Zatímco u původního Perlinova šumu rostl počet vektorových operací s dimenzí exponenciálně, u nového šumu roste lineárně. Srovnání principů algoritmů shrnuje Stefan Gustavson ve svém článku [13].



Obrázek 2.8: 2D Simplex šum



Obrázek 2.9: Volumetrický terén generovaný 3D Simplex šumem

Primární rozdíl spočívá v mřížce, která v Simplex šumu nerozděluje prostor do D-rozměrných krychlí, ale do tzv. *simplexů*, které mají pouze $D+1$ vrcholů (oproti 2^D u krychlí).

Pro určení vrcholů simplexu, ve kterém se nachází bod $\overrightarrow{pos}$, se využívá této transformace souřadnic:

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

kde D je dimenzionalita šumu. Zpětná transformace má vzorec

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

Pozice prvního vrcholu (nejbližšího k počátku souřadného systému) simplexu se vypočte jako

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

Zbylých D vrcholů se vypočte tak, že se k pozici předchozího vrcholu $\overrightarrow{node}_{i-1}$ přičte $unskew(\vec{k})$, kde $\vec{k}$ má právě v jedné souřadnici s jedničku ($\vec{k}[s] = 1$) a ve zbylých nulu. Hodnota s pro každou $\overrightarrow{node}_i$ je dána následovně: seřadíme indexy souřadnic vektoru $\overrightarrow{offset} = \overrightarrow{vec} - \overrightarrow{node}_1$ sestupně podle hodnoty dané souřadnice. Potom pro $\overrightarrow{node}_i = \overrightarrow{node}_{i-1} + \overrightarrow{vec}[s]$ je s dáno jako index největší souřadnice vektoru $\overrightarrow{offset}$ pro $i = 2$, druhý největší pro $i = 3$ a tak dále. Pro vrcholy se pseudonáhodně vygenerují gradienty stejně jako u Perlinova šumu. Výsledná hodnota šumu se poté vypočte jako

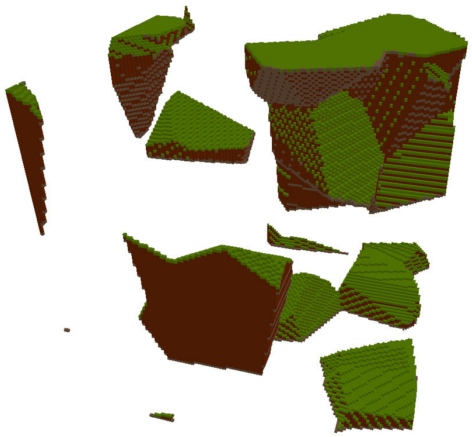
$$simplex(\overrightarrow{pos}, seed) = \sum_{i=1}^{D+1} max(0, 0.6 - |\vec{o}_i|^2) \cdot \left(\vec{o}_i \cdot \overrightarrow{grad}(\overrightarrow{node}_i, seed)\right), \tag{2.14}$$

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

2.1.3 Voroného diagramy

Mějme D-rozměrný prostor a množinu bodů $P \subset \mathcal{R}^D$ náhodně rozmístěných v prostoru. Prostor rozdělíme do regionů dle těchto bodů: ke každému bodu v prostoru $s \in \mathcal{R}^D$ určíme nejbližší bod $p \in P$. Tato struktura se jmenuje Voroného diagram [5].

Obrázek 2.10: Euklidovský 2D Voroného diagram

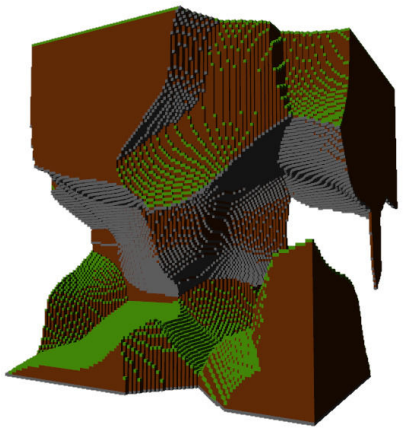


Obrázek 2.11: Volumetrický terén generovaný 3D Voroného diagramem

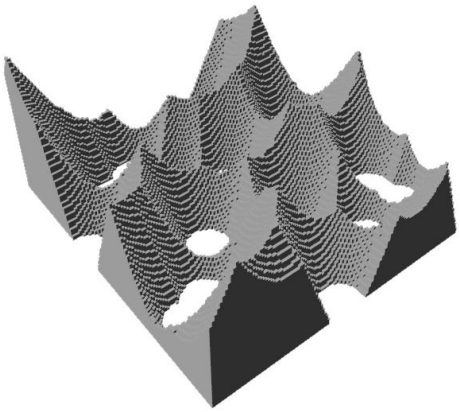
Díky své buněčné struktuře mohou být Voroného diagramy dobrým základem např. pro generování textur organických materiálů, kamenitých materiálů nebo mozaik. Dají se využít pro generování parcel nebo hranic regionů. Zaměříme-li se na hrany mezi regiony, ty se dají využít např. pro pobřežní linie nebo cesty mezi městy, která se nachází na uzlech. Samotný tvar diagramu může být inspirací uměleckého stylu – škála možných využití je široká. Volbou jiné metriky než Euklidovské lze vlastnosti diagramů dále upravovat a nacházet další možnosti aplikací. To demonstruje obrázek 2.12, který je generovaný s metrikou

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

Každému bodu v prostoru lze také přiřadit číselná hodnota odpovídající vzdálenosti k (n-tému) nejbližšímu bodu $p \in P$ – tento přístup využívá šum navržený Stevenem Worleym [27].



Obrázek 2.12: Volumetrický terén generovaný Voroného diagramem s neeuklidovskou metrikou



Obrázek 2.13: Volumetrický terén generovaný z výškové mapy založené na 2D Voroného diagramu; výška stoupá se vzdáleností od nejbližšího bodu $p \in P$

Pro náš formalismus přiřadíme také každému bodu $p \in P$ pseudonáhodně hodnotu $v_p \in \langle 0, 1 \rangle$. Poté

$$voronoi(\overrightarrow{pos}, seed) = v_p, \text{ kde } \vec{p} \in P \text{ a } \rho(\overrightarrow{pos}, \vec{p}) \text{ je minimální mezi všemi } \vec{p} \in P. \tag{2.16}$$

2.1.4 L-systémy

Téma Lindenmayerových systémů (L-systémů) je příliš komplexní na to, aby bylo vhodné zahrnout je celé do této práce. Metodicky jej zpracovává například Prusinkiewicz ve své knize [18]. Ve stručnosti jsou L-systémy definovány jako trojice $H = (V, P, \omega)$, kde V je množina symbolů, P množina pravidel ve tvaru $a \to x$, kde $a \in V$, $x \in V^*$ a $\omega \in V^+$ je počáteční řetězec. Podle pravidel v P se počáteční řetězec ω po určitý počet iterací přepisuje, až vznikne nějaký řetězec α. Tento řetězec se poté graficky interpretuje (k tomu se využívá agent, který jednotlivé symboly řetězce vykonává jako instrukce).

Jsou definovány různé druhy L-systémů; ty zavádějí mj. stochastickou aplikaci pravidel (stochastické L-systémy), větvení v grafické interpretaci (závorkové L-systémy) nebo rozšíření symbolů a pravidel o dodatečné parametry (parametrické L-systémy). Pro tuto práci

by L-systémy mohly nalézt uplatnění při procedurálním generování stromů, trávy nebo jiné vegetace.

Obrázek 2.14: Keř generovaný D0L-systémem, přejato z [24]

2.2 Reprezentace volumetrického terénu

Pakliže lze terén vyjádřit funkcí $ter(x, y, z) \rightarrow \{true, false\}$ z 2.1, lze teoreticky přímo použít tuto funkci při vizualizaci a generovaný terén není třeba jakkoli uchovávat. Taková funkce bude ale v praxi příliš výpočetně náročná na to, aby bylo vhodné ji volat s frekvencí několika desítek invokací za vteřinu pro každý voxel ve vykreslované scéně. Přitom funkci stačí vypočíst pro každý voxel jen jednou a výsledky uložit do vhodné struktury. Součástí zadání práce je navíc možnost terén upravovat, takže potřebujeme nějakým způsobem uchovávat alespoň příznaky změn.

Jako referenční hodnotu pro porovnávání paměťové náročnosti reprezentace dat uvažujme oblast 1024×1024×256, tedy přibližně 270 milionů voxelů. Dále uvažujme čtyři bajty informací na voxel, což odpovídá přibližně 1 GB paměti. V případě (teoreticky) nekonečného procedurálně generovaného světa bude však s pohybem kamery po scéně, a tedy generováním dalších částí terénu, objem terénu růst. Proto a pro umožnění persistence dat je nutné uvažovat mechanismus pro odkládání částí terénu z paměti na disk.

2.2.1 Pole

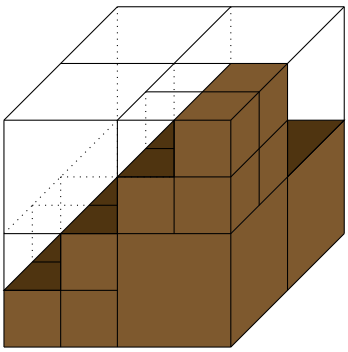
Relevantním řešením je umístění záznamu o každém voxelu do třírozměrného pole. Tento způsob uložení dat umožňuje náhodné čtení i zápis v konstantním čase. Pokud by měl ale celý terén být uložen v jednom poli, při jeho rozšiřování by bylo třeba pole realokovat, přičemž čas potřebný na realokaci lineárně stoupá s objemem terénu. Řešení také neumožňuje odkládání částí terénu na disk.

Pole by tedy bylo vhodnější kombinovat s další datovou strukturou, například *hashovací* tabulkou. Terén by se rozdělil na regiony o fixní velikosti; v rámci těchto regionů by byly voxely uloženy ve statickém poli. Odkazy na regiony by byly uloženy v *hashovací* tabulce, klíč by určoval pozici ve světě. Tato hybridní kombinace si stále uchovává konstantní čas náhodného čtení i zápisu, přičemž přístup k voxelům v rámci jednoho regionu by byl díky statickému poli velice rychlý, stejně tak i iterace. Implementace odkládání regionů na disk by v tomto případě byla triviální.

Nevýhodou tohoto způsobu ukládání dat je, že neumožňuje efektivněji reprezentovat velké homogenní regiony v rámci jednoho pole. Pokud například region obsahuje z 50 % vzduch v souvislém objemu, jiná datová struktura by mohla umožnit menší prostorovou náročnost.

2.2.2 Sparse voxel octree

Řídký oktalový strom (*sparse voxel octree* – SVO) je stromová datová struktura, kde každý uzel je buď list nebo má právě osm potomků. Každý uzel reprezentuje prostorový region, typicky krychli, a pakliže se nejedná o list, tak jeho osm potomků rozděluje tento region na osm menších částí (typicky krychlí o poloviční délce strany vůči rodiči). V našem případě listy představují oblasti terénu se stejnou hodnotou funkce $ter(x, y, z)$.



Obrázek 2.15: Vizualizace reprezentace volumetrického terénu pomocí *octree*

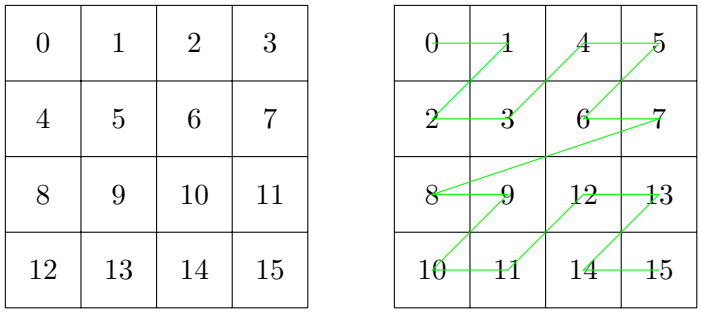
Při náhodném přístupu se struktura prochází sestupně od kořene; náhodné čtení bude mít tedy časovou složitost $O(\log n)$, kde n je objem scény. Čtení bude zahrnovat $O(\log n)$ nesekvenčních přístupů k paměti, což může být dále zpomalováno výpadky *cache*. Zápis má stejnou časovou složitost i stejné problémy a může dynamicky alokovat a dealokovat paměť při rozdělování původně listových uzlů a naopak.

SVO může značně snížit nároky na paměť i výkon v případě, že terén obsahuje veliké homogenní oblasti; celou takovou oblast lze v ideálním případě reprezentovat jedním listem oktalového stromu a vykreslit jako jeden velký blok. Naopak v nejhorší situaci, kdy mají sousední voxely vždy různou hodnotu (střídá se prázdný a plný), může být prostorová i časová náročnost několikařádově vyšší než v případě statického pole.

Odkládání podstromů na disk je pro SVO jednoduše realizovatelné. Pro zabránění snižování výkonu se zvětšujícím se terénem je však vhodné datovou strukturu kombinovat s *hashovací* tabulkou, stejně jako v případě statického pole.

2.2.3 3D textura na GPU

Při ukládání voxelové scény na GPU lze využít 3D textur, které zajišťují efektivní datovou strukturu i funkce pro čtení a zápis. Konkrétní formát ukládání dat není na úrovni grafických knihoven definován a může se v závislosti na ovladačích/grafické kartě lišit. Giesen [11] popisuje používané metody ukládání dat textur na GPU, které vycházejí z lineární paměti s tím, že se zavádí [swizzling], při kterém se permutují bity adresy texelu. Vhodnou volbou permutace lze dosáhnout snížení vzdálenosti adres sousedních texelů textury, např. vytvořením tzv. *Z-order*. Tato metoda nijak nenavyšuje velikost textury, ukládání voxelové scény v GPU texturách by tedy mělo využívat stejné množství paměti jako lineární pole na CPU.



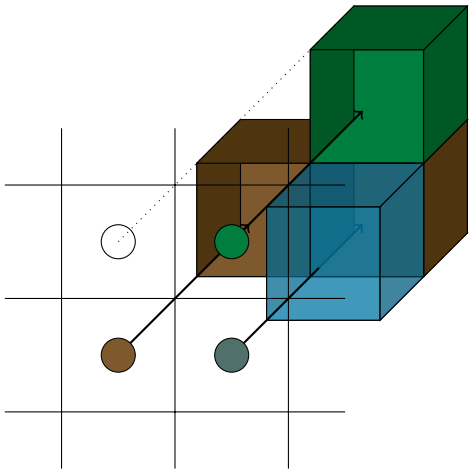
Obrázek 2.16: Demonstrace *Z-order swizzlingu* při ukládání 2D texturových dat na GPU

2.3 Vizualizace volumetrického terénu

Elvins ve svém přehledu [1] rozděluje metody vizualizace volumetrického terénu na metody přímého vykreslování (*direct volume rendering, DVR*) a metody realizující převod na hraniční reprezentaci (*surface-fitting, SF*).

2.3.1 Metody přímého vykreslování

Metody přímého vykreslování, jak název napovídá, pracují s volumetrickými daty „přímo“ – tedy si nevytváří jejich geometrickou reprezentaci. V kontextu této práce to znamená, že se terén nebude vykreslovat „kostka po kostce“. Typickým představitelem *DVR* je *ray casting*: pro každý pixel obrazovky je do scény „vržen“ paprsek, barva pixelu je pak určena podle voxelu, který odpovídající paprsek protne.

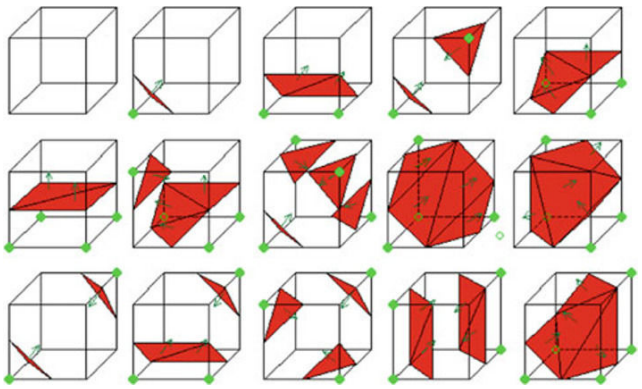


Obrázek 2.17: Ilustrace principu *ray castingu*

Průchod paprsku nemusí být zastaven prvním neprázdným voxelem: daný voxel může být částečně průhledný. V tom případě paprsek pokračuje dál a výsledná barva se míchá. Metoda *ray castingu* není osvětlovací technika, ale pouze vizualizační. Existují metody, které s paprsky pracují i za účelem fotorealistického osvětlení (typickým příkladem je *ray tracing*) – ty jsou zmíněny níže v oddílu 3.1.

2.3.2 Metody založené na hraniční reprezentaci

Surface-fitting metody převádějí volumetrická data na hraniční reprezentaci, která je poté vykreslována tradičními technikami. V naší situaci je nejjednodušší takovou technikou prosté vykreslení stěn neprázdných voxelů v terénu. Ta produkuje vždy kolmé stěny, z čehož je zřetelně vidět krychlová struktura. Tuto strukturu potlačuje například metoda *marching cubes* naznačená v obrázku 2.18. Existují i další metody, které pracují s křivkami a normálovými vektory a které se snaží hraničně reprezentovat i částečně průhledná voxelová data; lze se o nich dočíst například v knize Moderní počítačová grafika [14].



Obrázek 2.18: Varianty reprezentace hranic voxelů v metodě *marching cubes*, převzato z [6]

Kapitola 3

Grafické efekty

Ve skutečném světě je obraz, který vnímáme, zprostředkován fotony, které dopadají na sítnici našeho oka. Ačkoli lidské oko je schopné detekovat dopad jediného fotonu [25], za denního světla na sítnici našeho oka dopadají desítky bilionů fotonů za vteřinu.[1] Tyto fotony jsou po cestě od zdroje světla k oku odráženy, lámány, pohlcovány a opět vyzařovány. V počítačové grafice je snahou důsledky těchto jevů reprodukovat. V této kapitole shrneme vybrané jevy a techniky zabývající se touto problematikou.

3.1 Osvětlení

Samotné metody vizualizace bez modelu osvětlení předpokládají všudypřítomné zdroje světla v každém bodě scény. Krokem k fotorealismu je tedy zavedení světel – situovaných zdrojů fotonů. Fotony cestují ze zdroje světla do pomyslného oka pozorovatele; po cestě interagují s objekty ve scéně. Základní druhy interakce světla s objektem jsou odrazy od difuzního a zrcadlového (*specular*) tělesa. Při zrcadlovém odrazu se paprsek odráží ve stejné rovině – tento odraz bývá spojován s lesklými a zrcadlovými povrchy. Při difuzním odrazu se paprsek odráží náhodně všemi směry – tento odraz bývá spojován s matnými povrchy a nejvíce se podílí na vnímané barvě tělesa.

Žára ve své knize [14] dělí metody osvětlení na lokální a globální. Metody lokálního osvětlení jsou všeobecně výpočetně méně náročné, protože při výpočtech uvažují pouze zdroj světla, objekt a pozorovatele – paprsek při své cestě od zdroje světla k pozorovateli změní směr maximálně jednou, kdy se odrazí od nějakého předmětu. Metody globálního osvětlení naopak pracují se scénou jako s celkem, paprsek se při své cestě může odrazit vícekrát od různých objektů.

3.1.1 Metody globálního osvětlení

Klasickým představitelem metod globálního osvětlení je *ray tracing*. Metoda pro každý pixel obrazovky vysílá do scény paprsky a pro každý odraz paprsku od nějakého objektu počítá osvětlení objektu v daném bodě; počet odrazů je limitován. Metod sledujících trasy jednotlivých fotonů je více, liší se mj. výpočty při kolizi fotonu s tělesem nebo směrem sledování paprsků (od pozorovatele ke zdrojům světla nebo obráceně). Protože se současným hardwarem není výpočetně proveditelné v rozumném čase vypočítat výsledky analytickou metodou z rovnic popisujících chování světla, provádí se náhodné vzorkování metodami

[1] Velmi přibližný odhad, dle https://physics.stackexchange.com/questions/329971

Monte Carlo. Kvalita výsledku poté závisí na počtu vzorků vypočtených těmito metodami (tedy počtu vržených paprsků) – při nedostatečném počtu vzorků se výsledný obraz jeví jako zrnitý. I přes využití Monte Carlo metod ale výpočetní možnosti běžného uživatelského hardware nestačí k provádění výpočtů v reálném čase a tyto metody jsou tak využívány spíše jen pro *renderování* videí a statických obrázků. To se dnes pomalu mění s vývojem grafických karet optimalizovaných pro tyto aplikace, jako jsou nové nVidia RTX™[3].

Existují i další metody globálního osvětlení – *voxel cone tracing* [7] například nepracuje s paprsky jako s přímkami, ale jako s kužely. Jako doplněk k *ray tracingu*, který dostatečně dobře aproximuje chování lesklých povrchů, se používá *metoda radiozity*, která je vhodná pro simulaci nepřímého difuzního osvětlení.

3.1.2 Metody lokálního osvětlení

Jak již bylo zmíněno, metody lokálního osvětlení při výpočtech uvažují pouze zdroje světla, objekt a pozorovatele. Nedochází k nepřímému osvětlení ze sousedních objektů ani odrazům. Mezi zdrojem světla a bodem na tělese se neuvažují žádné překážky, ani objekt samotný. Lokální modely jsou využity i jako základ globálního osvětlení.

V roce 1977 navrhl Bui-Tuong Phong [22] empirický (tedy ne založený na fyzice) model pro výpočet lokálního osvětlení, který se stal standardem pro aplikace vykreslující v reálném čase. Model pracuje se třemi složkami osvětlení:

- **Ambientní složka** není nijak ovlivněna polohou světel ani objektu vůči pozorovateli. Popisuje „všudypřítomné světlo okolí“, je konstantní pro celou scénu a slouží k tomu, aby neosvětlené části těles nebyly zcela černé (v reálném světě také běžně není absolutní tma).
- **Difuzní složka** má zpravidla největší vliv na pozorovanou barvu objektu. Její intenzita závisí na úhlu osvětlené plochy vůči zdroji světla, ale ne na úhlu pozorovatele.
- Intenzita **zrcadlové složky** závisí jak na úhlu zdroje světla a osvětlené plochy, tak na úhlu plochy vůči pozorovateli. Největší intenzity nabývá, pokud jsou světlo, osvětlený bod a pozorovatel v jedné rovině a úhel mezi plochou a světlem je stejný jako úhel mezi plochou a pozorovatelem.

Jednotlivé složky mohou mít u různých materiálů různý vliv, například spekulární složka je u matných materiálů potlačena. Výsledné osvětlení objektu je pak suma těchto tří složek pro všechna světla ve scéně. Tento výpočet se provádí pro každý vykreslovaný bod scény, případně při *deferred shadingu* pro každý bod obrazovky.

3.1.3 Shadow mapping

Protože samotné metody lokálního osvětlení neobsahují mechaniky pro určení, zda je daný bod od daného světla zastíněn, je třeba tento problém řešit dodatečně. Obecně platí, že pokud je mezi bodem na tělese a zdrojem světla nějaký objekt (i třeba zkoumané těleso samotné), je bod zastíněn a světlo na tento bod nepůsobí. Neuvažujeme zde poloprůhledné objekty, které by problematiku dále zesložiťovaly. Jednou z metod, které problém stínů řeší, je *shadow mapping* Lance Williamse [26]. Ten pro každé světlo vykreslí scénu znovu; vždy „z pohledu“ daného světla, přičemž důležitá je hloubková informace uchovávaná v Z-bufferu. Při testování toho, zda je daný bod z pohledu pozorovatele zastíněn či nikoli, se poté uvažuje přímka mezi bodem a daným světlem: pokud vzdálenost mezi bodem a zdrojem

světla odpovídá hodnotě ze Z-bufferu světla, znamená to, že světlo na daný bod dopadá; pokud ne, je bod zastíněn.

3.2 Další grafické techniky

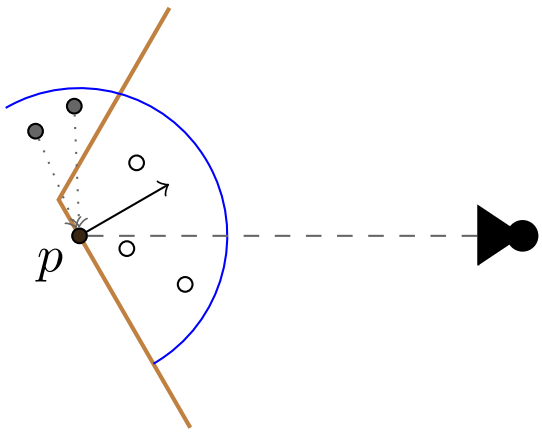
Ačkoli existují složitější vykreslovací/osvětlovací modely pokrývající širokou škálu optických jevů, pro svou složitost nebývají vhodné pro aplikace zobrazující v reálném čase. Proto bývají používány jednodušší modely, které tyto jevy nereplikují, a aproximace různých jevů pak implementovány dodatečně jako samostatné efekty.

3.2.1 *Depth of field*

Depth of field (DoF) je jev způsobený optikou lidského oka, případně kamery, které jsou schopné zaostřit pouze na jednu konkrétní hloubku; s rostoucí vzdáleností od této hloubky jsou objekty rozostřenější. Různé způsoby implementace DoF jsou popsány mj. v knize GPU Gems [10]. Jednou z technik je například *Reverse-Mapped Z-Buffer Depth of Field*, která pracuje ve *screen space* a která na každý pixel aplikuje rozostření v míře závisející na jeho vzdálenosti od hloubky ostrosti (tato informace se získá z *depth bufferu*).

3.2.2 *Screen Space Ambient Occlusion*

Ambient Occlusion je jev globálního osvětlení, kde na místa těles v zúžených, konkávních oblastech dopadá méně *ambientního* světla (protože je kolem nich hodně hmoty, která světlo zastiňuje). Jedná se typicky o rohy místností, dutiny apod. – v důsledku tohoto jsou tyto oblasti ztmavené. Jedna z technik, která tento jev aproximuje při použití metod lokálního osvětlení, je *Screen Space Ambient Occlusion* (SSAO). Ta pro každý pixel p obrazovky vzorkuje několik okolních hodnot *depth bufferu* a odvozuje, kolik těchto vzorků se nachází uvnitř polokoule danou bodem p a jeho normálou. Každý vzorek, který se nachází uvnitř této polokoule, potom přispívá ke ztmavení pixelu p (viz obrázek 3.1). Tato technika byla poprvé představena v CryEngine 2 [17].



Obrázek 3.1: Princip *screen space ambient occlusion*

3.2.3 *God rays*

Krepuskulární paprsky, také nazývané *sun rays* nebo *god rays*, jsou jev způsobený světlem pronikajícím ne zcela průhledným vzduchem. Vzduch může světlo rozptylovat skrze prachové částice, částice neprůhledných plynů, aerosoly apod. V důsledku toho se sám vzduch, jímž světlo prochází, jeví, jako by mírně zářil. Je-li zdroj světla částečně zakryt, vzduch za překážkou je zastíněn. Střídaní osvětlených a zastíněných částí vzduchu tak vyvolává dojem, že pozorovatel vidí dráhu paprsků světla.

Grafický efekt replikující tento jev se nazývá *volumetric light scattering*. Možné způsoby implementace se různí. Kenny Mitchell [2] popisuje metodu, která míru rozptýleného světla v každém pixelu obrazovky odvozuje tak, že vzorkuje *depth buffer* v bodech mezi daným pixelem a zdrojem světla ve *screen space*; míra rozptýlení pak závisí na tom, jaká část vzorků se nachází za zdrojem světla a jaké body zdroj zastiňují. Benjemin Glatzel ve své prezentaci na konference Digital Dragons 2014 [12] popisuje metodu, která pro každý pixel vzorkuje body na paprsku vrženém do scény a testuje, zda se daný vzorek nachází ve stínu.

Obrázek 3.2: Fotografie s viditelnými krepuskulárními paprsky, autor Les Chatfield

3.2.4 *Order independent transparency*

Vykreslování průhledných primitiv vyžaduje řešení dodatečných problémů. Operace prolínání průhledných pixelů totiž není komutativní, tedy je pro korektní zobrazení nutné znát jejich pořadí z pohledu kamery. To tradičně vyžadovalo seřazení primitiv dle vzdálenosti od kamery a postupné vykreslování odzadu dopředu. Pro scény s velkým počtem primitiv však může nutnost jejich řazení přinášet velké zpomalení. Proto byly zkoumány techniky, které toto řazení nevyžadují – tzv. *order independent transparency* metody. Tyto techniky řeší řazení nikoli na úrovni primitiv, ale na úrovni vykreslovaných fragmentů. Přístupů je několik; lze zmínit například *stochastickou průhlednost* [8], kde jsou fragmenty vykreslovány s pravděpodobností odpovídající jejich průhlednosti. Při použití této metody vzniká šum, který je potlačován vícenásobným vykreslením průhledných primitiv a průměrováním. Yang [28] prezentuje metodu, která pro každý bod obrazovky generuje lineární seznam průhledných fragmentů, který je dále zpracován. Výhodou této metody je, že vyžaduje jediný průchod vykreslení průhledných objektů. McGuire [16] také přichází s přístupem, který vyžaduje jediný vykreslovací průchod; nekomutativní funkce pro prolínání je zde nahrazena komutativní aproximací.

Dalším přístupem je tzv. *depth peeling* [9], který zavádí několik vykreslovacích průchodů pro průhledné objekty; v každém průchodu je proveden dodatečný hloubkový test, který zahodí ty fragmenty, které jsou ve stejné hloubce nebo blíže ke kameře, než je fragment na stejných souřadnicích z předchozího průchodu. V důsledku toho s každým průchodem vznikne jedna průhledná vrstva; výsledný obraz pak vzniká kombinací těchto vrstev. Další publikace přinášejí různé optimalizace této techniky, např. *dual depth peeling* [4].

Kapitola 4

Návrh

V této kapitole jsou popsány jednotlivé aspekty návrhu systému pro generování, reprezentaci a vizualizaci volumetrického terénu. Předmětem kapitoly je pouze obecná koncepce, implementační detaily jsou poté zdokumentovány v kapitole 5.

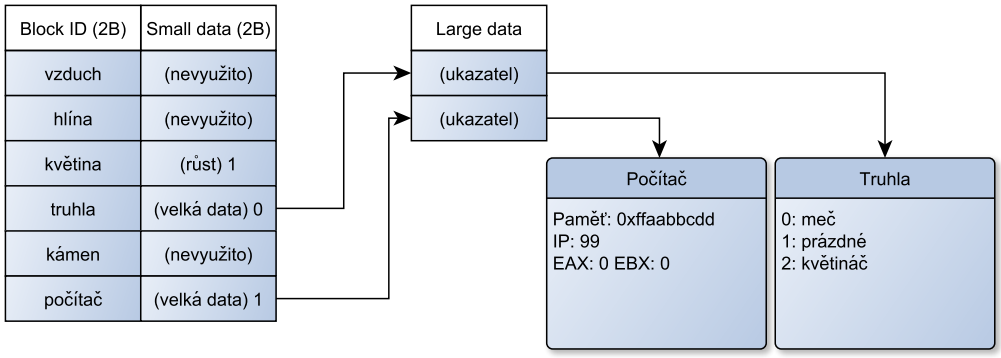
4.1 Reprezentace světa

Pro reprezentaci světa v paměti byla zvolena tato hierarchická struktura:

- Svět je rozdělen do tzv. *chunků*, které ho dělí do oblastí o 16×16×256 voxelech (v dokumentu budeme voxely označovat i jako *bloky*). Chunky jsou organizovány hashovací tabulkou, kde klíč je pozice chunku ve světě.
- V chunku jsou jednotlivé voxely uloženy v lineárním poli.

Hashovací tabulka umožňuje mít v paměti načtenou pouze oblast kolem hráče (svět je potenciálně nekonečný) a jednotlivé chunky mohou být načítány a odkládány podle toho, jak se hráč hýbe. Přístup k jakémukoli bloku ve světě je přitom velmi rychlý a s konstantní složitostí. Uvažujeme heterogenní voxelová data s velikou rozmanitostí, kdy by stromové struktury měly oproti poli větší paměťové nároky a vícenásobné indirekce by značně snižovaly propustnost paměti. Ukládání dat do pole navíc oproti jiným strukturám umožňuje data velice jednoduše kopírovat z a na GPU, čehož je v tomto projektu hojně využíváno. Stejně jednoduché je i ukládání a načítání z disku.

Pro každý voxel se ukládá dvoubajtová hodnota určující typ voxelu (*block ID*); pro účely této práce by stačila pouze jednobajtová hodnota, nicméně pokud by se projekt měl rozvíjet dál, tak ze zkušenosti s podobnými hrami je zřejmé, že 256 typů voxelu není dostatečné. Ačkoli tento mechanismus není v práci plně implementován ani demonstrován, pro ukládání dodatečných dat (např. růst květin, informace o otevření/zavření dveří, ...) se každému voxelu vyčleňují další dva bajty (*small data*). Pokud by voxel potřeboval uchovávat více dat (např. pokud se jedná o truhlu obsahující inventář), v této dvoubajtové hodnotě bude uložen index do dynamicky alokovaného pole, ve kterém bude odkaz na strukturu s daty (*large data*).



Obrázek 4.1: Struktura dat pro ukládání informací o voxelech

Výška světa je omezena na jeden chunk: kvůli výpočtům denního osvětlení by stejně všechny chunky po vertikální ose (z) musely být načteny (výpočty popsány v oddíle 4.3); omezování dohledové vzdálenosti do výšky by také příliš nedávalo smysl a mohlo by být pro hráče zbytečně matoucí. Uspokojivých výsledků však lze dosáhnout s relativně malým výškovým maximem (oproti dohledové vzdálenosti v horizontálním směru) a prostředky potřebné pro větší výškový rozsah mohou být lépe využity pro zvýšení horizontální dohledové vzdálenosti.

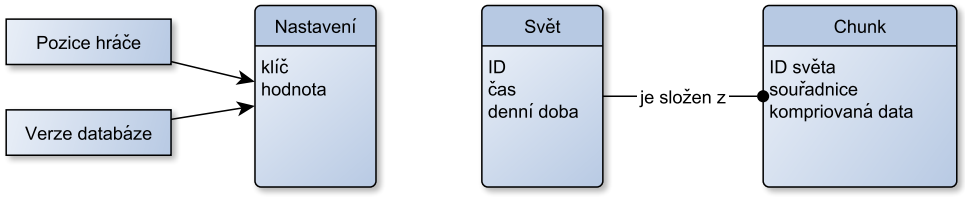
Volba velikosti chunku je ovlivněna mnoha faktory:

- Chunk by neměl být příliš malý, aby se nároky na jeho režii dostatečně rozdělily mezi voxely, které sdružuje.
- Rozměry by měly být mocniny dvou kvůli různým optimalizacím.
- Pro jednoduchost by šířka a délka chunku měly být stejné.
- Výška chunku určuje výšku celého světa: je třeba dostatečný rozsah pro podzemí, oceán a hory.
- Pro implementační jednoduchost by měly všechny jevy ovlivňovat okolí do vzdálenosti maximálně jednoho chunku (8-okolí). To například znamená, že světlo z nějakého světla by se mělo být schopno šířit nejdále do sousedních chunků v 8-okolí. Chunk by tedy měl být dostatečně velký, aby mohl být dostatečně velký i dosah světla.
- Objem chunku by měl odpovídat nebo být menší vůči rozsahu hodnot, které lze uložit jako dodatečná data k voxelu. Takto bude zajištěno, že každý voxel v chunku může mít *large data*.

Na základě těchto kritérií byla zvolena právě velikost 16×16×256. Výška 256 rozumně pokrývá nároky generátoru terénu; některé generované hory mají převýšení i přes 100 bloků, a menší výška (128) by tedy byla značným omezením (kromě hor je třeba dedikovat i jisté výškové vrstvy podzemí, mořské vodě a nadmořské výšce). Horizontální rozměry 16×16 jsou dostatečně malé na rozumné členění terénu a současně je maximální dosah světla 16 bloků dostatečný. Hodnoty 0–15 se dají reprezentovat čtyřmi bity, což se dobře zarovnává do bajtů. A objem chunku je 2^{16}, což přesně odpovídá dvěma bajtům, do kterých se ukládají *small data*, resp. adresy na *big data*, voxelů.

4.2 Ukládání světa na disk

Jako datový formát určený pro ukládání světa na disk byla zvolena databáze SQLite. Jednotlivé chunky jsou reprezentovány řádky tabulky `chunks`, ke kterým se přistupuje pomocí klíče definovaného souřadnicemi chunku. V dalším sloupci jsou pak v binárním formátu uloženy *block IDs* (1 : 1 kopie z paměti komprimovaná pomocí zlib). *Small data* ani *large data* nejsou implementována.



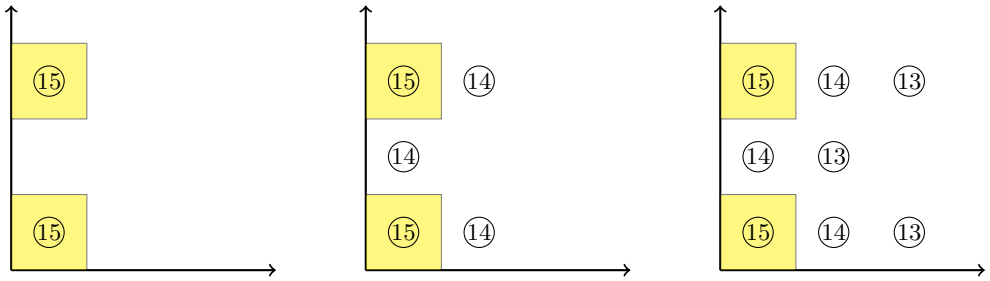
Obrázek 4.2: Struktura dat pro ukládání světa na disk v databázi SQLite

SQLite obstarává veškerou režii fragmentace místa na disku pro data jednotlivých chunků, celé řešení je tak velice jednoduché a univerzální, vše je uloženo v jednom souboru. Díky kompresi je uložený svět relativně malý (velikost souboru se zvětšuje s tím, jak hráč prochází světem a generují se další chunky).

4.3 Osvětlovací model

Základ osvětlovacího modelu je shodný s tím ve hře Minecraft. Model má konstantní složitost v závislosti na počtu světel. Lze ho chápat jako metodu *Light Propagation Volumes* se sférickou harmonickou funkcí prvního stupně [15]. Dá se také popsat celulárním automatem.

Nad světem je zavedena krychlová mřížka o stejné velikosti, jakou má mřížka voxelová; uzly mřížky jsou situovány do středů voxelů. Pro každý uzel mřížky je definována hodnota úrovně světla; ta neobsahuje žádné směrové informace. Světlo je mřížkou šířeno tak, že s každým krokem od zdroje jeho úroveň klesne o 1 (plus další útlum definovaný typem bloku). Zdroje světla jsou kombinovány nikoli aditivně, ale pomocí funkce max.



Obrázek 4.3: Demonstrace šíření světla dle modelu použitého v práci (ve 2D, jednotlivé diagramy odpovídají krokům výpočtu)

Světlo je složeno ze čtyř složek: červená, zelená, modrá a denní světlo. Hodnota každé složky je uložena ve 4 bitech, dohromady tedy každý uzel potřebuje 2 bajty. Barva denní složky světla se mění v závislosti na denní době; směrem dolů se v chunku složka pro denní světlo šíří bez úbytku intenzity.

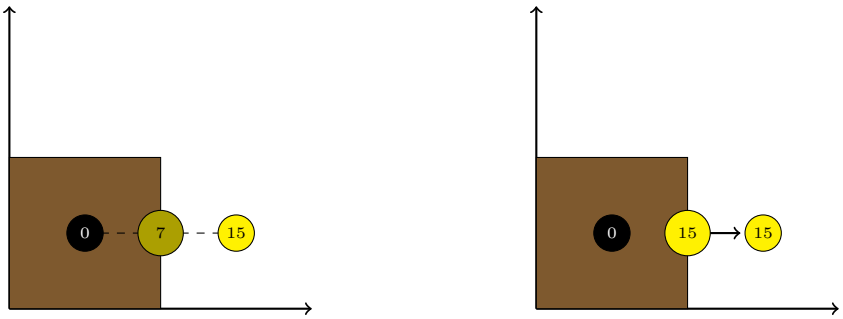
Výpočet osvětlení tedy probíhá takto:

1. Nastav hodnoty všech uzlů na nula; pokud je voxel odpovídající danému uzlu zdroj světla, nastav hodnotu uzlu dle jeho intenzity.

2. Jdi odshora dolů a v každém sloupci postupně nastavuj hodnotu složky denního světla. Nastavovaná hodnota začíná nahoře na maximální intenzitě a klesá na základě průhlednosti bloků ve sloupci.

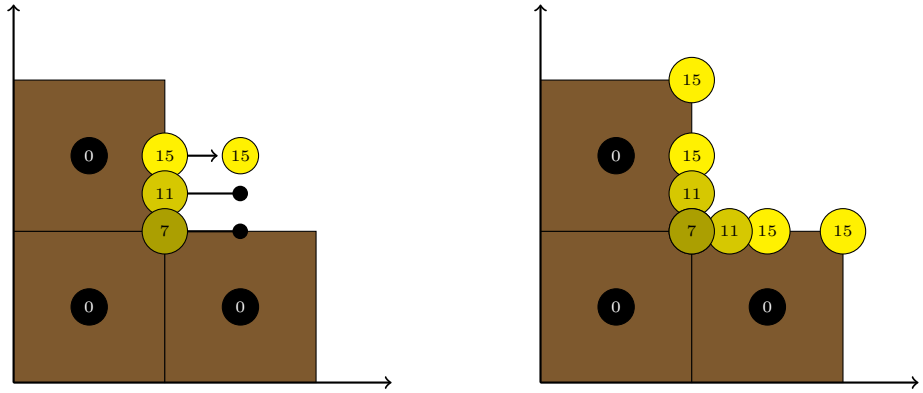
3. Iteruj, dokud se hodnoty mění: uprav hodnotu v každém uzlu podle následujícího vzorce: $v_x \leftarrow \max(v_x, n_{0,x} - d, ..., n_{5,x} - d)$, kde $x \in \{R, G, B, D\}$ jsou jednotlivé složky světla, $n_{i,x}$ jsou intenzity světla sousedních uzlů (v 6-okolí) a $d = 1 + d_v$ je útlum světla (d_v je útlum definovaný typem bloku na dané pozici, 0 = žádný útlum).

Pro libovolnou pozici ve světě se hodnota osvětlení dá vypočítat lineární interpolací mezi osmi nejbližšími uzly mřížky. Při vykreslování stěn bloků vzorkujeme tuto mapu osvětlení s posunem o půl bloku ve směru normály; bez toho bychom u neprůhledných bloků získávali příliš tmavé hodnoty, protože stěna je v polovině vzdálenosti mezi vnějším uzlem (který je osvětlen) a uzlem ve středu bloku (který není osvětlen) a interpolace by tedy navracela hodnotu zkreslenou tmou uprostřed bloku.

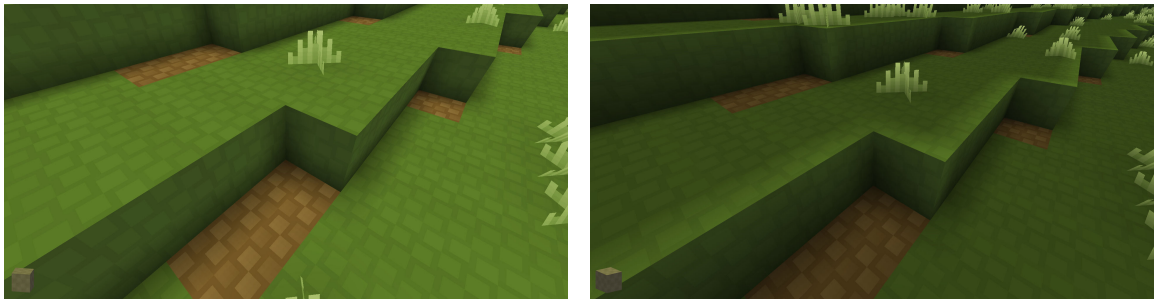


Obrázek 4.4: Demonstrace posunu vzorkovaní osvětlení o půl bloku ve směru normály (ve 2D): bez posunu (vlevo) a s posunem (vpravo)

Důsledkem tohoto modelu je i přirozený *ambient occlusion*, kdy dochází ke ztmavení ve vnitřních rozích:

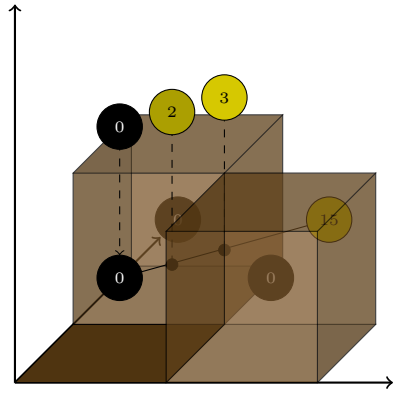


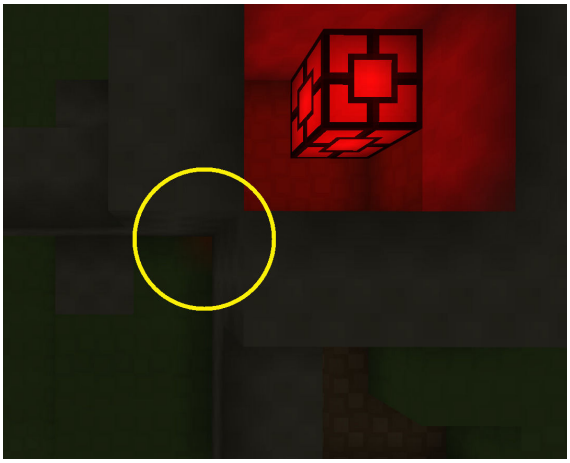
Obrázek 4.5: Demonstrace přirozeného *ambient occlusion* v modelu osvětlení (ve 2D)



Obrázek 4.6: Model osvětlení se vzorkováním s posunutím normálou (vlevo) a bez posunutí normálou(vpravo)

Nevýhodou zvoleného osvětlovacího modelu je, že v určitých situacích produkuje artefakty, kde světlo „přetéká“ skrze ostré hrany (hrany, které oddělují v 4-okolí [resp. 6-okolí ve 3D], ale ne v 8-okolí [resp. 26-okolí]):



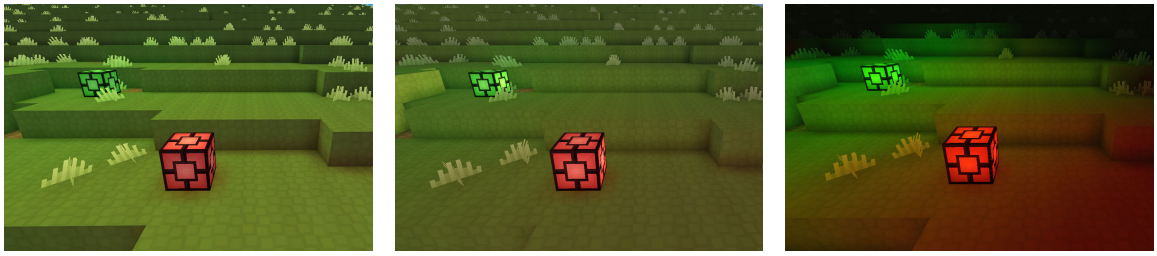


Obrázek 4.7: Artefakt způsobený osvětlovacím modelem: světlo proniká skrze ostré rohy

Denní světlo je dále rozděleno na dvě složky – ambientní a směrovou. Výsledná barva pixelu se vypočte jako

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

kde $\vec{c}_{albedo}$ je barva pixelu před stínováním, $\vec{l}_{rgb}$ je vektor RGB složek osvětlení pro daný bod, g je vlastní emise daného bodu (může být předávána v alfa kanálu textury), $u \in \langle 0, 1 \rangle$ je útlum umělých světel, l_{day} je denní složka osvětlení pro daný bod, $\vec{d}_{amb}$ je barva ambientní složky denního světla, $\vec{n}_{face}$ je normála objektu v daném bodě, $\vec{n}_{sun}$ je normála udávající směr denního světla a $\vec{d}_{dir}$ je barva směrové složky denního světla. Během dne je útlum umělých světel až $u = 0.8$, což má reflektovat to, že denní světlo je silnější než umělé zdroje osvětlení, a tedy se na denním světle tyto zdroje jeví slabší. Barva a směr denního světla se může měnit, aniž by se musela znovu počítat propagace světla.



Obrázek 4.8: Demonstrace směrového efektu denního světla a útlumu umělého osvětlení

Současně s tímto modelem je zaveden i běžný *shadow mapping*, ten je však počítán pouze pro sluneční světlo (a tedy ne pro světlo vydávané voxely).

Návrh tohoto modelu je shodný s osvětlovacím modelem hry Minecraft v celulárním způsobu osvětlení, v maximálním dosahu světla a v prezenci denní složky. Detailnější informace o způsobu implementace osvětlení ve hře nejsou známy. Oproti Minecraftu je zavedeno RGB osvětlení, směrová složka denního světla a mechanismus vzorkování světelných dat s posunutím normálou.

4.4 Generování terénu

Procedurální generování terénu využívá 2D a 3D Perlinova šumu a Voroného diagramů, jejichž principy jsou popsány v oddílu 2.1. Část výpočtů je prováděna ve 2D, tedy pro celý sloupec voxelů se stejnými x, y souřadnicemi (výšku určuje souřadnice z). Nejprve se pro každý bod ve 2D určí hodnoty popisující vlastnosti terénu (jde o 2D Perlinův šum s vysokou velikostí oktáv); tyto hodnoty mají sémantiku např. *míra stromů*, *míra pouště*, *hornatost* apod. Budeme je zapisovat ve formátu e_{stromy}.

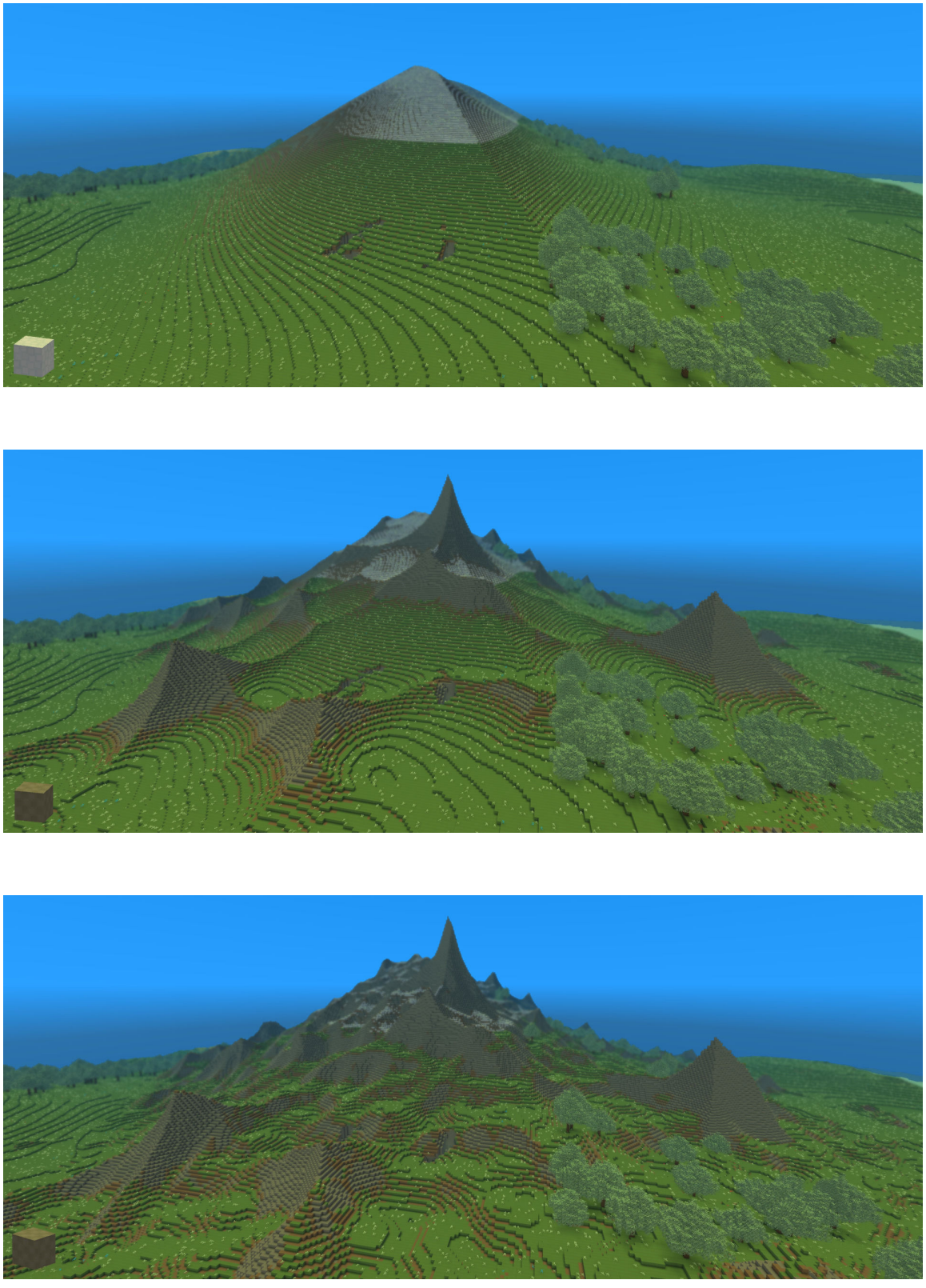
Terén je generován na základě výškové mapy, která je dána vzorcem:

$$z = \begin{cases} z_{ocean} \text{ pro } e_{ocean} > 0 \\ z_{zaklad} + \min(z_{reky}, \max(z_{poust}, z_{kopce}, z_{hory}, z_{plane})) \text{ pro } e_{ocean} <= 0 \end{cases} \tag{4.2}$$

z_{zaklad} zastupuje základní nadmořskou výšku terénu, ke které se přičítají další prvky krajiny, a je vypočtena Perlinovým šumem s relativně velkou velikostí oktávy. z_{plane} popisující pláně a z_{kopce} popisující kopce jsou také pouze Perlinovy šumy. Ostatní parametry budou (neformálně) popsány v následujících oddílech. Většina těchto hodnot je upravena (vynásobením s hodnotou odvozenou od e_{ocean}) tak, aby se u pobřeží blížila k nule. Typ bloku na úrovni země je určen na základě e_{XX} koeficientů a na základě gradientu výškové mapy.

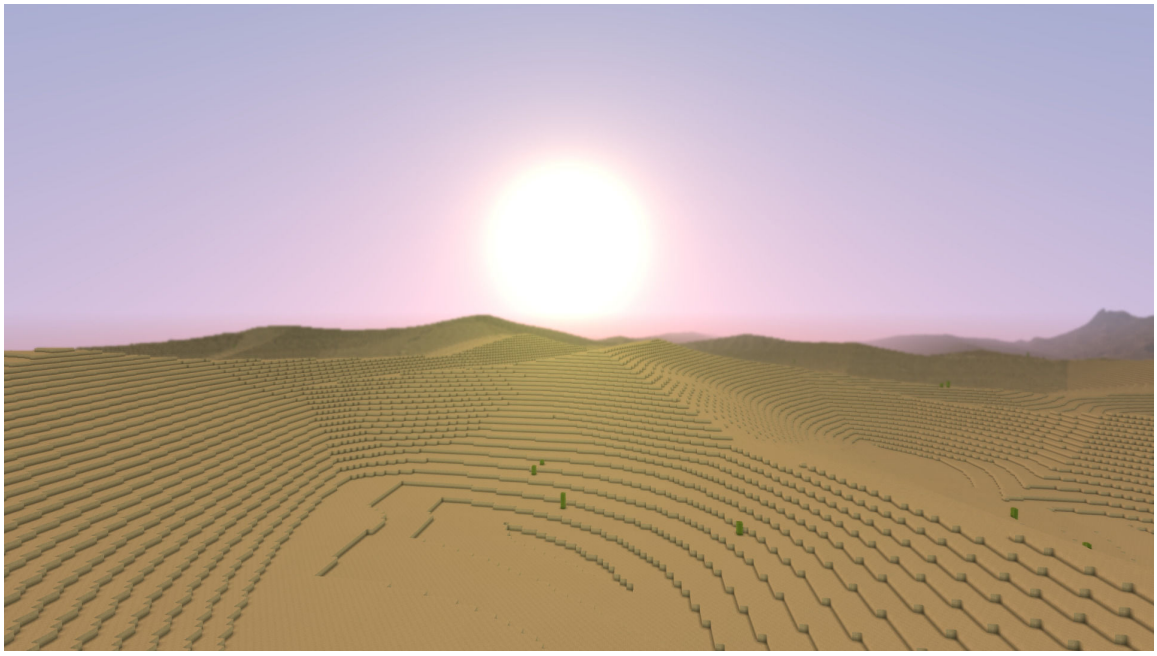
4.4.1 Hory

Hory jsou generovány kompozicí dvou 2D Voroného diagramů a jednoho 2D Perlinova šumu. Princip generování hornatého terénu byl demonstrován na obrázku 2.13 v sekci 2.1.3: výška terénu stoupá se vzdáleností od nejbližšího generovaného bodu, což vytváří ostré hrany připomínající horský masiv. Tento princip je při generování hor uplatněn dvakrát: jednou ve větším měřítku na samotné vrcholy a hřebeny, podruhé v menším měřítku na sekundární vrásnění. K těmto dvěma komponentám je ještě přičten Perlinův šum pro nepravidelné nerovnosti.



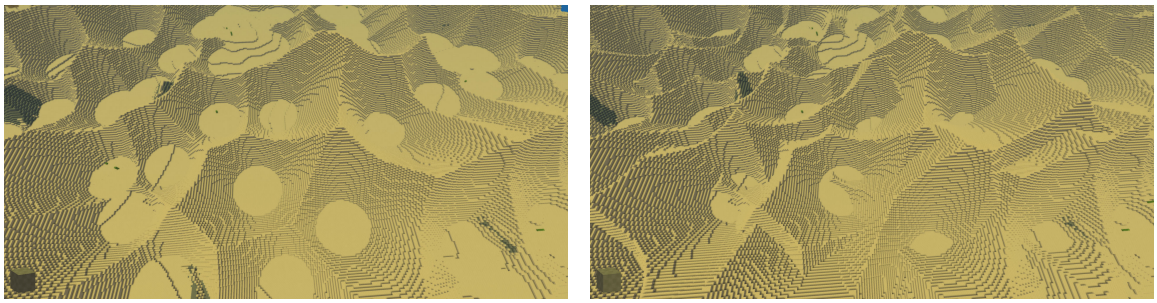
Obrázek 4.9: Tři komponenty při generování hor: Voroného diagram pro hřebeny, Voroného diagram pro sekundární vrásnění a Perlinův šum pro nerovnosti (postupně aplikováno odshora)

4.4.2 Pouště



Obrázek 4.10: Poušť

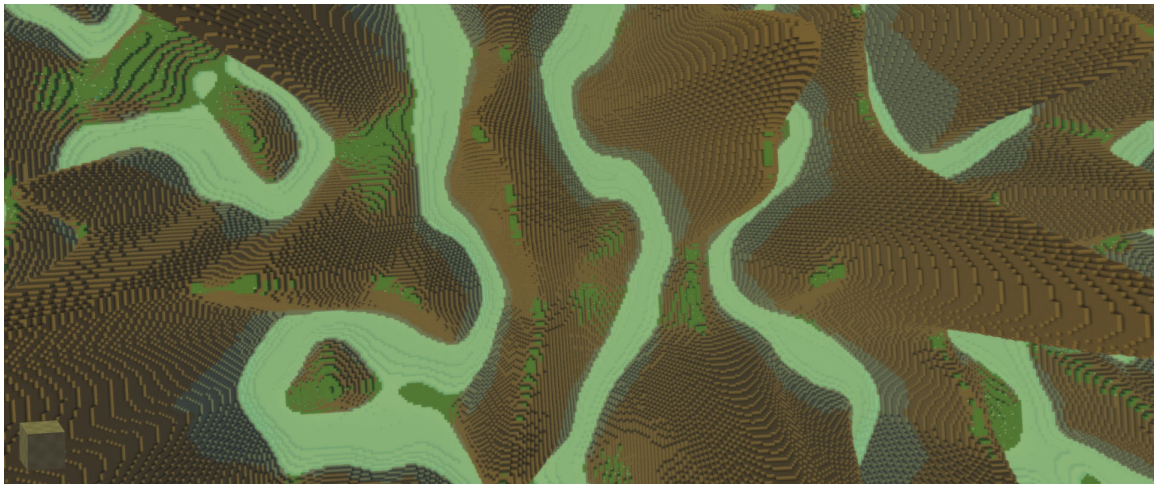
Poušť taktéž využívá kombinaci Perlinova šumu a Voroného diagramů; v tomto případě zde Voroného diagramy modelují duny. Je zde počítáno maximum ze dvou Voroného diagramů (o stejné velikosti oktávy), aby se omezila zřetelnost vzoru, který při využití Voroného diagramů vzniká.



Obrázek 4.11: Duny generované jedním (vlevo) a dvěma (vpravo) Voroného diagramy (zmenšené měřítko)

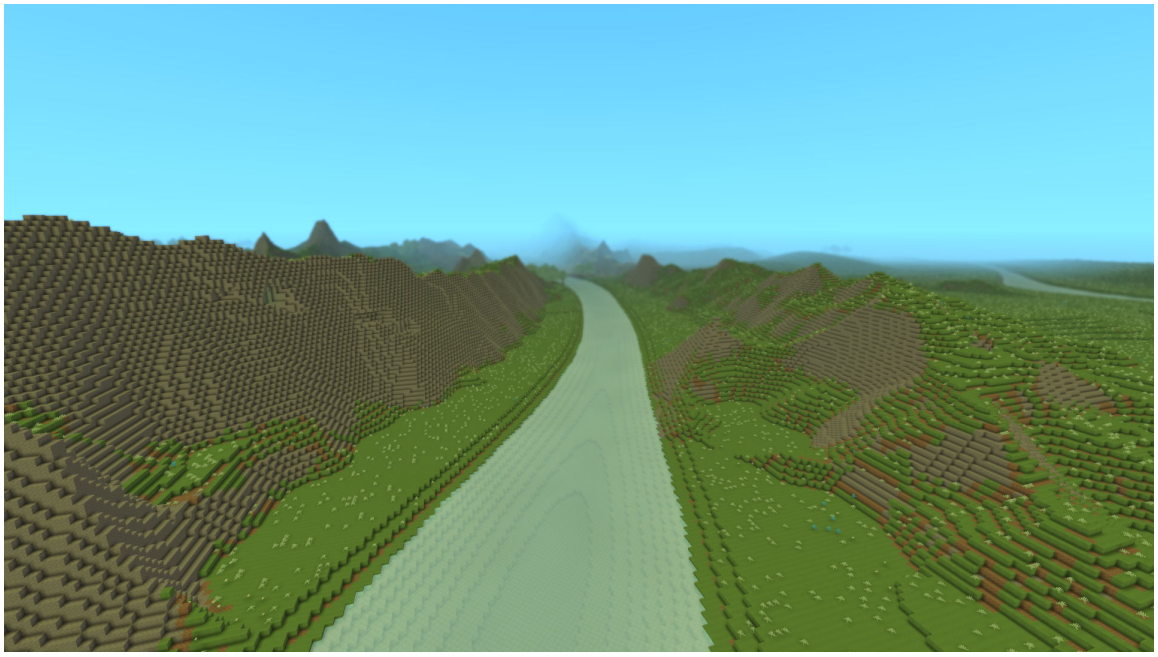
4.4.3 Řeky

Řeky jsou generovány pomocí Perlinova šumu. Absolutní hodnota Perlinova šumu generuje vzor s ostrými minimy v nule. Řeky jsou generovány právě v místě těchto minim.



Obrázek 4.12: Vizualizace absolutní hodnoty Perlinova šumu. Řeky jsou generovány prahováním této hodnoty.

Všechny řeky jsou generovány na úrovni mořské hladiny. Výška okolní krajiny je v okolí řek postupně snižována (na základě výstupu šumu, pomocí kterého byly řeky generovány).



Obrázek 4.13: Snižování nadmořské výšky krajiny okolo řek, aby jejich hladina mohla být na úrovni moře

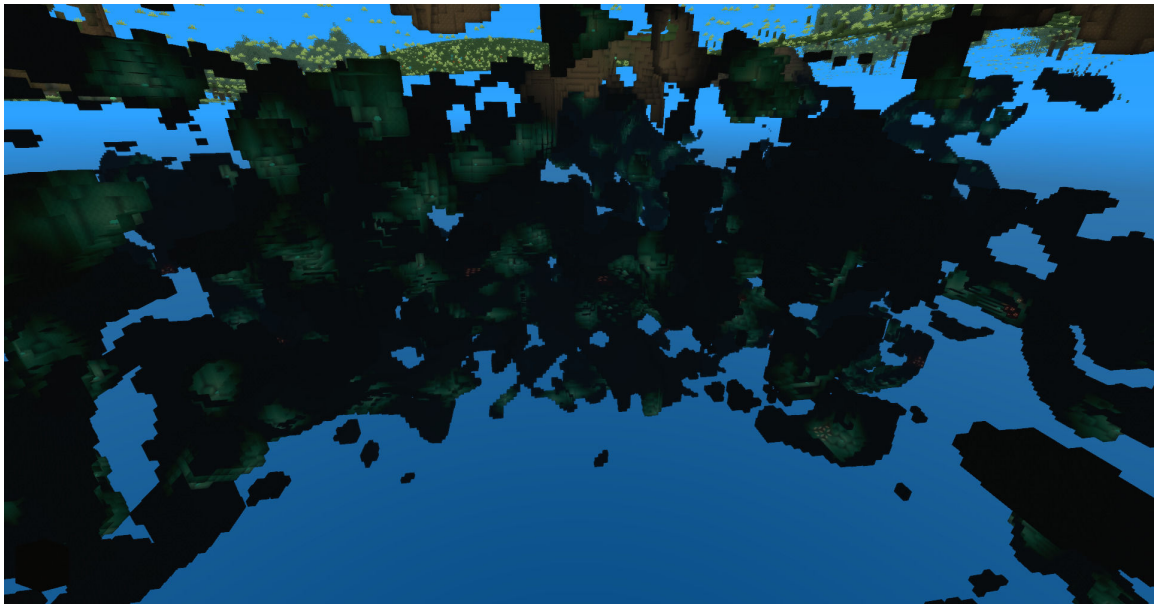
4.4.4 Stromy

Stromy jsou z pohledu matematického popisu relativně jednoduché. Každému stromu je náhodně určena výška v určitém intervalu. Velikost koruny je pak odvozena od velikosti výšky, listy jsou generovány s pravděpodobností snižující se se vzdáleností od kmene (*manhattan distance*). Rozmístění stromů je dáno rozmístěním bodů ve Voroného diagramu; terén je rozdělen do mřížky, v každé oblasti mřížky je 1 až N bodů.

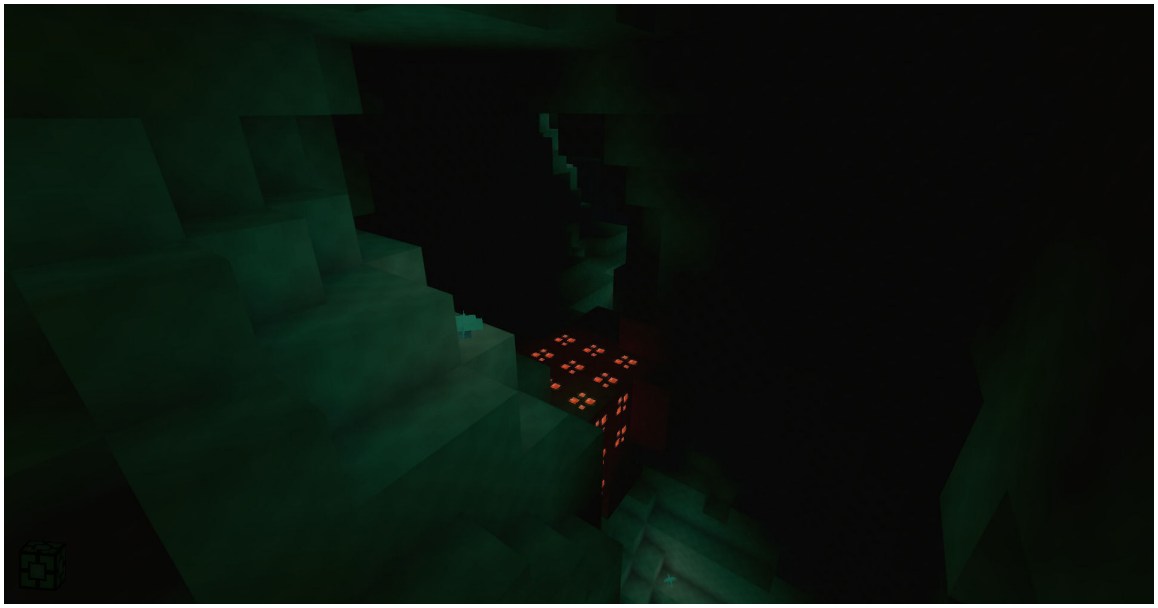
Obrázek 4.14: Les

4.4.5 Jeskyně

Při generování jeskyní se používají dva 3D Perlinovy šumy; jeden definuje oblasti, kde se jeskyně vyskytují (větší oktávy), druhý dává podobu samotným jeskyním (menší oktávy). Oba šumy jsou prahovány a vzájemně v konjunkci (jeskynní prostor je vytvořen, pokud jsou hodnoty obou šumů pro daný voxel nad určitým prahem). Aby vstupy do jeskyní nebyly příliš časté a rozsáhlé, jsou jeskyně těsně pod povrchem v některých místech potlačovány pomocí 2D Perlinova šumu.



Obrázek 4.15: Generovaný jeskynní komplex



Obrázek 4.16: Pohled uvnitř jeskyně

Kapitola 5

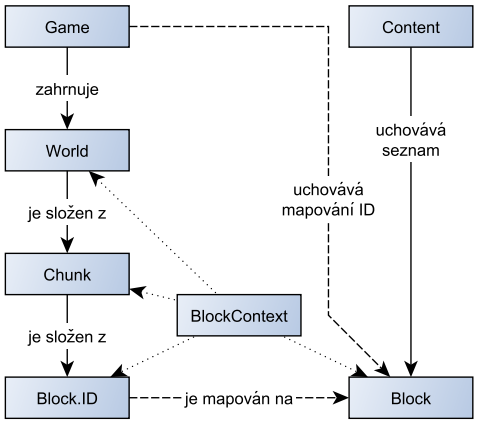
Implementace

Jako součást této práce byla vytvořena aplikace implementující metody zmíněné v tomto dokumentu. Zdrojový kód aplikace je v jazyce D. Jako grafické API pro práci s GPU bylo zvoleno OpenGL ve verzi 4.6 (knihovna `bindbc-opengl`). Z této verze bylo využito anizotropní filtrování (`ARB_texture_filter_anisotropic`) a `gl_DrawID` ve vertex shaderu (`ARB_shader_draw_parameters`). Z modernějších použitých technologií lze zmínit také například *compute shaders* (GL 4.3), *bindless* textury (není *core feature*) nebo *direct state access* (GL 4.5).

Pro načítání obrázků, správu okna a uživatelských vstupů a vykreslování textu byla použita knihovna `derelict-sfml2`. Jako rozhraní pro SQLite (který byl zvolen jako formát pro ukládání terénu na disk) byla zvolena knihovna `d2sqlite3`. Grafika pro projekt byla přejata z „C-tetra texture pack“[1], který je publikován pod licencí CC BY-NC 4.0[2] (volně použitelné pro nekomerční projekty, nutné zmínit autora).

5.1 Reprezentace světa

Třídy vztahující se k reprezentaci světa jsou znázorněny na následujícím diagramu:



Obrázek 5.1: Diagram popisující vztahy mezi třídami souvisejícími s reprezentací světa

[1] https://www.planetminecraft.com/texture_pack/16x-c-tetra-1-13/
[2] https://creativecommons.org/licenses/by-nc/4.0/

Třída `Block` reprezentuje různé typy voxelu. Její metody definují veškeré vlastnosti voxelu – jeho vzhled, průhlednost, světelnou emisi, jeho kolizní model apod. Nový typ voxelu lze vytvořit definováním nové třídy dědící třídu `Block`, alternativně jej lze sestavit z různých komponent (komponenty definující vzhled, komponenty definující kolize, ...) s využitím tříd `ComponentBlock` a `BlockComponent_XXX`. Třída `Content` potom uchovává databázi všech registrovaných bloků.

Třída `Game` reprezentuje *hru*. Návrh počítá s možností mít více světů v jedné hře, implementace ale podporuje pouze jeden. Třída `Game` spravuje databázový soubor ke hře přiřazený a uchovává mapování dvoubajtového *block ID* (viz sekce 4.1) na instance třídy `Block`.

Třída `World` pak reprezentuje jeden svět. To zahrnuje mj. načítání a odkládání jednotlivých chunků do/z paměti a přístup k nim.

Třída `Chunk` reprezentuje jeden chunk (termín *chunk* je popsán v sekci 4.1). Obsahuje pole ID bloků, které chunk tvoří.

Třída `BlockContext` slouží k reprezentaci libovolného voxelu ve světě. Používá se jako „ukazatel“ na daný blok; obsahuje informace o tom, v jakém světě a na jaké pozici se voxel nachází, referenci na `Block` dle typu voxelu apod. Tato třída je předávána jako parametr u metod ve třídě `Block`, například metoda volaná pro vykreslení bloku je

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

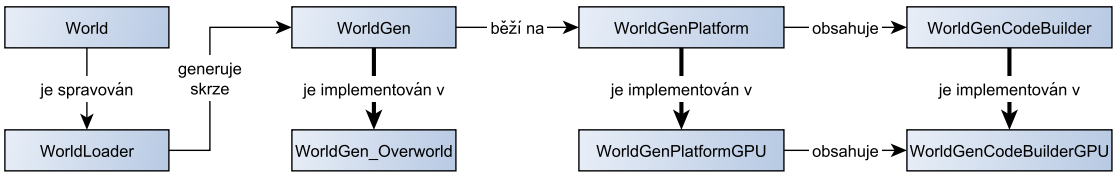
Příkladem uvedeme kód pro vytvoření bloku (kamene) na pozici $(0, 10, 0)$:

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

Pro reprezentaci pozice bloku ve světě je použita struktura `WorldVec`, což je 3D vektor s komponentou typu `int` (čtyři bajty na jakékoli platformě).

5.2 Generování a ukládání světa

Subsystém generování a ukládání světa je založen na těchto třídách:



Obrázek 5.2: Diagram popisující vztahy mezi třídami, které souvisí s generováním a ukládáním světa

Třída `World` spravuje pouze chunky, které jsou načtené v paměti. Jakmile nějaký požadovaný chunk v paměti není, je předán třídě `WorldLoader` požadavek na jeho načtení. Třída `WorldLoader` běží na dedikovaném vlákně a žádaný chunk, pokud již byl generován, načte z SQLite databáze, v opačném případě zajistí jeho vygenerování. Chunky, které již není žádoucí uchovávat v paměti, jsou taktéž předány této třídě a ta zajistí jejich uložení do databáze.

Systém je navržen tak, že na chunky, které mají být načtené v paměti, je třeba se periodicky dotazovat. *Chunky*, na které nebyl během pěti sekund žádný dotaz, jsou pře-

dány k odložení. Pro tento mechanismus se využívají funkce `World.maybeChunkAt(WorldVec pos)`, `World.maybeLoadChunkAt(WorldVec pos)` a `World.chunkAt(WorldVec pos)`. První zmíněná funkce navrátí chunk na zadané pozici pouze v případě, že je načtený v paměti (jinak navrátí `null`). Druhá funkce se chová stejně jako první, ale nenačtený chunk zadá k načtení (které neproběhne hned, ale někdy v budoucnu) a u načteného chunku vynuluje časovač odložení. Funkce `chunkAt` počká, dokud se chunk nenačte do paměti, a nikdy nenavrací `null`. *Chunky*, které jsou načtené v paměti, nazýváme *aktivní*.

Požadavky na vygenerování nového chunku jsou předány třídě `WorldGen`. Ta je určena pro *subclassing*, kde jednotlivé podtřídy mají definovat různé generátory terénu. V této práci je definována pouze třída `WorldGen_Overworld`, která generuje svět připomínající Zemi. Systém generování je navržen tak, aby mohl stejný generátor běžet na různých platformách. V této práci bylo implementováno pouze generování akcelerované na GPU (`WorldGenPlatformGPU` a `WorldGenCodeBuilderGPU`). Tento návrh byl proveden s ohledem na to, aby do budoucna mohl být vytvořen *multiplayer* a aby mohl existovat dedikovaný server běžící na hardware, který nemusí podporovat GPU akceleraci.

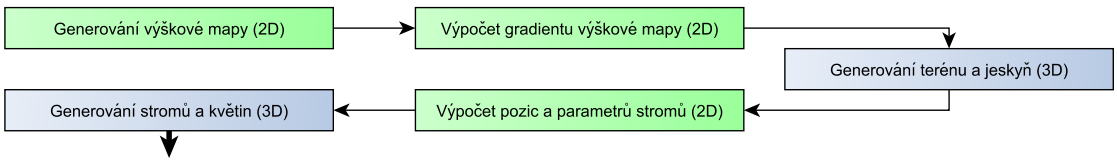
Generování světa je realizováno více 2D i 3D průchody, které mohou být kombinovány v libovolném pořadí. Průchody mají funkcionální charakter – průchod se počítá pro každý pixel (pro 3D průchody voxel) nezávisle, v rámci průchodu lze číst data z předchozích průchodů, výsledky jsou pak dostupné v následujících průchodech. Jednotlivé průchody zastupuje třída `WorldGenCodeBuilder`, která současně poskytuje rozhraní pro definici jejich chování. Ačkoli tedy generování běží na GPU v *compute shaderech*, jeho programování probíhá pomocí kódu v jazyce D. Nevýhodou tohoto přístupu je, že jelikož jazyk D nepodporuje přetěžování některých operátorů s dostatečnou možností přizpůsobení (konkrétně operátory porovnávání, přiřazení nebo logické `&&` a `||`), je komfort při psaní kódu o něco nižší, než kdyby se pracovalo přímo s compute shadery. Výhodou tohoto přístupu je jeho nezávislost na cílové platformě a také to, že struktury pro předávání dat mezi jednotlivými průchody mohou být automaticky generovány a integrace průchodů je tedy velmi jednoduchá.

```
with (platform.add2DPass()) {
  auto seaLevel = c(seaLevelVal);
  auto mountainess = clamp01(perlin2D(256, [c(0.25), c(0.5), c(1)]).x + 0.2);
  auto elevationZ = 64 * clamp01(0.2 + perlin2D(baseOctave, coefs).x);

  // Big mountain peaks
  auto bigPeakVoronoi = voronoi2D(1024, 8);
  auto bigPeakVal = pow((bigPeakVoronoi.x - 0.05) * 5, c(2)) * mountainess;

  // Small mountain peaks
  auto smallPeakVal = pow(voronoi2D(64, 4).x * 3, c(4));

  // Lil' perlin to smooth things out
  auto smoothPerlin = perlin2D(16, [c(0.3), c(0.2), c(0.1), c(0.1)]).x;

  auto mountainsZ = max(c(0), bigPeakVal * 128 + smallPeakVal * 32 +
     smoothPerlin * 32 - 16) * mountainess;

  set2DData("groundZ", seaLevel + elevationZ + mountainsZ);
  finish();
}
```

Kód 5.1: Příklad programování generátoru světa; jedná se o výňatek z prvního průchodu (2D) generátoru `WorldGen_Overworld`

Funkce pro Perlinův šum a Voroného diagramy na GPU využívají vláknové kooperace. Detailnější specifikace algoritmů však rozsahově nezapadá do této práce.

Generátor `WorldGen_Overworld` pracuje s pěti průchody (2×2D a 3×2D). Jejich funkce je popsána v následujícím diagramu.



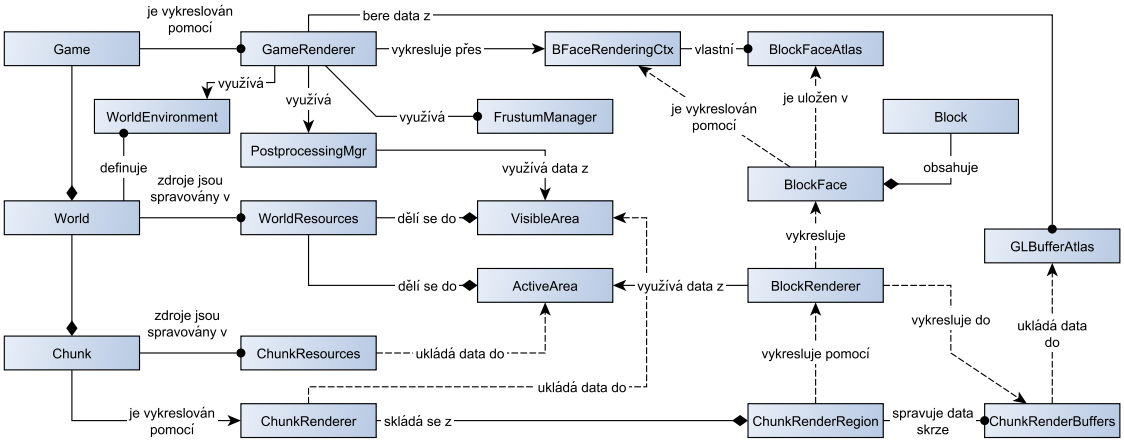
Obrázek 5.3: Diagram průchodů generátoru `WorldGen_Overworld`

Vegetace je generována v odděleném 3D průchodu, protože je třeba zajistit, aby se negenerovala nad vstupy do jeskyní, kde terén neodpovídá výškové mapě. Proto má první 3D průchod 2D výstup, který udává, zda je na dané x, y pozici vstup do jeskyně a tedy se v daném místě nemá generovat vegetace. 3D průchody mohou mít 2D výstup, ale v celém sloupci do něj smí zapsat pouze jediná invokace; v opačném případě není výsledek definován. V případě jeskyní zapisuje do 2D výstupu invokace odpovídající voxelu na úrovni výškové mapy.

Při generování každého chunku je interně generována oblast 3×3 chunků s výstupním chunkem uprostřed: tato redundance je třeba například pro situace, kdy se mají generovat listy ke stromu, jehož kmen je ve vedlejším chunku, a kde je třeba vědět, zda generování stromu není potlačeno kvůli vstupu do jeskyně. Při generování opět vyvstává omezení lokality dané volbou velikosti chunku, jak bylo zmíněno v oddíle 4.1.

5.3 Vykreslování světa

S vykreslováním světa souvisí tyto třídy:



Obrázek 5.4: Diagram tříd souvisejících s vykreslováním světa

5.3.1 *Block faces*

Stěny voxelů jsou vykreslovány standardní vykreslovací *pipeline* jako trojúhelníky. Voxely v aplikaci nemusí mít vzhled krychle, mohou mít libovolný tvar.

Obrázek 5.5: Různé tvary voxelů

Veškeré vykreslování je nicméně založeno na otexturovaných obdélnících (dále v tomto textu nazývané *block faces*). Různé varianty těchto *block faces* reprezentují instance třídy `BlockFace`. Ta definuje nejen texturu, ale i další vlastnosti týkající se vykreslování:

- Práce s *alpha* kanálem:
  - Kanál může udávat průhlednost, která se prahuje (*alpha testing*). Tato metoda je rychlá na vykreslování, ale zamezuje použití *mipmappingu*. V aplikaci je použita pro vykreslování květin a listů stromů.
  - Kanál může udávat průhlednost (*alpha blending*). V tom případě je *face* vykreslován pomalejšími mechanismy podporující průhlednost (*depth peeling*).
  - Kanál může udávat míru vlastní emise daného pixelu (*glow map*).
- *Cull facing*: některé *faces* (například vegetace) se vykreslují z obou stran, stěny neprůhledných krychlí stačí vykreslovat pouze směrem ven.

- *Wrapping* udávající, zda má být k textuře přistupováno jako k opakujícímu se vzorku (tato volba se projevuje na vyšších úrovních *mipmap*).
- Animace vrcholů. *Face* může „povlávat“ ve větru, buď ve všech osmi rozích (listy stromů), nebo pouze ve vrchní části (květiny). Dále je zde možné zvolit animaci pro hladinu kapalin. Tento jev je implementován na úrovni *vertex shaderu*.
- Rozlišení textury.

Faces se stejnou konfigurací jsou agregovány do celku reprezentovaného třídou `BlockFaceRenderingContext` (dále označován jako *face context*). Ten uchovává informace o konfiguraci (která je jednotná pro všechny *faces* jím spravované) a *shadery* použité při vykreslování. Všechny *shadery* jsou sestaveny z kódu uloženého v souborech `res/shader/render/blockRender.(vs|fs).glsl`, jejich chování se upravuje vložením různých `#define` direktiv. Každý *face context* obsahuje více variant *shaderu*: pro standardní vykreslování, pro vykreslování do *shadow mapy* a pro vykreslování pro *depth peeling* (kde je navíc implementován *near depth test*).

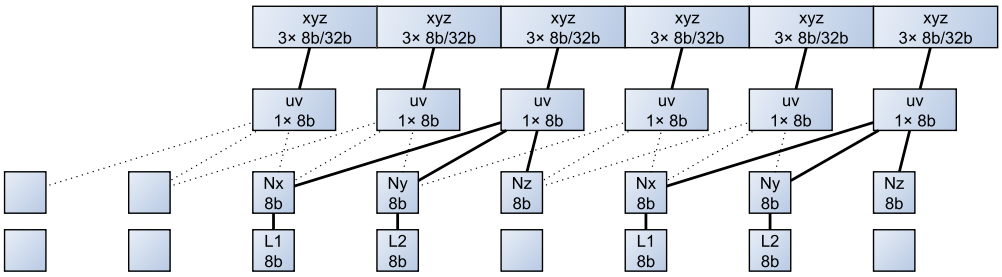
Jednotlivé *faces* jsou uloženy v atlasu textur (třída `BlockFaceAtlas`); každému *face kontextu* náleží vlastní atlas. V textuře, kterou atlas spravuje, jsou *faces uloženy* v poli (`GL_TEXTURE_2D_ARRAY`).

5.3.2 Reprezentace dat pro vykreslování

Každý chunk je vertikálně rozdělen do několika regionů; ty jsou reprezentovány třídou `ChunkRenderRegion`. Tyto regiony jsou pak nejmenší jednotkou vykreslování (region se vykresluje celý, nebo vůbec). Pro každý *face context* region uchovává vlastní sadu *bufferů* obsahující data pro vykreslování. Tato data jsou aktualizována pouze v případě, kdy je to potřeba – při prvním zobrazení chunku nebo pokud hráč provede zásah do terénu.

Data jsou na GPU uložena v jednom velkém bufferu, který spravuje třída `GLBufferAtlas`. Je implementován vlastní systém pro správu paměti; ten neumožňuje realokaci (ta ani není potřeba, data jsou nejprve sestavena na CPU a až potom jsou nahrána na GPU) a alokace zabírá vždy nejmenší volný region paměti, který je současně dostatečně veliký.

Pro každý vrchol se uchovávají tři atributy: pozice, UV texturovací souřadnice a normála. Protože normála je pro všechny tři vrcholy trojúhelníku stejná, je zavedena optimalizace, díky které lze normálu uchovávat pouze jednou pro celý trojúhelník: buffer s normálou je připojen třikrát, jednou jako atribut N_x, podruhé jako N_y a potřetí jako N_z. Střída je pro všechny tři připojení nula, tedy údaje ve všech třech atributech postupně prochází x, y i z komponenty normály. *Offset* je pro každé napojení upraven tak, aby vždy třetí vrchol trojúhelníku (resp. *provoking vertex*, který je implicitně nastaven na `GL_LAST_VERTEX_CONVENTION`) obsahoval správná data pro všechny tři komponenty. Atributy komponent normály jsou označeny jako `flat`, *fragment shader* tedy pro všechny tři vrcholy dostává data z *provoking* vrcholu. Nevýhodou tohoto přístupu je, že platná data normál jsou ve *vertex shaderu* dostupná pouze v jednom vrcholu trojúhelníku, a proto ve *vertex shaderu* nelze dělat *per-vertex* operace s normálami (např. posun normál při animacích bloků, které jsou realizovány ve *vertex shaderu*).



Obrázek 5.6: Struktura *vertex arrays* předávaných *vertex shaderu*

Aplikace uchovává data o velkém množství vrcholů (řádově až desítky milionů vrcholů při maximální dohledové vzdálenosti – v testované scéně bylo při dohledové vzdálenosti 32 chunků naměřeno 24 milionů vrcholů), proto je žádoucí maximální optimalizace paměti. Normály jsou proto ukládány jako vektor osmibitových celých čísel. Protože jsou *block faces* uloženy v poli textur, x a y souřadnice budou vždy buď 0, nebo 1, a lze je obě uložit do jednoho bajtu. Souřadnice z udávající číslo vrstvy je pro všechny vrcholy *face* stejná, takže je uložena stejným způsobem, jako data normály (jeden bajt na každý vrchol, data se poté spojí pro zvýšení rozsahu hodnot a předají se *fragment shaderu* jako *flat* parametr).

Komponenty souřadnic vrcholů jsou taktéž uloženy v jednom bajtu; pozice regionu ve světe je předávána dodatečně jako konstanta pro celý *buffer*. Díky tomu musí být chunk vertikálně rozdělen na alespoň dva regiony, protože by jinak nešlo reprezentovat pozice horních vrcholů voxelů nejvýše v chunku ($z = 255$ by reprezentovalo spodní hrany nejsvrchnější vrstvy, pro vrchní vrcholy by již došlo k přetečení).

Protože ne všechny typy bloků mají vykreslované *faces* zarovnané do voxelové mřížky (např. kaktus a obilí; květiny mají sice mají atypický tvar, ale souřadnice jsou zarovnané do mřížky, trojúhelníky jdou křížem přes voxel, jak je demonstrováno na obrázku 6.5), byly zavedeny dvě sady bufferů: v jedné sadě jsou souřadnice ukládány v 3× 1B, ve druhé sadě jsou souřadnice uloženy ve 3× 32 bitech s plovoucí čárkou (*float*). Výskyt těchto bloků ve scéně je značně nižší, takže kapacita této sady bufferů může být mnohem menší.

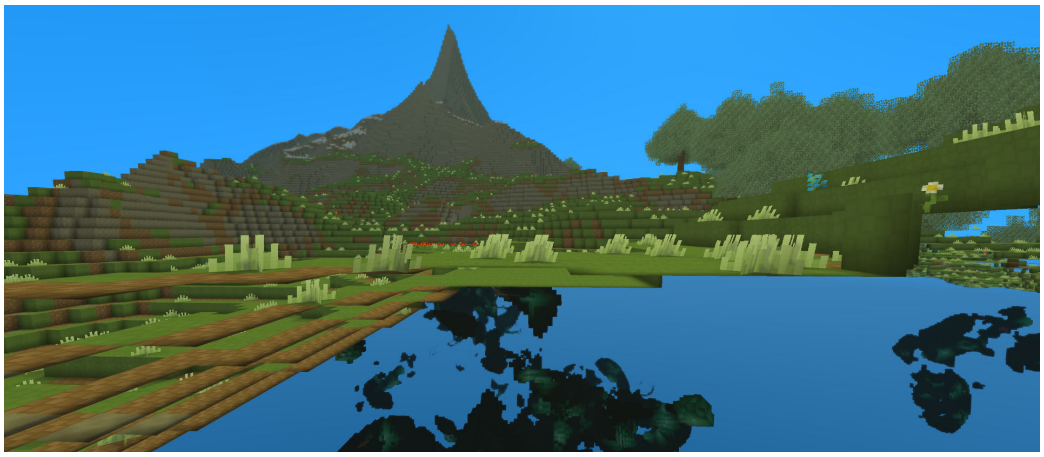
Obrázek 5.7: Bloky kaktus a obilí, jejichž *faces* nejsou zarovnané do voxelové mřížky

Paměťové nároky na jeden vrchol jsou tedy 3×1 + 2×1 + 1 = 6 bajtů při 8b souřadnicích a 3×4 + 2×1 + 1 = 15 bajtů při 32b souřadnicích. Při testované scéně, která obsahuje 24 milionu 6B vrcholů a 100 tisíc 15B vrcholů, činí paměťové nároky na VRAM 145 MB. Pokud by všechny vrcholy byly ukládány ve 32 bitech, zvýšily by se nároky na 360 MB. Nejedná se o pro moderní GPU nijak dramatická čísla, nicméně buffery nejsou jediné paměťově náročné zdroje, které aplikace využívá. Menší velikost složek navíc snižuje nároky na datovou propustnost, která bývá nejběžnějším úzkým hrdlem při práci s GPU.

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.



Obrázek 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 výpočtům 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*.

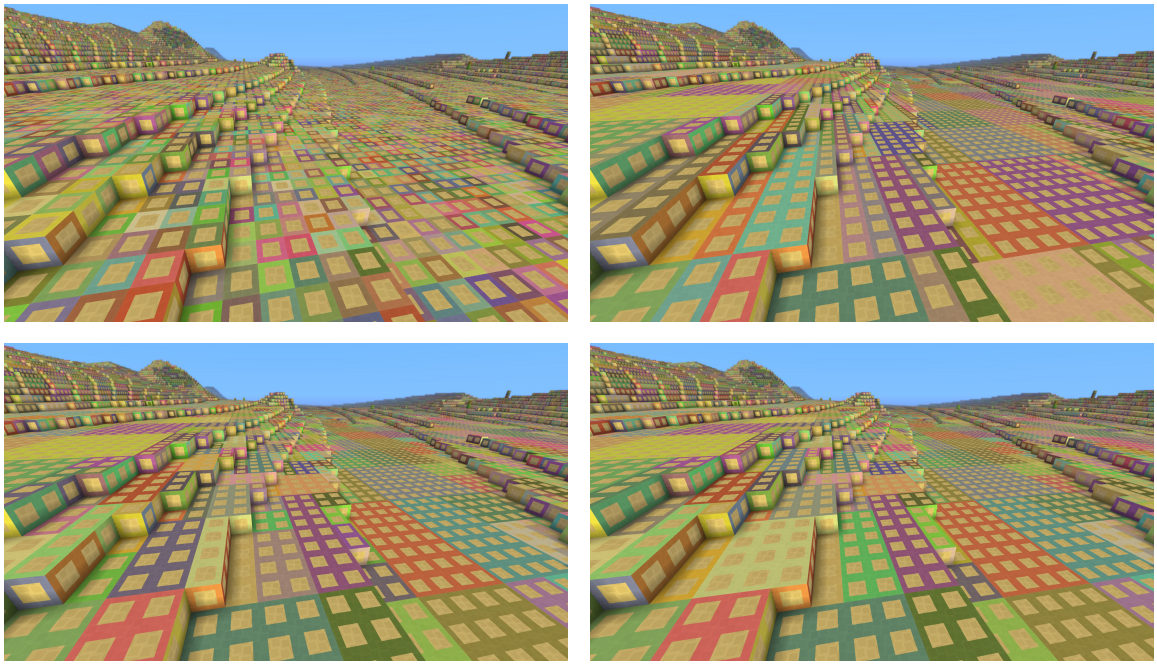
Všechny výše zmíněné optimalizace jsou akcelerovány na GPU. Z toho důvodu je v GPU uchovávána textura 3D textura s *block IDs* (jedná se o 1 : 1 kopii dat z CPU) a buffer s informacemi o vlastnostech jednotlivých typů bloků. Data *block IDs* jsou agregována do 3D textur o velikosti 32×32×256 bloků (tedy 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 prázdné 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 agregace je 8×8 stěn, což je dáno velikostí pracovní skupiny *compute shaderu* (8×8×8) a tím, že uv souřadnice jsou uloženy ve 2× 4 bitech (rozsah přípustných hodnot je 0–15, pro 16× agregaci by byl třeba rozsah 0–16). Vizualizaci agregace lze v aplikaci aktivovat nastavením vizualizovaných dat (první *combobox* shora) na „Aggregation“. Byly implementovány tři metody agregace (lze mezi nimi přepínat 3. *comboboxem* zespod v menu):

- **`Lines`** agreguje nejprve do maximální míry v jednom směru (2 směry agregace v rovině stěny) a poté spojuje obdélníky stejné délky ve směru druhém.
- **`Squares`** agreguje 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.

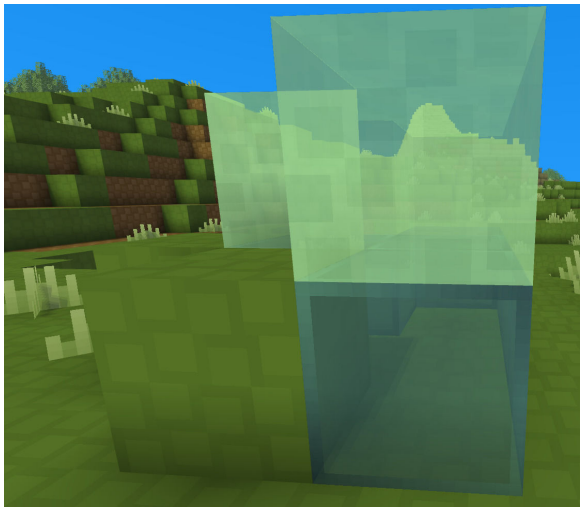
Porovnání efektivity jednotlivých metod je realizováno v oddílu 6.2.3. Problémem agregace je, že vytváří tzv. *T-junctions* (tedy situace, kdy na sebe trojúhelníky nenavazují hranami, ale vrchol jednoho trojúhelníku se nachází na hraně druhého trojúhelníku), které způsobují artefakty v podobě průhledných 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.



Obrázek 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.

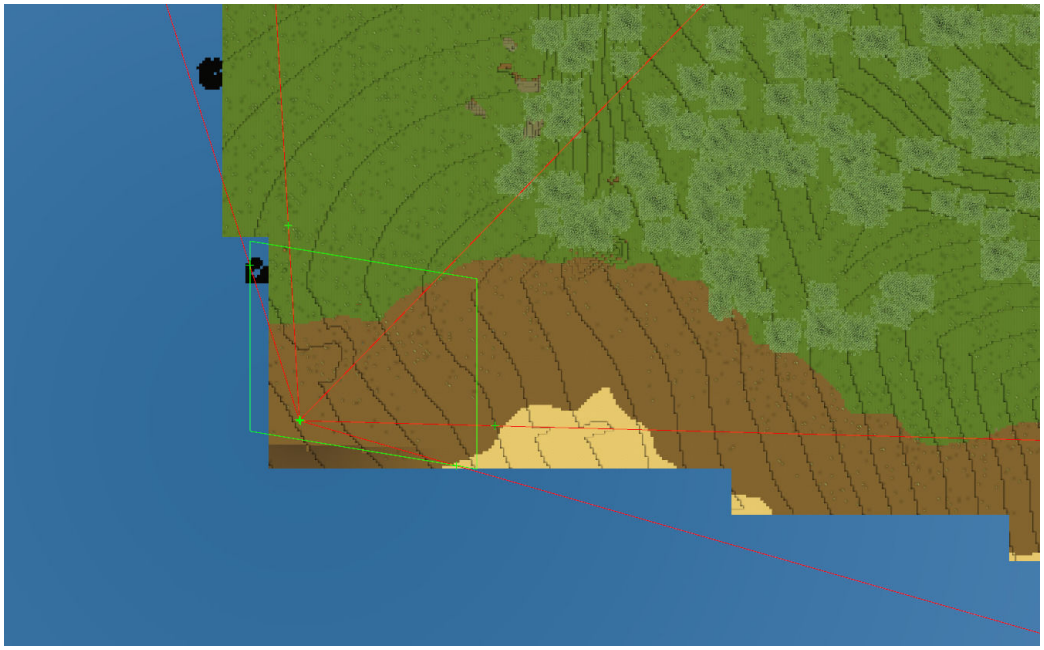


Obrázek 5.10: Průhledné bloky a jejich interakce s bloky neprůhlednými

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 matice) a následným porovnáním vůči 1 nebo -1; pakliže je komponenta w při projekci menší než nula, jsou výsledky porovnání bočních okrajů invertovány.

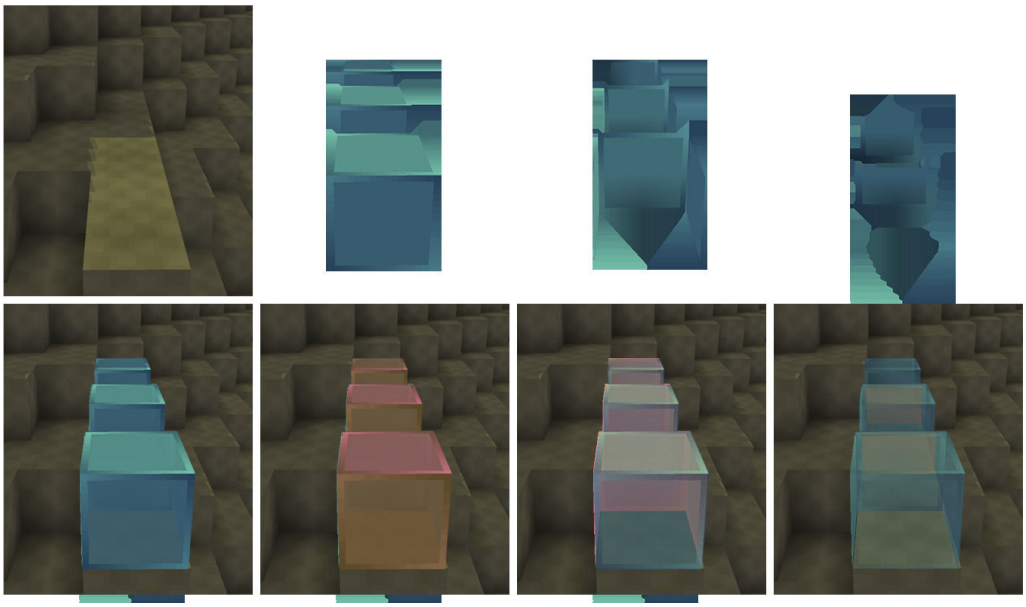


Obrázek 5.11: Demonstrace *frustum cullingu*; červené čáry v horním obrázku označují hrany a střed projekčního jehlanu. Zelený čtyřúhelník označuje oblast pokrytou *shadow mapou* (pouze orientační). V aplikaci lze do pohledu shora přepnout klávesou F3.

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.



Obrázek 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 128×128×256 voxelů, každá tato oblast odpovídá jedné 3D textuře a je reprezentována třídou `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).

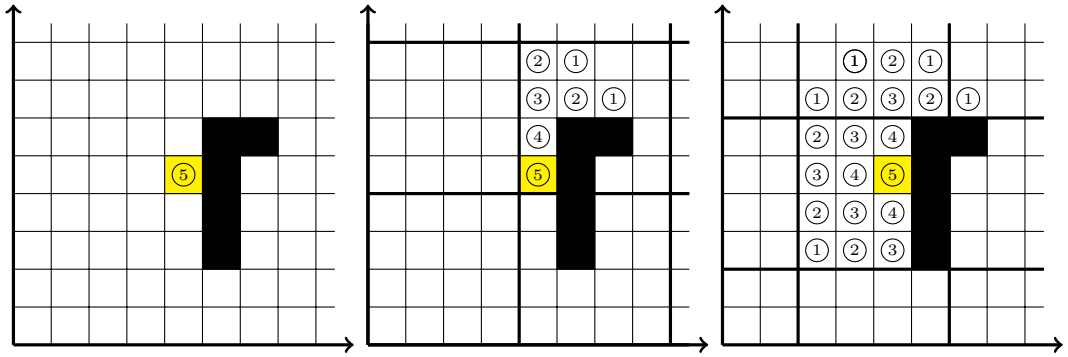
Do 4 bitů lze uložit celkem 16 různých úrovní osvětlení (0–15), což souhlasí s omezením dosahu jevů danou velikostí chunku, který má v horizontální rovině rozměr 16 voxelů. Zvýšení maximálního možného dosahu světla by tedy vyžadovalo zvětšení velikosti chunku, a kromě toho také přechod z `GL_R16UI` na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

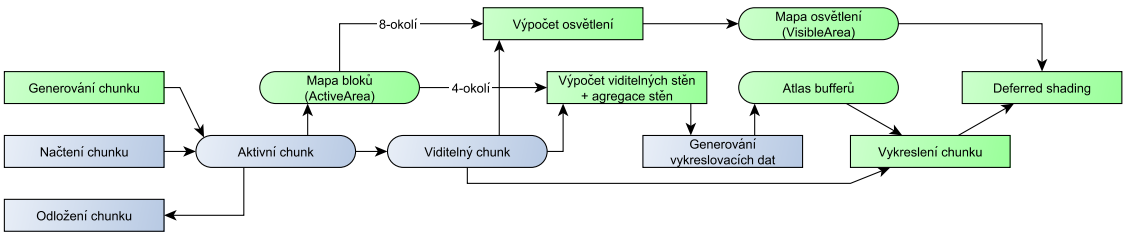
Výpočet dat osvětlení také probíhá na GPU; celulární algoritmus popsaný v sekci 4.3 je vhodný pro akceleraci na grafickém koprocesoru. Pro výpočet osvětlení se využívají i *block IDs* sousedících chunků (8-okolí). Výpočet probíhá v několika krocích (jeden krok odpovídá jednomu spuštění *kernelu*):

- 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 8×8×8 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 automat. 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.



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

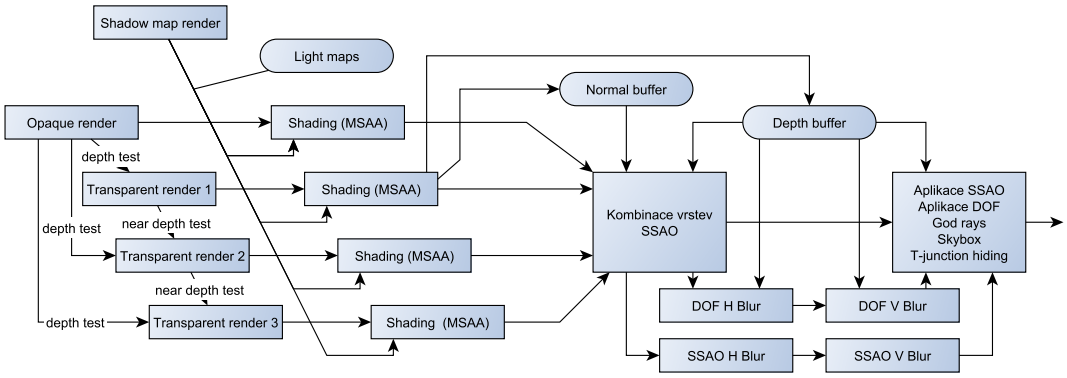
Souhrnně jsou tedy k vykreslování chunků třeba tyto kroky:



Obrázek 5.14: Souhrn procesů nutných k vykreslování. Zeleně jsou označeny prostředky a procesy vázané na GPU, modře na CPU.

5.4 *Postprocessing*

Postprocessing se skládá z následujících průchodů:



Obrázek 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 promítnou 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 128×128×256 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.

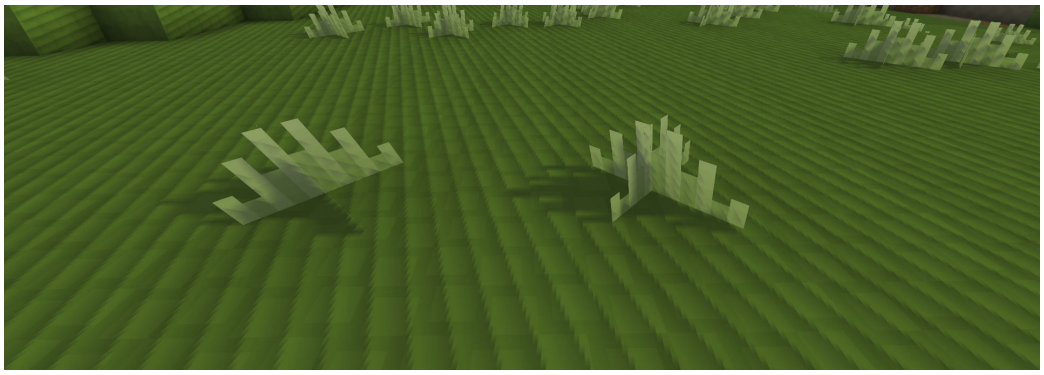
V aplikaci je implicitně aktivován 2× *multisampling*; lokální průchody v důsledku toho pracují s *multisample* texturami. Výstupem lokálních průchodů jsou již jednovrstvé textury se stínovaným obrazem. Z prvního *depth peeling průchodu* (tedy pro objekty nejblíž ke kameře) je pro další *postprocessing* 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).

Obrázek 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.



Obrázek 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.

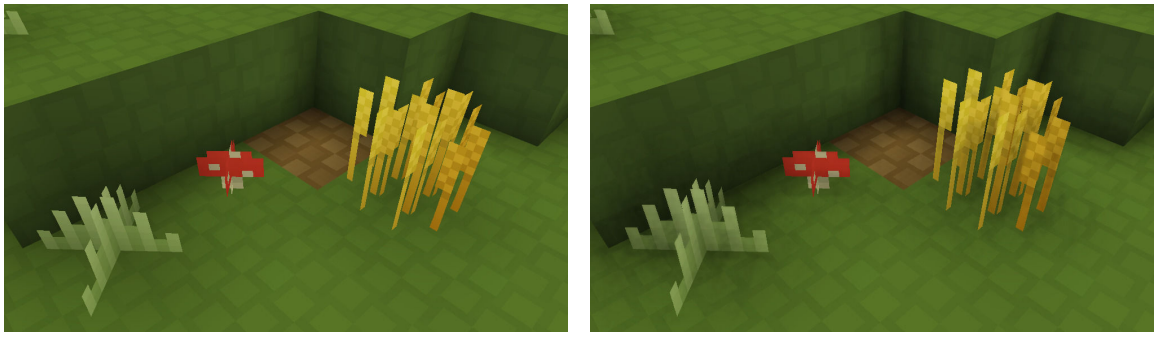
Shadow mapping vyžaduje dodatečné vykreslení scény ortogonální maticí „z pohledu slunce“. Při vykreslování do stínové mapy je *face culling* zapnutý pro všechny kontexty, aby se předešlo např. prosvítání slunečního světla skrze stěny jeskyní. Pro kontexty, které

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 výpočtům *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.

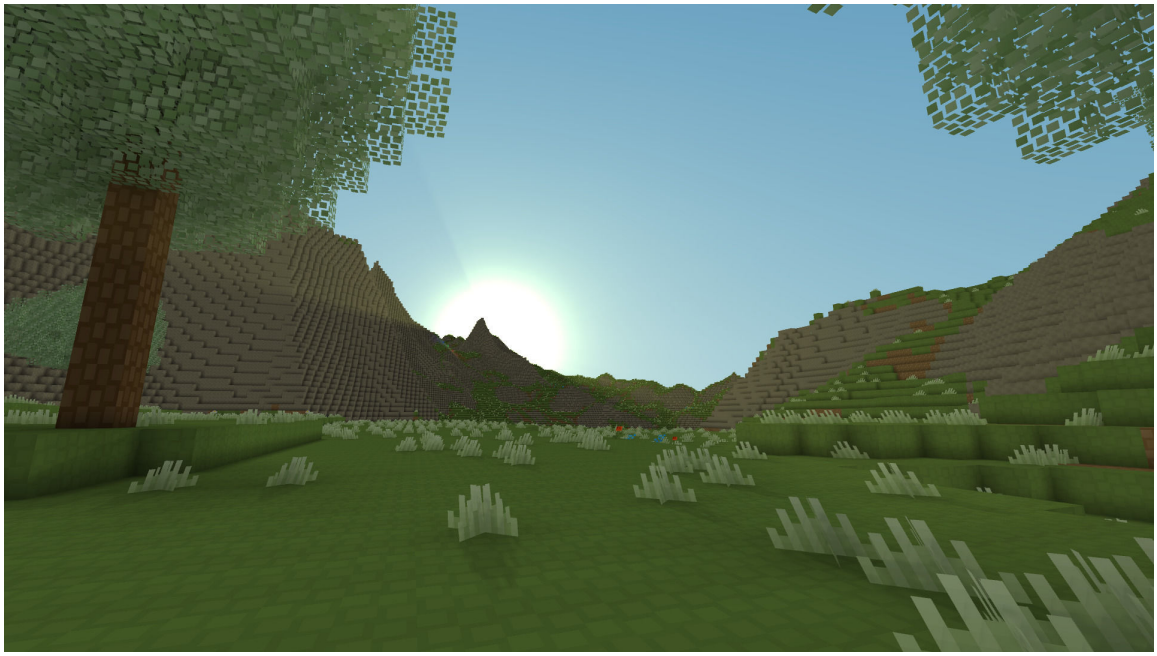
SSAO pro každý pixel sbírá 2 až 8 vzorků hloubky (dle vzdálenosti pixelu od kamery) v polokouli dané vzdáleností a normálou pixelu. Vzorky jsou umístěné do 2D spirály vůči normále, hloubka je pak daná uniformním rozložením. Na efekt *SSAO* je aplikováno 9×9 Gaussovo rozostření se $\sigma = 3$.



Obrázek 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.



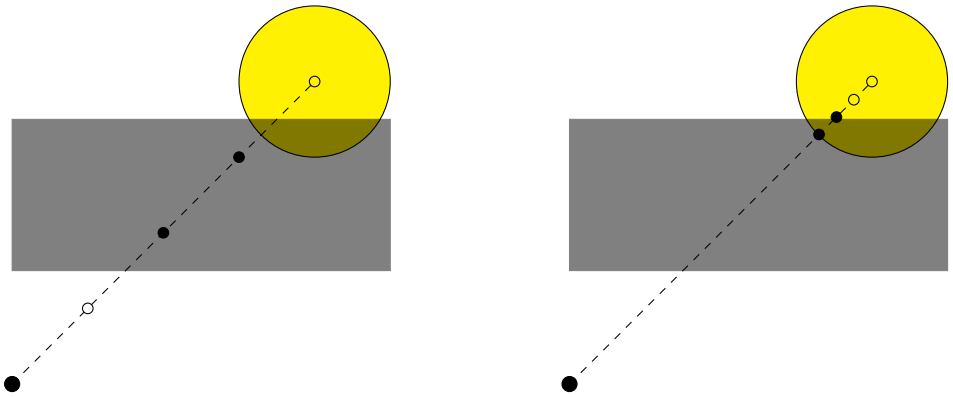
Obrázek 5.19: Scéna s deaktivovaným *depth of field* a atmosférickým efektem

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

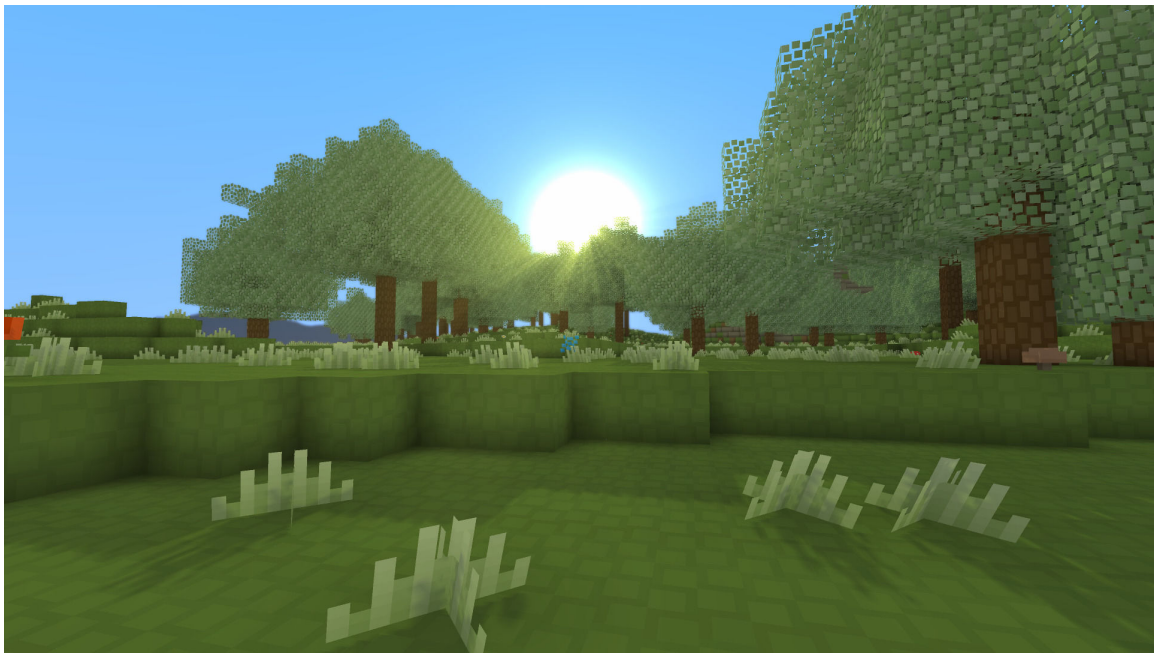
5.4.4 *God rays*

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 „emisivitu“ 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.

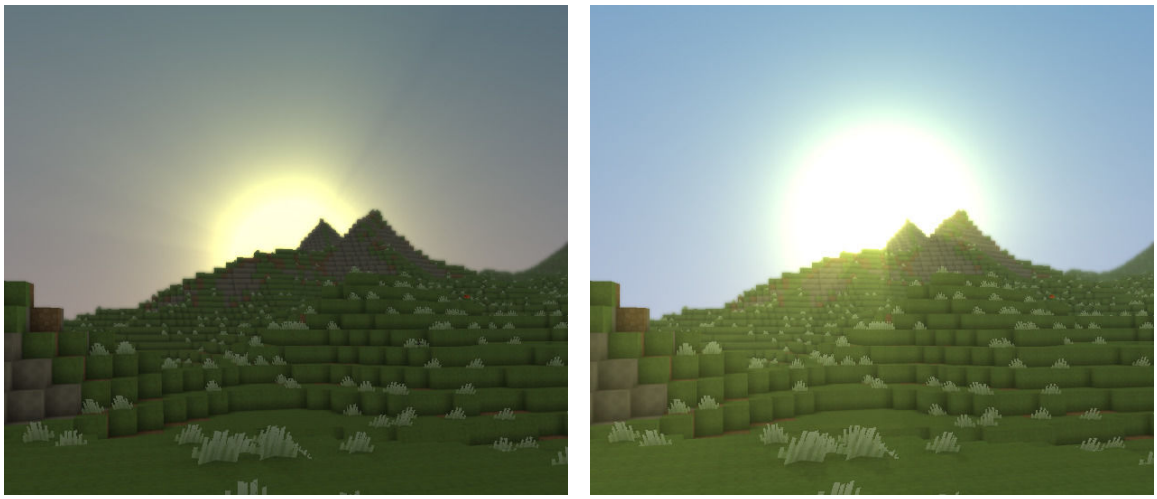


Obrázek 5.21: Původní navržené vzorkování (vlevo) vs. implementované vzorkování (vpravo) v efektu *god rays*



Obrázek 5.22: *God rays* v aplikaci

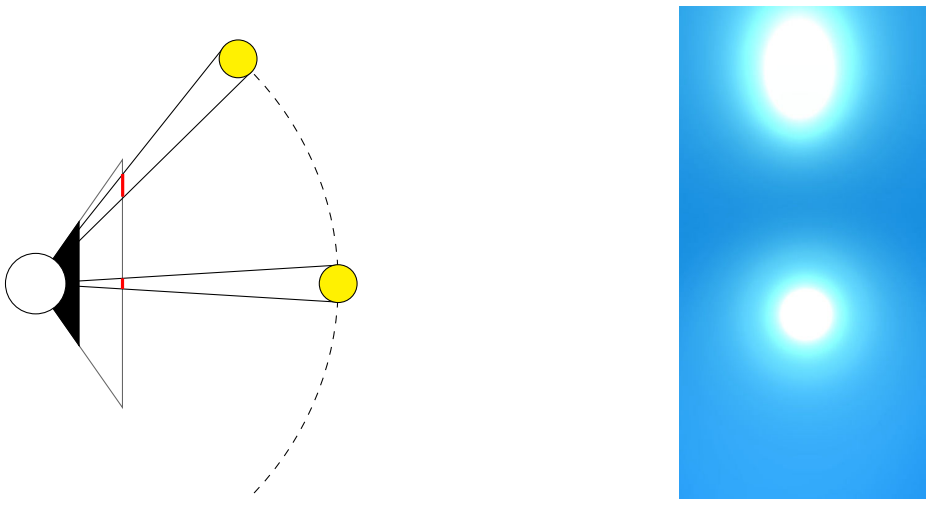
Tato implementace ke slunci částečně přistupuje jako k bodovému zdroji světla (nikoli jako ke kruhovému). Důsledkem toho je, že pokud je zakryta spodní část slunce (od středu dolů), slunce vrhá paprsky pouze nahoru, i když by vrchní polovina slunce měla taktéž vrhat paprsky. Pro odstranění tohoto jevu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



Obrázek 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.



Obrázek 5.24: Demonstrace deformace slunce při promítání na rovinu

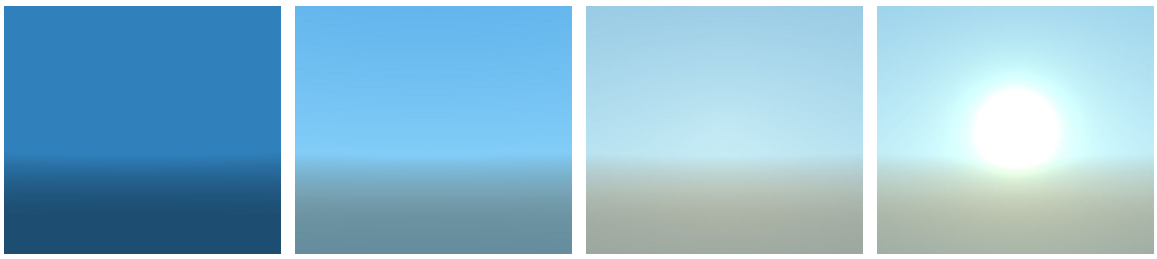
Výpočet pak probíhá pouze pro pixely s průhledností menší než jedna (*skybox* se tedy nepočítá pro pixely zcela překryté popředím). Barva každého pixelu se počítá dle následující rovnice:

$$\begin{aligned} k_1 = &\ \text{clamp}_{\langle 0,1\rangle}((\vec{n}_z + 0.1)\cdot 5)^2 && \text{(výška nad horizontem)} \\ d_{sun} = &\ |\vec{n}_{light} - \vec{n}| \\ \vec{c} = &\ \vec{c}_{base}\cdot(0.4 + 0.6\cdot k_1) && \text{(základní barva)} \\ &+\vec{c}_{horizon}\cdot(1 - |\vec{n}_z|) && \text{(horizont)} \\ &+\vec{c}_{halo}\cdot \max_0(1 - d_{sun}\cdot 0.5), && \text{(záře od slunce)} \\ &+\vec{c}_{sun}\cdot min(1, s_{size}/d_{sunPx})^{s_{pow}}\cdot k12 && \text{(slunce)} \end{aligned} \tag{5.1}$$

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

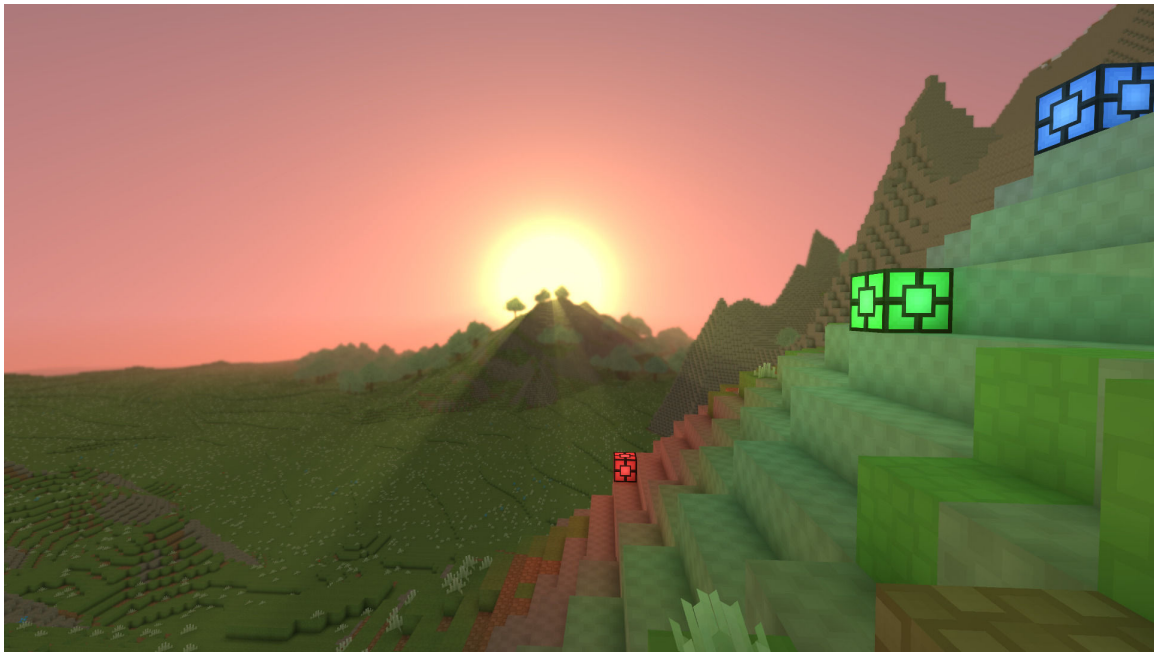


Obrázek 5.25: Postupná aplikace jednotlivých složek *skyboxu*

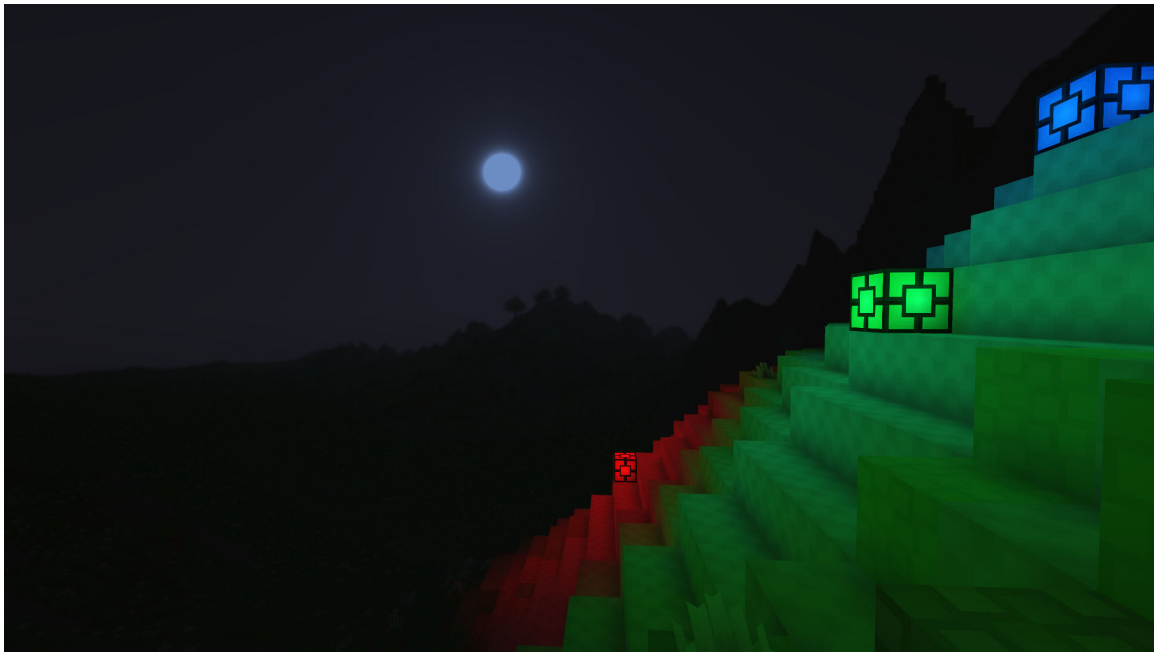
Parametry ovlivňující vzhled *skyboxu* jsou *shaderu* předávány přes uniformní paměť. Hodnoty nejsou konstantní, počítají se interpolací z přednastavených hodnot na základě denní doby. Hodnoty parametrů pro *skybox* určuje třída `WorldEnvironment` (implementace pro výchozí svět je v třídě `WorldEnvironment_Overworld`). 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.

Obrázek 5.26: Denní doba v aplikaci: ráno

Obrázek 5.27: Denní doba v aplikaci: dopoledne



Obrázek 5.28: Denní doba v aplikaci: večer



Obrázek 5.29: Denní doba v aplikaci: noc

Kapitola 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 nápovědu
- `--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 ladicí režim OpenGL s vypisováním chyb do konzole
- `--colectPerfData`: Aktivuje ukládání metrik do csv souboru.
- `--position=X`: Umístí kameru 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 *frustum cullingu*)
F4	Zapnutí/vypnutí vizualizace načítání chunků
F5	Uložit aktuální pozici kamery do `positions.txt`
F6	Načíst pozici kamery z `positions.txt` (cyklicky prochází všechny záznam
O/P	Posun denní doby
I	Zapnutí/vypnutí automatického posunu denní doby

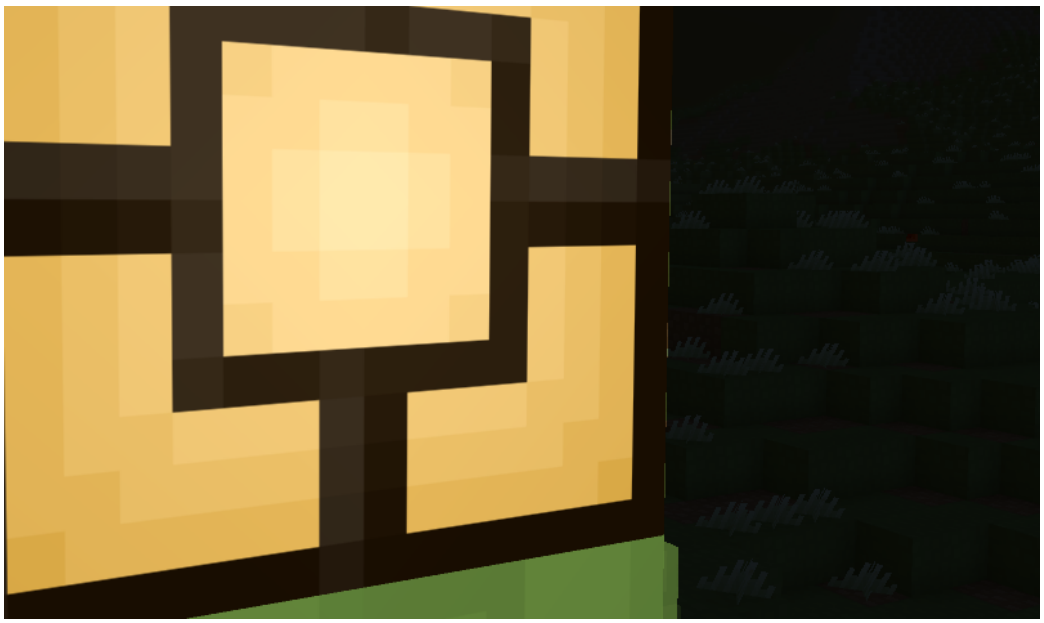
Tabulka 6.1: Ovládání aplikace

Obrázek 6.1: Náhled aplikace

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

1. **Zobrazovaná data**; Pro ladicí účely lze místo textur zobrazovat např. normálu, hloubkové informace apod. Jednou z možností je „Aggregation“, která vizualizuje agregaci stěn.

2. ***Shading***. Druhým *comboboxem* 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.



Obrázek 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

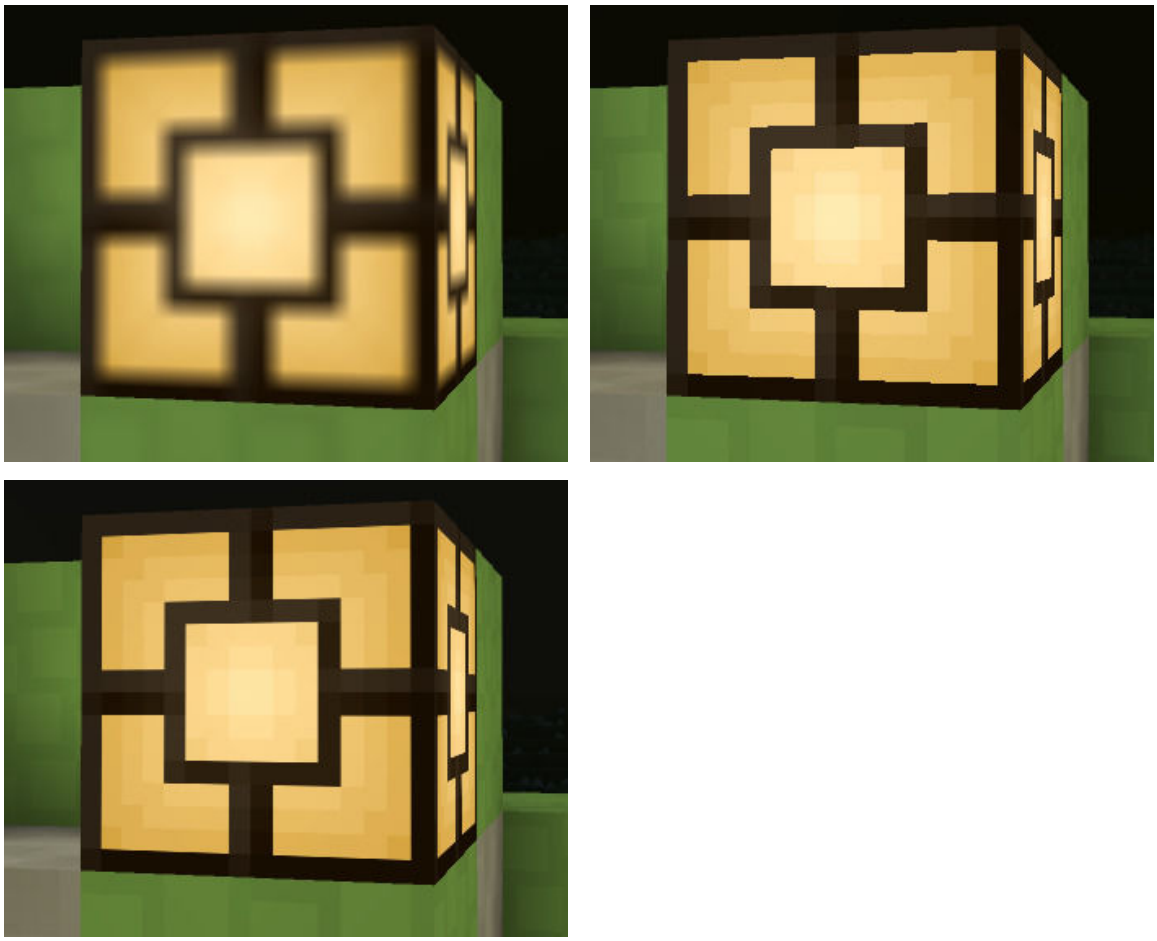
8. ***God rays***

9. **Dohledová vzdálenost**. Celkově je zobrazeno $(x + 1 + x)^2$ chunků, kde x je číslo udané v GUI.

10. **Zobrazení jednotlivých vrstev *depth peelingu***

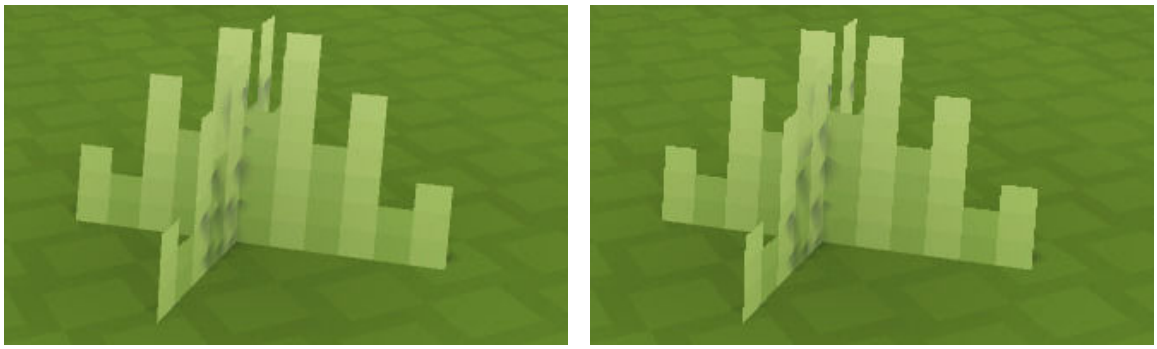
11. **Počet vrstev *depth peelingu***. V tomto menu lze transparentní vykreslování úplně vypnout nebo změnit počet průhledných vrstev. Vizualizace vypadá plnohodnotně už při jedné vrstvě a protože je tato technika výpočetně náročná, pro slabší počítače se doporučuje počet vrstev omezit.

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 rozmazaně; `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 by byly artefakty patrné.



Obrázek 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).



Obrázek 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.

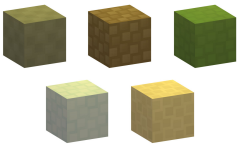
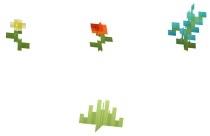
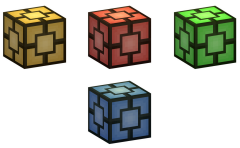
15. ***T-junction hiding*** aktivuje efekt v *postprocessingu*, který potlačuje artefakty způsobené *t-junctions*, které jsou při agregaci vytvářeny, viz sekce 5.3.3.

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.

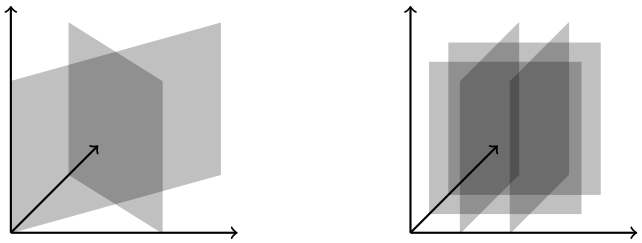
18. **Denní doba**. Posunem jezdce lze měnit denní dobu světa.

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/vespod
	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).



Obrázek 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`)

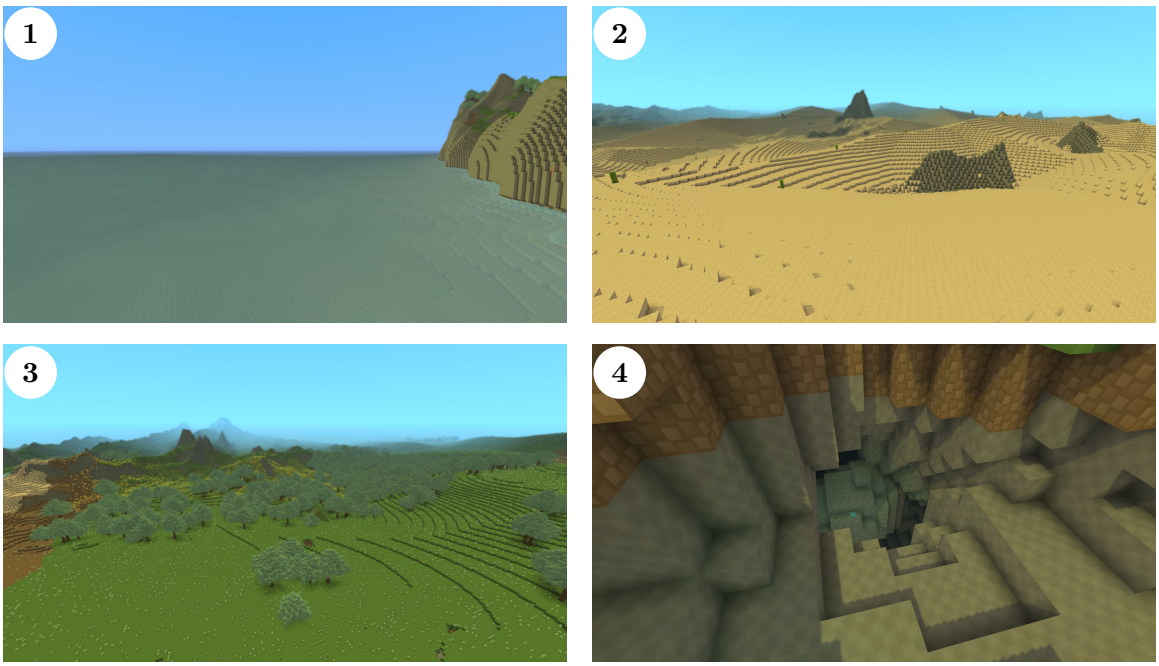
Pokud není uvedeno jinak, měření probíhá při těchto podmínkách:

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

Měření budou prováděna na níže uvedených scénách. Scéna definuje přesnou pozici a natočení 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 zkopírovat na disk a soubor terénu `/referenceScenes.sqlite` přesunout do složky `save` (umístěné na stejné úrovni se složkou `bin`). Poté je možné jednotlivé scény zobrazit příkazem `./ac --saveName=referenceScenes --position=x`, kde `x` je číslo scény mínus 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 – *alpha testing*, 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



Obrázek 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.
$N_\triangle$	–	Počet vykreslovaných trojúhelníků.
$N_\triangle^+$	$O/T/S$	Počet vykreslovaných trojúhelníků. Údaj je rozdělen na trojúhelníky vykreslované jako neprůhledné (O), trojúhelníky vykreslované v jedné vrstvě *depth peelingu* (T) a trojúhelníky vykreslované do *shadow mapy* (S). Celkový počet vykreslovaných trojúhelníků je $O+T\cdot d+S$, kde d je počet vrstev *depth peelingu*.

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 sestavování 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.

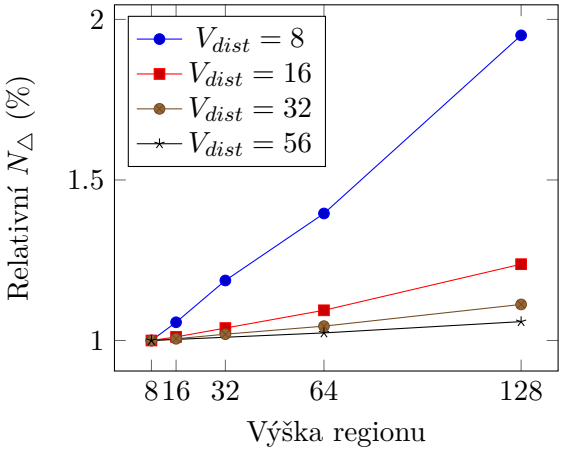
Zvýšení velikosti regionu může znamenat:

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 taktéž 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 znovu sestavit vykreslovací data pro celý region. Sestavování jednoho regionu je ale příliš rychlé na to, aby tento efekt mohl být znát.

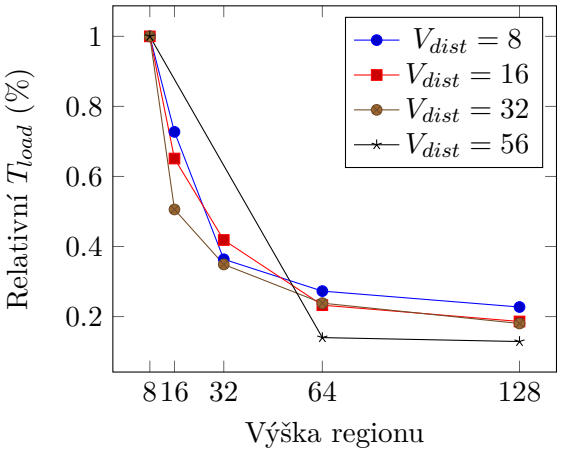
Bylo provedeno měření závislosti snímkové frekvence a doby načítání na výšce regionu v referenčním prostředí 3:

Dohl. vzd.		Výška regionu 8	16	32	64	128
$V_{dist} = 8$	FPS	79	79	79	79	79
	$N_\triangle$	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
	$N_\triangle$	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
	$N_\triangle$	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
	$N_\triangle$	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 $N_\triangle$ 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 nezkrátí, 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 rychlost 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	Poušť	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	Poušť	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.

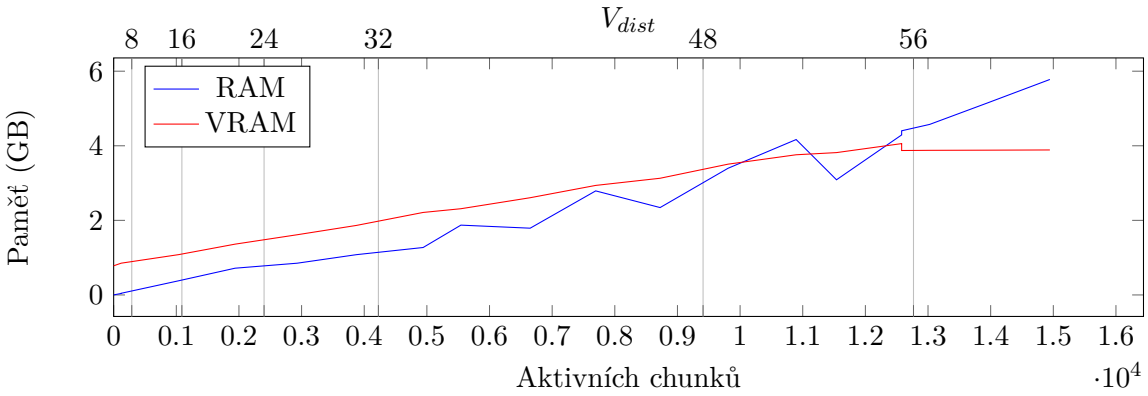
Scéna			**Velikost *light map* textur (v chuncích)** 2×2	4×4	8×8	16×16	32×32
1	Oceán	VRAM	1 931 MB	1 967 MB	2 043 MB	2 219 MB	2665 MB
		FPS	51	56	57	57	57
2	Poušť	VRAM	1 931 MB	1 965 MB	2 041 MB	2 217 MB	2667 MB
		FPS	52	54	55	56	56
3	Krajina	VRAM	1 931 MB	1 967 MB	2 043 MB	2 219 MB	2 665 MB
		FPS	41	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 = 128×128×256 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 agregační metody na snímkovou frekvenci a počet vykreslovaných trojúhelníků:

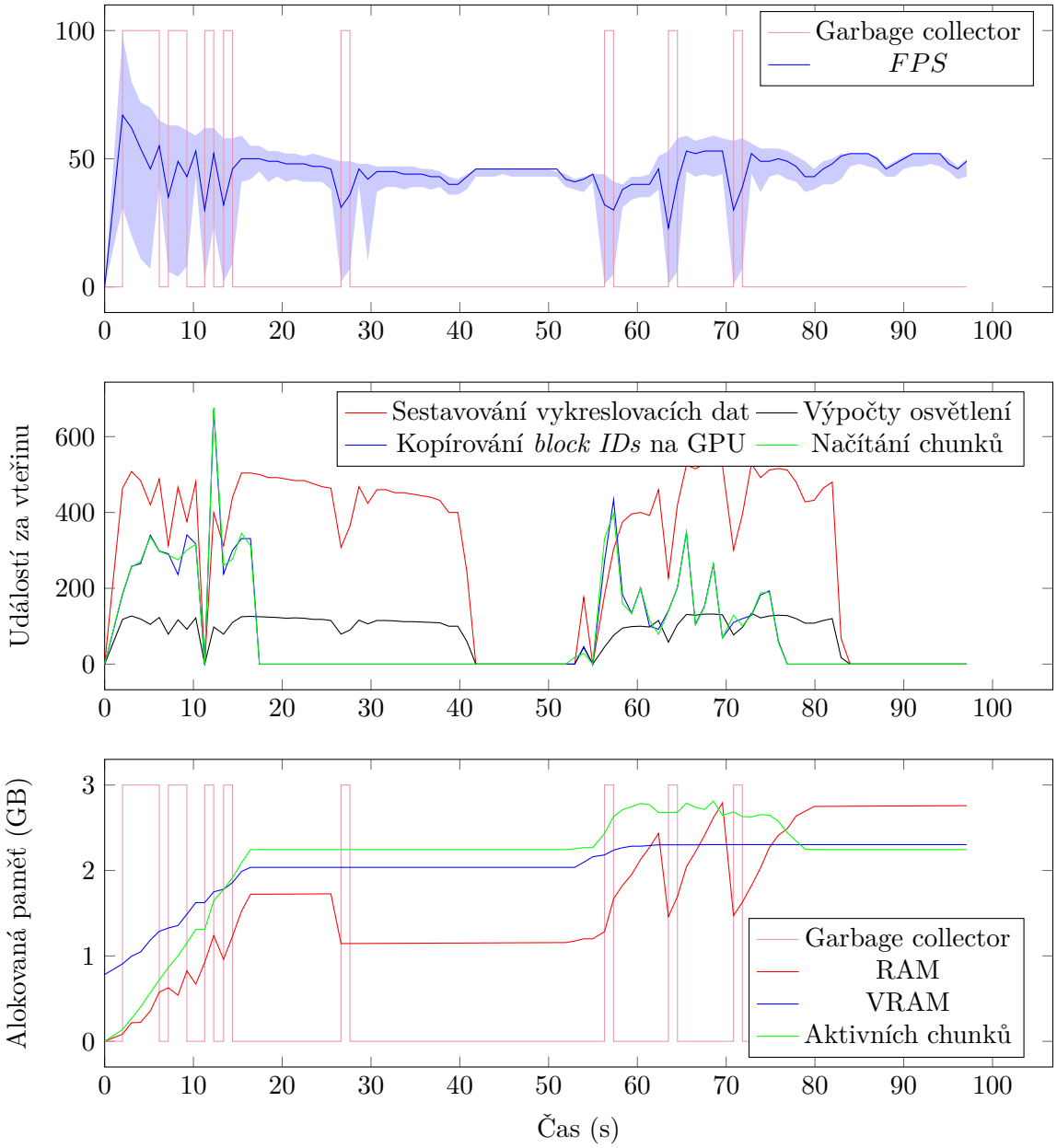
Scéna		Bez agregace	Lines	Squares	Squares ext
1 Oceán	$N_{\triangle}^{O}$	1611 k (100 %)	639 k (40 %)	553 k (34 %)	480 k (30 %)
	$N_{\triangle}^{T}$	560 k (100 %)	71 k (13 %)	10 k (2 %)	9 k (2 %)
	$N_{\triangle}^{S}$	73 k (100 %)	32 k (44 %)	30 k (40 %)	26 k (36 %)
	FPS	35	54	56	56
2 Poušť	$N_{\triangle}^{O}$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

Tabulka 6.9: Závislost snímkové frekvence a $N_{\triangle}^{+}$ na metodě agregace

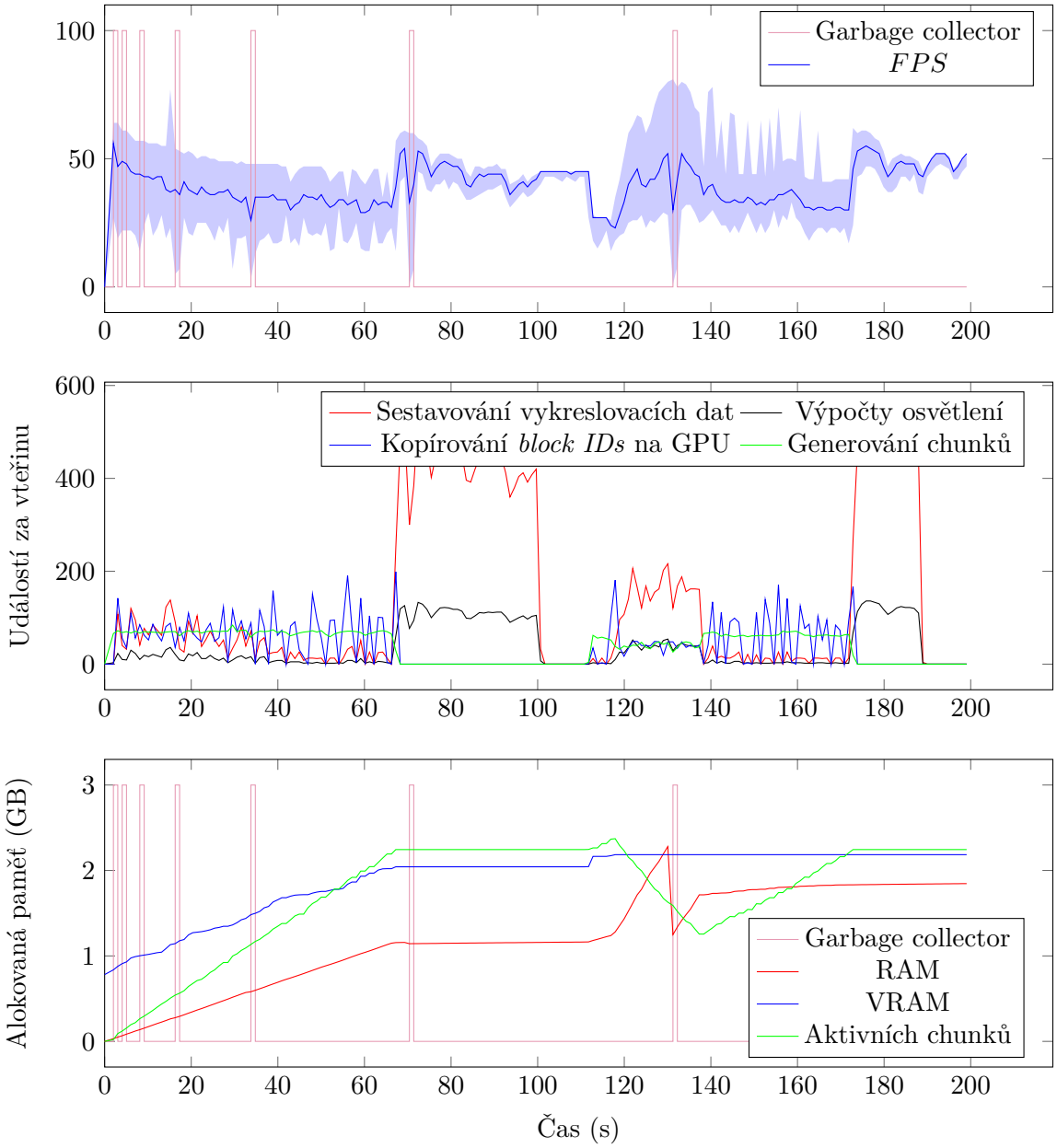
Data ukazují, že agregace je velmi účinnou optimalizační technikou, která dokáže značně zvýšit rychlost hry. Nejefektivnější z testovaných metod je `Squares ext`, která dokáže omezit počet vykreslovaných trojúhelníků až o 70 % u neprůhledných bloků a až o 98 % u průhledných bloků. Drastické snížení počtu 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 kameře. 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 20 s 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×) 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 prodleva 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 rychlost 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). Rychlost 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 dále 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× MSAA oproti výchozím 2×.

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

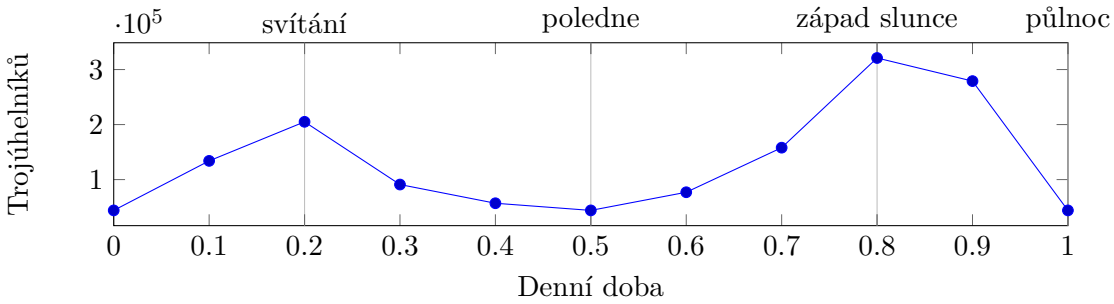
Tabulka 6.10: Závislost snímkové frekvence na dohledové vzdálenosti a metodě *shadingu* při 4× MSAA

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 stejné dráze, jako slunce, pouze rychleji a v opačném směru.

V_{dist}:	4	8	16	32
$N^S_\triangle$ v poledne:	39 k	50 k	50 k	50 k
$N^S_\triangle$ 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*.

	Shadow mapping			
MSAA	*off*	1024^2	2048^2	4096^2
1×	54	49	39	24
2×	47	41	35	22
4×	35	32	28	20
8×	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 *depth peelingu*** 0	1	2	3
1	Oceán	FPS	81	64	61	57
		$N_\triangle$	506 k	515 k	525 k	534 k
2	Poušť	FPS	68	61	59	56
		$N_\triangle$	1 163 k	1 163 k	1 163 k	1 163 k
3	Krajina	FPS	54	50	47	45
		$N_\triangle$	1 811 k	1 811 k	1 811 k	1 812 k
4	Jeskyně	FPS	100	86	80	75
		$N_\triangle$	102 k	102 k	102 k	102 k

Tabulka 6.13: Závislost snímkové frekvence a $N_\triangle$ 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	Poušť	Krajina	Jeskyně
Výchozí nastavení	57	56	41	59
Depth of field off	66	65	46	71
God rays off	59	56	42	59
Lepší texturování *off*	57	56	41	59
MSAA alpha test off	57	56	41	59
T-junction hiding off	57	56	41	59
Animace bloků *off*	57	56	41	59

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

Kapitola 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 výpočtů osvětlení na GPU, systém agregace stěn voxelů do jednoho primitiva (taktéž 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, taktéž 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 zhotoveno v souladu se zadáním.

Literatura

[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.

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