Bakalářská práce

České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra počítačové grafiky a interakce

Herní engine pro hry typu Minecraft

Zdeněk Kolář

Vedoucí: Ing. Ladislav Čmolík, Ph.D.
Obor: Otevřená informatika
Studijní program: Počítačové hry a grafika
Květen 2021

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

ČVUT
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Kolář** Jméno: **Zdeněk** Osobní číslo: **474379**

Fakulta/ústav: **Fakulta elektrotechnická**

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

Studijní program: **Otevřená informatika**

Studijní obor: **Počítačové hry a grafika**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Herní engine pro hry typu Minecraft

Název bakalářské práce anglicky:

Game engine for Minecraft-like games

Pokyny pro vypracování:

Seznamte se s možnostmi pro generování a vykreslování herního prostředí pro hry typu Minecraft. Na základě analýzy s využitím Unity navrhněte a implementujte herní engine umožňující generování herního prostředí podobného hře Minecraft a jeho vykreslování pomocí grafu viditelnosti. Dále využijte metod pro vyhodnocení zastínění objektů dostupných pro Unity (např. hierarchický z-buffer) a experimentujte s možnostmi členění vykreslované scény (např. regulární mřížka, octree). Schopnosti herního engine otestujte vygenerováním a vykreslením alespoň pěti různých herních prostředí. Změřte a porovnejte vliv způsobu členění scény a použité metody pro vyhodnocování zastínění objektů na výkon vykreslování vygenerovaných herních prostředí.

Seznam doporučené literatury:

[1] N. A. Borromeo. Hands-On Unity 2020 Game Development: Build, customize, and optimize professional games using Unity 2020 and C#. Packt Publishing, 2020.
[2] H. Zhang, D. Manocha, T. Hudson, and K. E. Hoff III. Visibility culling using hierarchical occlusion maps. In Proceedings of the 24th annual conference on Computer graphics and interactive techniques, pp. 77-88. 1997.
[3] J. Bittner, V. Havran, and P. Slavik. Hierarchical visibility culling with occlusion trees. In Proceedings. Computer Graphics International, pp. 207-219. IEEE, 1998.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Ladislav Čmolík, Ph.D., Katedra počítačové grafiky a interakce

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

Datum zadání bakalářské práce: **11.02.2021** Termín odevzdání bakalářské práce: **21.05.2021**

Platnost zadání bakalářské práce: **30.09.2022**

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

III. PŘEVZETÍ ZADÁNÍ

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

.

Datum převzetí zadání	Podpis studenta

© VUT v Praze, Design: VUT v Praze, VIC

Poděkování

Chtěl bych poděkovat panu Ing. Ladislavu Čmolíkovi, Ph.D. za cenné a užitečné rady, a hlavně za jeho trpělivost při vedení této práce. Dále bych chtěl poděkovat mým nejbližším, kteří mě při psaní této práce vždy podporovali.

Prohlášení

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

V Praze, 20. května 2021

Abstrakt

Tématem této bakalářské práce je generování a vykreslování herních prostředí pro hry podobným hře Minecraft. Pro tyto hry je specifický zdánlivě nekonečný procedurálně generovaný volumetrický svět, ve kterém se hráč může volně pohybovat a libovolně ho měnit. Primárním cílem této práce je využít metod vyhodnocování zastínění objektů (graf viditelnosti používaný v Minecraftu, hierarchický ZBuffer) a experimentovat se způsoby členění scény (oktalové stromy, pravidelná mřížka) a porovnat jejich vliv na výkon vykreslování. Během práce je navržen jednoduchý framework pro generování a vykreslování takových světů. V herním enginu Unity byla na základě navrženého frameworku vytvořena aplikace umožnující generování a vykreslování pěti různých prostředí. V této aplikaci je dále změřen vliv výše zmíněných metod na výkon vykreslování.

Klíčová slova: voxel, Minecraft, viditelnost, odstraňování okluzí, oktalový strom, hierarchický Z-Buffer, vykreslování, výkon, Unity

Vedoucí: Ing. Ladislav Čmolík, Ph.D.

Abstract

The theme of this Bachelor thesis is generating and rendering Minecraft-like game environments. Minecraft-like games are specific by seemingly endless procedurally generated volumetric worlds, where a player can freely move and modify any part as pleased. The primary goal of the thesis is to measure the impact of the application of different occlusion culling methods, such as a visibility graph used in Minecraft and hierarchical ZBuffer, and scene hierarchies, as an octree and a uniform grid, on the rendering performance. Within the thesis, a design of a simple framework allowing the generation and rendering of such worlds is presented. Based on the designed framework, an application was created in the Unity game engine, enabling the generation and rendering of five different environments. In the application, the impact of the abovementioned methods on the rendering performance was measured.

Keywords: voxel, Minecraft, visibility, occlusion culling, octree, hierarchical Z-Buffer, rendering, performance, Unity

Title translation: Game engine for Minecraft-like games

Obsah

1 Úvod 1

1.1 Cíle práce 2

1.2 Struktura práce 3

2 Analýza a Návrh 5

2.1 Generování 6

2.1.1 Šumy 6

2.1.2 Funkce orientované vzdálenosti 11

2.1.3 Proces generování v demonstračním frameworku 12

2.2 Reprezentace volumetrických scén 13

2.2.1 Dělení světa v Minecraftu . . . 15

2.2.2 Reprezentace scény ve frameworku 16

2.3 Vykreslování 19

2.3.1 Metody přímého vykreslování 19

2.3.2 Vykreslování skrze geometrická primitiva 20

2.3.3 Určování viditelnosti objektů ve scéně 21

2.3.4 Vykreslování světa v Minecraftu 24

2.3.5 Vykreslování scény ve frameworku 25

3 Implementace 31

3.1 Herní engine Unity 31

3.2 Reprezentace světa 33

3.3 Generování světa 36

3.4 Vykreslování 39

3.4.1 Vyhodnocování viditelnosti . . 40

3.4.2 Procházení scénou 42

3.4.3 Vykreslování 42

4 Výsledky 45

4.1 Demonstrační aplikace 45

4.1.1 Voxely 45

4.1.2 Generátory scén 48

4.1.3 Uživatelské rozhraní 52

4.2 Měření výkonu vykreslování 58

4.2.1 Hory ve vodě 59

4.2.2 Oceán 59

4.2.3 Poušť 62

4.2.4 Krajina s jeskyněmi 62

4.2.5 Jeskyně 63

4.2.6 Čas vyhodnocování viditelnosti při aplikaci hierarchického ZBufferu 63

5 Závěr **67**

Reference **69**

A Seznam příloh **73**

Obrázky

1.1 Terén hor v Minecraftu. Zdroj: [11] 2

2.1 Nalevo: Spojitý šum hodnot. Napravo: Princip interpolace v 3D prostoru. Zdroj: [21] 7

2.2 Tři části výpočtu perlinova šumu. Shora: mřížka gradientů, částečně vizualizovaný skalární součin pro body v mřížce, interpolované skalární součiny. Zdroj: [15] 9

2.3 Voroného diagram. Zdroj: [23] . . 10

2.4 Voroného diagramy pro různé typy pravidelných mřížek. Červené body značí body množiny $\mathcal{P}$. Zdroj: [8] . 10

2.5 Vliv míry relaxace r na pozici bodu $\overrightarrow{p}$ v pravidelné čtvercové mřížce. Zleva: $r = 0$, $r = 0.5$ a $r = 1$. Je zobrazena vzdálenost od nejbližšího bodu $\overrightarrow{p}$. Tmavší barva značí menší vzdálenost. 10

2.6 Sčítání šumů. Každý šum je vzorkován s dvakrát větší frekvencí než ten předchozí (zleva) a má dvakrát menší amplitudu. Výsledný šum je zobrazen vpravo. 11

2.7 Voroného diagram. Pozice bodů v prostoru jsou při výpočtu náhodně posunuty. 11

2.8 Zleva: Ukázka trojrozměrných primitiv reprezentovaných pomocí SDF (zdroj: [16]) a komplexní scéna z nich vytvořená (zdroj: [16]) 12

2.9 Oblasti generování a mazání bloků voxelů 13

2.10 Zleva: Dělený region, odpovídající oktalový strom výšky 2. Zdroj:[13] 15

2.11 Plná část sloupce voxelů. Zdroj: [10, příspěvek "Chunk"] 15

2.12 Struktura octree voxelů v bloku. 16

2.13 Vliv slučování na hierarchii scény Pouště 18

2.14 Hierarchie scény 19

2.15 Vrhání paprsku dvourozměrnou scénou. 19

2.16 Hraniční reprezentace koule vytvořená z krychlí (vlevo) a pomocí techniky *marching cubes* 20

2.17 Princip pohledového jehlanu . . 22

2.18 Vykreslená scéna (vlevo) a korespondující Z-Buffer. Zdroj: [24] 22

2.19 Hierarchický Z-Buffer a vykreslená scéna, jejíž hloubku reprezentuje. Zdroj: [6] 23

2.20 Průchod scénou podle kriterií uvedených v 2.3.4. Zdroj: [4] 24

2.21 Graf viditelnosti pro dvourozměrný blok 16 × 16 voxelů. Zdroj:[4] 25

2.22 Proces vygenerování geometrie bloku voxelů. 26

2.23 Transformovaný hierarchický ZBuffer bez korekce a s korekcí. . . . 28

2.24 Neprůhlednost stěny v grafu viditelnosti 28

2.25 Procházení scénou 29

3.1 Objektový diagram reprezentace světa. 34

3.2 Odvození indexu potomka z pozice. Zdroj: [7] 35

3.3 Objektový diagram reprezentace světa doplněný o jeho generování . 36

3.4 Formát grafu viditelnosti 39

3.5 Objektový diagram vykreslovacího enginu frameworku 39

3.6 Formát dat s informacemi o uzlu 40

3.7 Promítnutí krychle na pohledový jehlan a ohraničující box 41

3.8 Formát výstupu `CullingCS` pro jeden uzel 41

4.1 Scéna hor ve vodě 48

4.2 Scéna oceánu 49

4.3 Hierarchický ZBuffer pro scénu oceánu (mip-level 0) 49

4.4 Scéna pouště 50

4.5 Scéna krajiny s jeskyněmi 50

4.6 Scéna jeskyně 51

4.7 Sekce uživatelského rozhraní s možností zobrazení sekcí 52

4.8 Sekce uživatelského rozhraní s nabídkou generátorů scén 52

4.9 Sekce uživatelského rozhraní s nabídkou zobrazení 53

4.10 Pohled kamerou třetí osoby . . . 53

4.11 Sekce uživatelského rozhraní se statistikami 54

4.12 Sekce uživatelského rozhraní s nápovědou k ovládání 56

4.13 Sekce uživatelského rozhraní s nastavením aplikace 56

4.14 Vliv slučování dat k vykreslení a kombinací metod rozhodování o viditelnosti na snímkovou frekvenci ve scéně Hory . . . 60

4.15 Průměrné časy fází vykreslování ve scéně Hor pro vzdálenost 32 . . . 60

4.16 Vliv slučování dat k vykreslení a kombinací metod rozhodování o viditelnosti na snímkovou frekvenci ve scéně Ocean . . . 61

4.17 Průměrné časy fází vykreslování ve scéně Oceán pro vzdálenost 32 . 61

4.18 Vliv slučování dat k vykreslení a kombinací metod rozhodování o viditelnosti na snímkovou frekvenci ve scéně Poušť . . . 62

4.19 Průměrné časy fází vykreslování scény Pouště pro vzdálenost 32 . . . 63

4.20 Vliv slučování dat k vykreslení a kombinací metod rozhodování o viditelnosti na snímkovou frekvenci ve scéně Krajina s jeskyněmi . . . 64

4.21 Průměrné časy fází vykreslování scény Krajiny s jeskyněmi pro vzdálenost 32 . . . 64

4.22 Vliv slučování dat k vykreslení a kombinací metod rozhodování o viditelnosti na snímkovou frekvenci ve scéně Jeskyně . . . 65

4.23 Průměrné časy fází vykreslování scény Jeskyně pro vzdálenost 32 . . 65

4.24 Vliv generování hierarchického ZBufferu na čas vyhodnocování viditelnosti . . . 66

Tabulky

Úvod

Minecraft[1] je již deset let jednou z nejpopulárnějších a nejprodávanějších počítačových her. Svou popularitu si získal především díky zdánlivé nekonečnému procedurálně generovanému volumetrickému světu. Svět je reprezentován voxelovou mřížkou a je možné ho po jednotlivých voxelech, které komunita Minecraftu označuje jako bloky, libovolně modifikovat. Tato bezprecedentní úroveň svobody, jež je hráči v Minecraftu poskytnuta, dala za zrod ohromné komunitě fanoušků z celého světa. Stavitelé jsou schopni postavit stavby s neuvěřitelným stupněm realismu. Technicky zdatní zase dokáží díky pouhé interakci mezi sousedními bloky sestavit komplexní mechanismy jakými jsou kalkulačka nebo jednoduché CPU. Dále tu jsou tvůrci modifikací, jež Minecraft obohacují o různorodou funkcionalitu. Ať už v podobě optimalizací, vizuálních prvků (textury apod.) nebo nových typů voxelů a předmětů.

Otevřený a plně modifikovatelný svět Minecraftu spolu s možností vytvářet vlastní modifikace skutečně otevírá dveře fantazii tvůrců z celého světa. Nicméně realizace takového herního prostředí je netriviální úlohou z pohledu správy paměti a počítačové grafiky.

[1] www.minecraft.net

Obrázek 1.1: Terén hor v Minecraftu. Zdroj: [11]

1.1 Cíle práce

Realizace světa podobného tomu v Minecraftu se dá rozdělit do tří podúloh:

1. Vygenerování volumetrických dat
2. Komprese a správa paměti s těmito daty
3. Vykreslování vygenerovaných dat

Cílem této práce je zmapovat metody používané při generování, ukládání a vykreslování herních prostředí podobných Minecraftu (na obrázku 1.1 je vidět klasický terén hor v Minecraftu). Na základě této analýzy navrhnout framework, s nímž bude možné generovat a vykreslovat taková prostředí. V herním enginu Unity vytvořit demonstrační aplikaci využívající navržený framework. Dále využít metod pro vyhodnocování zastínění objektů (hierarchický Z-Buffer) a členění světa (octree, pravidelná mřížka) a zhodnotit jejich vliv na rychlost vykreslování světa. Po implementaci bude na nejméně pěti různých vygenerovaných prostředích změřen dopad zmíněných metod na výkon vykreslování.

1.2 Struktura práce

Následující text práce je členěn do 4 kapitol. Kapitola 2 se soustředí na zmapování metod používaných pro generování, ukládání a vykreslování. Dále na návrh frameworku umožňujícího vygenerovat a vykreslovat světy stejného charakteru jako v Minecraftu. Kapitola 3 a 4 je věnována implementaci demonstrační aplikace v herním enginu Unity a testování výsledného produktu. Důraz je kladen především na vykreslování.

Kapitola 2

Analýza a Návrh

Jak bylo naznačeno v kapitole 1, Minecraft je charakteristický svým ohromným a plně modifikovatelným světem. Tento svět je tvořen 3D objekty, jež jsou zarovnány do pravidelné mřížky. Z toho důvodu jsou těmito 3D objekty většinou krychle reprezentující nejrůznější materály. Nicméně, mohou jimi být i objekty s komplexnější geometrií, například zábradlí, truhla apod. [9, sekce Gameplay]. Takto zarovnané objekty jsou v komunitě Minecraftu označovány jako bloky. V této práci je však budeme nazývat voxely vzhledem k tomu, že každý blok představuje elementární část prostoru (*volume element*) ve světě. Každý voxel obsahuje informace vztahující se k jeho konkrétnímu typu. Voxel reprezentující kámen může obsahovat pouze data k jeho vykreslení. Naopak voxel reprezentující truhlu obsahuje kromě vykreslovaných dat i informace o položkách, jež se v truhle nachází.

Základní herní prostor světa Minecraftu je ohraničen boxem o rozměrech $60 * 10^6$ voxelů (bloků) v horizontálních osách X a Z souřadnicového systému a 256 voxelů ve vertikální ose Y ([10, příspěvek "World boundary"]). Je snadné si dopočítat, že uložení celého světa na disk by vyžadovalo ohromné množství místa. Svět je tedy procedurálně generován okolo hráče. Generováním se zabývá sekce 2.1 této kapitoly. Jsou v ní probrány techniky, kterými je možné přiřadit typ voxelu libovolné pozici ve světě Minecraftu, aby svět vypadal realisticky. Dále je zde zmíněno jakým způsobem se generuje svět v navrhovaném frameworku.

Přestože se generuje pouze část scény okolo hráče, je nutné vygenerovaná data efektivně ukládat do paměti. Touto problematikou se zabývá sekce 2.2, kde jsou podobně jako v sekci 2.1 popsány techniky reprezentace volumetrických scén, jak tuto problematiku řeší Minecraft a jak je řešena v této práci.

Poslední sekcí je sekce 2.3 pokrývající zobrazování volumetrických scén. Opět jsou zde představeny používané techniky, jak je zobrazován svět Minecraftu a jak je zobrazování řešeno v navrhovaném frameworku.

2.1 Generování

Svět Minecraftu je organizován do pravidelné pravoúhlé mřížky. Při generování hledáme funkci, jež voxelu na libovolných souřadnicích v mřížce přiřadí jeho typ. Minecraft dále umožňuje generovat libovolné světy na základě tzv. *seedu*, který hráč zadá před vygenerováním světa. Hledáme tedy funkci:

$$voxelType(\overrightarrow{pos}, seed) : \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{N}_0 \rightarrow \mathcal{V} \tag{2.1}$$

kde $\overrightarrow{pos} = [x, y, z]$ představuje souřadnice voxelu v mřížce a $\mathcal{V}$ je množina všech možných typů voxelu. Vertikální osu souřadnicového systému zvolíme Y.

2.1.1 Šumy

Funkce šumů jsou funkce, jež souřadnici v prostoru přiřadí náhodnou hodnotu. Pro účely této práce budeme uvažovat pouze šumy pro souřadnicové systémy dimenze tři a dva. Dvourozměrné šumy lze využít například pro vygenerování výškové mapy terénu, definování polohy měst, či jednotlivých biomů. Trojrozměrné šumy, je zase možné využít například k vymezení trojrozměrné oblasti, kde se mají vyskytovat jeskyně.

$$noise_{3D}(\overrightarrow{coord}, seed) : \mathbb{R}^3 \times \mathbb{N}_0 \rightarrow \mathbb{R} \tag{2.2}$$

$$noise_{2D}(\overrightarrow{coord}_{xz}, seed) : \mathbb{R}^2 \times \mathbb{N}_0 \rightarrow \mathbb{R} \tag{2.3}$$

Symbol $\overrightarrow{coord}$ označuje souřadnici v trojrozměrném kartézském souřadnicovém systému. Aby se definiční obor funkcí šumů shodoval s definičním oborem funkce $voxelType$, bereme pouze celou část prvků $\overrightarrow{coord}$. Jelikož mají funkce šumu spojitý výstup, definujeme práh $t \in \mathcal{R}$ pomocí něhož určíme, jestli se daný typ voxelu $v \in \mathcal{V}$ na souřadnicích nachází nebo ne. Dále definujeme měřítko $\overrightarrow{s} \in \mathbb{R}^3$, pomocí něhož můžeme ovlivňovat vzorkovací frekvenci na jednotlivých osách souřadnicového systému.

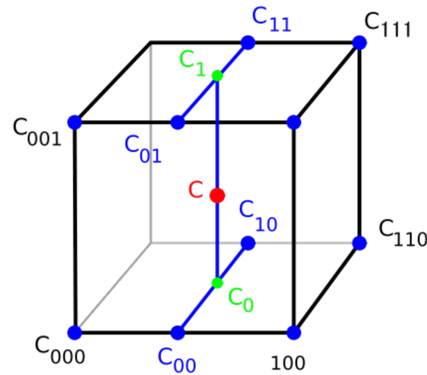
$$noise_{3D,\mathcal{Z},v}(\lfloor \overrightarrow{coord} \otimes \overrightarrow{s} \rfloor, seed) : \mathbb{Z}^3 \times \mathbb{N}_0 \rightarrow \mathcal{V} \tag{2.4}$$

$$\overrightarrow{pos} = \lfloor \overrightarrow{coord} \otimes \overrightarrow{s} \rfloor$$
$$noise_{3D,\mathcal{Z},v}(\overrightarrow{pos}, seed) = \begin{cases} v & noise_{3D}(\overrightarrow{pos}, seed) \leq t \\ o \in \mathcal{V} \setminus \{v\} & noise_{3D}(\overrightarrow{pos}, seed) > t \end{cases} \quad (2.5)$$

Operace $\vec{a} \otimes \vec{b}$ značí skalární součin vektorů $\vec{a}$ a $\vec{b}$. Typ voxelu o může být dále určen dalšími funkcemi šumů. Obdobně pro $noise_{2D}$. Pro dvojrozměrný šum můžeme nastavit $t = \lfloor \overrightarrow{coord_y} \rfloor$. Můžeme tak získat výškovou mapu terénu.

Šum hodnot

Value noise, v překladu, šum hodnot je zřejmě nejjednodušší funkcí šumu. Zde je každé pozici $\overrightarrow{pos}$ přiřazena náhodná hodnota. Aby byla zajištěna hodnota pro jakoukoli souřadnici kartézského souřadnicového systému, je výsledná hodnota pro $\vec{C} = \overrightarrow{coord} \otimes \vec{s}$ rovna konvexní kombinaci vzorků na pozicích $\overrightarrow{C_{xxx}} = \overrightarrow{pos} + (x, y, z)$, kde $x, y, z \in \{0, 1\}$. Jako metoda konvexní kombinace může být zvolena například trilineární nebo bilineární interpolace. Na obrázku 2.1 je ukázka šumu hodnot interpolovaného hermitovskou kubickou interpolační metodou a ukázka pozic $\overrightarrow{C_{xxx}}$ vzhledem k $\vec{C}$ v trojrozměrném prostoru.



Obrázek 2.1: Nalevo: Spojitý šum hodnot. Napravo: Princip interpolace v 3D prostoru. Zdroj: [21]

Perlinův a simplexový šum

Jelikož šum hodnot pouze interpoluje mezi hodnotami sousedních buněk, výsledná textura nemusí působit přirozeně. O něco realističtější výsledky za cenu vyšší výpočetní složitosti produkuje perlinův případně simplevoxý šum.

Perlinův i simplexový šum pochází od profesora Kenneth H. Perlina[15]. Jedná se o gradientní šumy. Mohou být implementovány pro libovolný počet dimenzí, nicméně nejvíce se využívají varianty pro dvou a troujrozměrné prostory. Výpočet perlinova šumu se dělí na tři části [15]:

1. Definování mřížky gradientů.
2. Vypočítání skalárního součinu gradientu a vektoru odsazení $\overrightarrow{C}$ od bodu v mřížce pro daný gradient
3. Interpolace mezi skalárnímy součiny bodů v mřížce obklopující bod $\overrightarrow{C}$

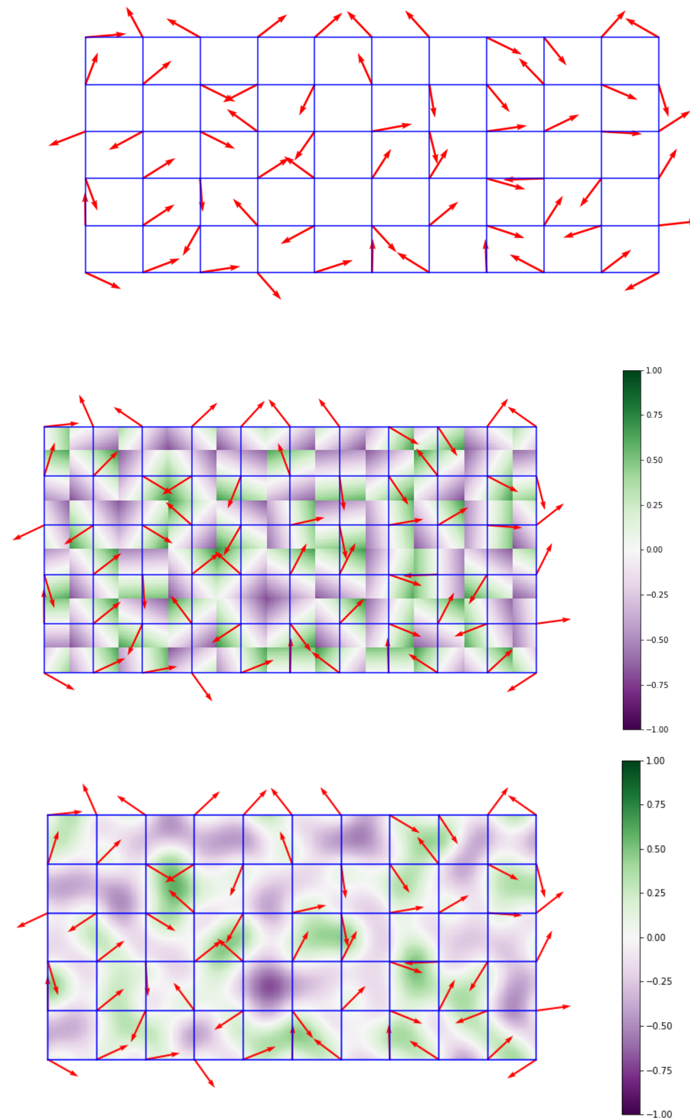
Na obrázku 2.2 jsou vyzualizovány tyto části výpočtu. Simplexový šum je vylepšení perlinova šumu. K největším výhodám patří například menší asymptotická složitost $\mathcal{O}(n^2)$ oproti $\mathcal{O}(2^n)$ perlinova šumu [18], kde n je počet dimenzí prostoru. Detailnější popsání algoritmu perlinova a simplexového šumu lze nalézt v práci Ing. Daniela Čejchana [1], která se rovněž zabývá zobrazováním volumetrických scén, v kapitole "Procedurální generování volumetrického terénu", nebo na příslušných stránkách wikipedie [15] a [18], kde jsou uvedeny reference na další práce zabývající se těmito tématy.

Voroného diagram

Pomocí voroného diagramů lze náhodně definovat oblasti. Těmto oblastem lze přiřadit stejné vlastnosti. V kontextu generování světů je možné tento šum využít k vymezení oblastí jednotlivých biomů, center měst a států apod.

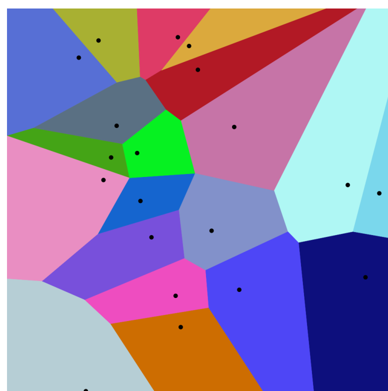
Voroného diagram rozděluje n-dimensionální prostor $\mathbb{R}^n$ do regionů podle množiny bodů $\mathcal{P} \subset \mathbb{R}^n$ a to tak, že pro každý bod $\overrightarrow{C} \in \mathbb{R}^n$ nalezne nejbližší bod $\overrightarrow{p} \in \mathcal{P}$ podle stanovené metriky. Bod $\overrightarrow{C}$ tak spadá do regionu bodu $\overrightarrow{p}$. Celému regionu pak připadají stejné vlastnosti, například stejná barva (viz Obrázek 2.3). Obrázek 2.3 ukazuje dělení roviny na regiony podle euklidovské metriky.

Pro velké prostory a obsáhlé množiny $\mathcal{P}$ by bylo výpočetně náročné hledat nejbližší bod $\overrightarrow{p}$. Můžeme ale každý bod množiny omezit buňkou pravidelné mřížky (viz. Obrázek 2.4). Regiony diagramu budou sice pravidelnější, ale získáme tak kontrolu nad tím, jaké body ověřovat na vzdálenost. Na obrázku 2.5 je dále zobrazeno, jak se projeví míra relaxace $r \in [0, 1]$ pozice bodu $\overrightarrow{p}$ v buňce mřížky. Míra relaxace $r = 0$ znamená, že pozice bodu $\overrightarrow{p}$ v buňce

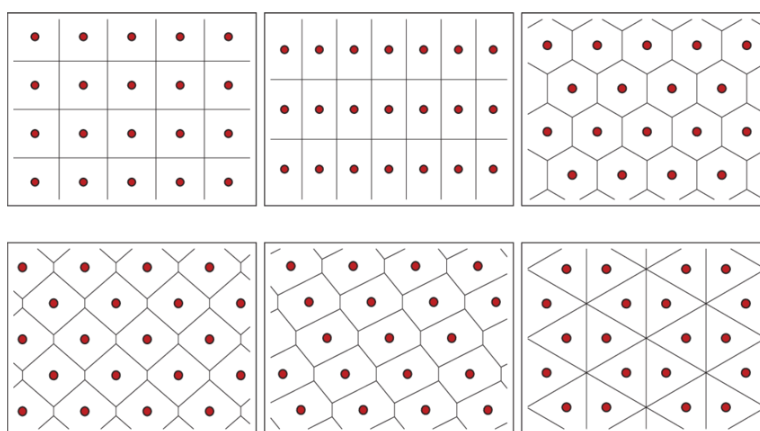


Obrázek 2.2: Tři části výpočtu perlinova šumu. Shora: mřížka gradientů, částečně vizualizovaný skalární součin pro body v mřížce, interpolované skalární součiny. Zdroj: [15]

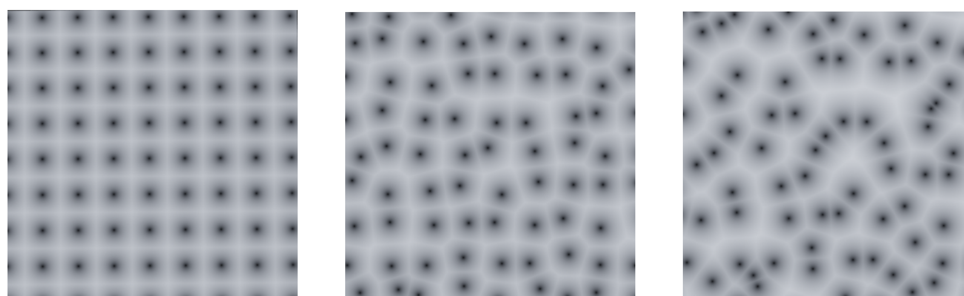
mřížky je jasně definována, $r = 1$ naopak značí, že bod $\overrightarrow{p}$ se může vyskytovat kdekoliv v buňce mřížky.



Obrázek 2.3: Voroného diagram. Zdroj: [23]



Obrázek 2.4: Voroného diagramy pro různé typy pravidelných mřížek. Červené body značí body množiny $\mathcal{P}$. Zdroj: [8]



Obrázek 2.5: Vliv míry relaxace r na pozici bodu $\overrightarrow{p}$ v pravidelné čtvercové mřížce. Zleva: $r = 0$, $r = 0.5$ a $r = 1$. Je zobrazena vzdálenost od nejbližšího bodu $\overrightarrow{p}$. Tmavší barva značí menší vzdálenost.

Skládání šumů

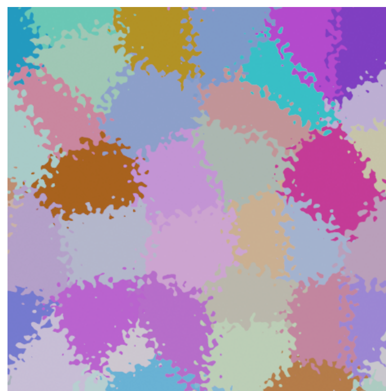
Pomocí jedné funkce šumu sice budeme schopni vygenerovat voxelovou scénu. Výsledek by však nemusel působit realisticky, což se od her jako je Minecraft

očekává. Vyšší realističnosti se dá docílit skládáním více funkcí šumů.

Sčítání šumů. Pro danou pozici ve scéně můžeme navzorkovat více šumů a jejich hodnoty následně sečíst, to je užitečné zejména při vytváření výškových map, kdy se vícekrát vzorkuje dvourozměrný šum vždy s jinou frekvencí a amplitudou. Tyto vzorky jsou sečteny a výsledkem je realističtější výšková mapa terénu (viz Obrázek. 2.6).

Obrázek 2.6: Sčítání šumů. Každý šum je vzorkován s dvakrát větší frekvencí než ten předchozí (zleva) a má dvakrát menší amplitudu. Výsledný šum je zobrazen vpravo.

Odsazení vzorkovací polohy. Zde je každá souřadnice parametru $\overrightarrow{coord}$ z rovnice 2.2 odsazena pomocí dalších funkcí šumu. Kdybychom například používali voroného diagram pro definování různých biomů, můžeme tento přístup použít pro plynulejší přechod mezi sousedními regiony (viz. Obrázek 2.7).



Obrázek 2.7: Voroného diagram. Pozice bodů v prostoru jsou při výpočtu náhodně posunuty.

2.1.2 Funkce orientované vzdálenosti

Funkce šumů jsou silným nástrojem pro vygenerování realistických scén. Někdy je ale nutné být schopný vygenerovat geometrická primitiva jako je kužel

pro vytvoření korun stromů, nebo jejich kombinace pro komplexnější konstrukty, například dům. Tento problém, lze vyřešit aplikací funkcí orientované vzdálenosti.

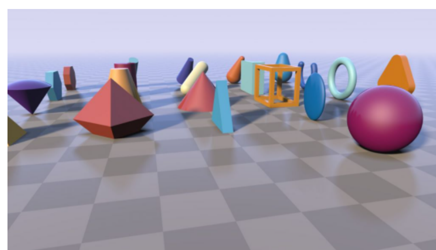
Mějme množinu $\mathcal{P} \subset \mathbb{R}^n$, kde $\mathbb{R}^n$ je metrický prostor s metrikou d. Funkce orientované vzdálenosti (Signed distance function - SDF) je funkce, jež bodu $\overrightarrow{x} \in \mathbb{R}^n$ přiřadí nejkratší vzdálenost od hranice množiny $\mathcal{P}$ podle metriky d. Pokud $\overrightarrow{x} \in \mathcal{P}$, je vzdálenost kladná, jinak je záporná. [17] Vhodnými kandidáty pro reprezentaci pomocí SDF jsou jednoduchá geometrická primitiva, jako je například koule nebo kvádr. Pro tato primitiva je možné vyjádřit vzdálenost od bodu matematickou rovnicí. Skládáním těchto primitiv můžeme vytvořit komplexní scény jako na obrázku 2.8.

$$d(\overrightarrow{a}, \overrightarrow{b}) : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^+ \tag{2.6}$$

$$SDF(\overrightarrow{x}) : \mathbb{R}^n \to \mathbb{R} \tag{2.7}$$

$$SDF(\overrightarrow{x}) = \begin{cases} -d(\overrightarrow{x}, \partial\mathcal{P}) & \overrightarrow{x} \in \mathcal{P} \\ d(\overrightarrow{x}, \partial\mathcal{P}) & \overrightarrow{x} \notin \mathcal{P} \end{cases} \tag{2.8}$$

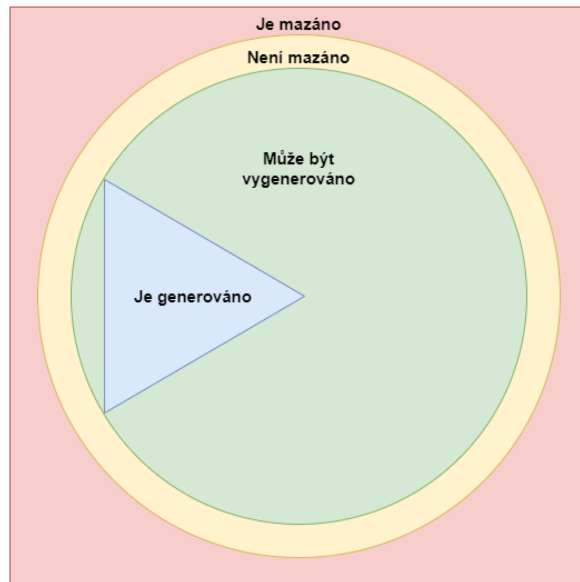
Úprava vstupu a výstupu je obdobná jako pro funkce šumů. Je vidět, že fukce orientované vzdálenosti není závyslá na parametru *seed*. To je možné vyřešit kombinací SDF s funkcemi šumů, například posunutím množiny $\mathcal{P}$ do středu regionu voroného diagramu.



Obrázek 2.8: Zleva: Ukázka trojrozměrných primitiv reprezentovaných pomocí SDF (zdroj: [16]) a komplexní scéna z nich vytvořená (zdroj: [16])

2.1.3 Proces generování v demonstračním frameworku

Jako v Minecraftu a podobných hrách je svět organizován do trojrozměrné kartézské mřížky. Každý voxel představuje krychlový region v prostoru, který je zarovnán do této mřížky. Z režijních důvodů (více v sekcích 2.2 a 2.3) je svět okolo hráče generován po blocích $16 \times 16 \times 16$ voxelů. Jaké bloky se mají vygenerovat je určováno na základě směru pohledu kamery a vzdálenosti od hráče. I když je svět generován po blocích, každý voxel je generován nezávisle



Obrázek 2.9: Oblasti generování a mazání bloků voxelů

na existenci ostatních. To znamená, že pokud je generování voxelu závislé na typu okolních voxelů, musí tyto okolní voxely být znovu vygenerovány, přestože už vygenerovány byly. Jak již bylo naznačeno v úvodní kapitole 1, realističnost a efektivita generování není primárním cílem práce. Proto jsou ve frameworku pro vygenerování scén dostupné pouze dvě funkce šumu. Konkrétně to je šum hodnoty (*Value noise* a voroného diagram, kde je pozice každého bodu $\overrightarrow{p} \in \mathcal{P}$ omezena buňkou kartézské mřížky. Konkrétní scény vygenerované pomocí těchto dvou funkcí jsou popsány v kapitole 4.1.2. Dále jsou definovány dvě vzdálenosti. První určuje maximální vzdálenost od hráče, kdy je možné ještě generovat bloky a druhá stanovuje prahovou vzdálenost, kdy jsou vygenerované bloky mazány z paměti. Mazání bloků umožní pohyb ve scéně aniž by došla paměť. Obě tyto vzdálenosti jsou odvozené od velikosti pohledového jehlanu kamery vykreslující scénu. Na obrázku 2.9 jsou znázorněné oblasti určené těmito vzdálenostmi. Béžová barva označuje oblast, kde bloky nejsou generované, ale pokud jsou vygenerované, nejsou mazány. V zelené oblasti je dovoleno bloky generovat. Po pohybu hráče se změní i oblasti a je nutné vyčnívající bloky (červená oblast) vymazat z paměti.

2.2 Reprezentace volumetrických scén

Jelikož je svět Minecraftu plně modifikovatelný a vygenerování jednotlivých voxelů a jejich vykreslovacích dat pro každý snímek by bylo výpočetně příliš

náročné, je nutné vygenerované voxely ukládat do paměti. Pro tuto úlohu jsou vhodnými strukturami trojrozměrné pole a oktalový strom (octree).

Pole

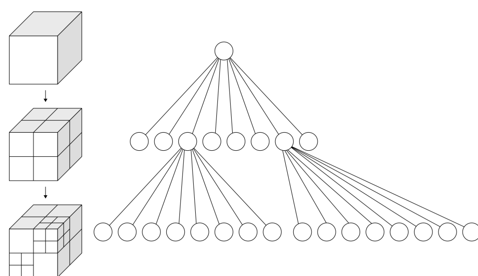
Ukládání jednotlivých voxelů do trojrozměrného pole by bylo vhodné, pokud by se svět často měnil. Pole totiž umožňuje konstatní časovou složitost pro náhodný zápis a čtení. Další výhodou je prostorová lokalita dat v poli. Při sekvenčním procházení, tak bude nastávat méně výpadků vyrovnávací paměti procesoru. Nevýhodou pole však je, že souvislý region prostoru (kámen, vzdych, apod.) je stále reprezentován jako jednotlivé voxely. Takový problém je možné vyřešit například run-length kompresí (RLE), pokud by pole bylo převedeno na pole jedné dimense. Touto kompresí by se ale zvýšila časová složitost přístupu k voxelu na určitých souřadnicích. Pokud bychom kompresí uchovávali data ve formátu $(v, l) \in \mathcal{V} \times \mathbb{N}$, kde $\mathcal{V}$ je množina typů voxelů a l je délka sekvence, mělo by vyhledání typu voxelu na konkrétním indexu časovou složitost $\mathcal{O}(n)$, kde n je počet voxelů v poli. Kdybychom si k tomu zaznamenávali i počáteční index sekvence $s \in \mathbb{N}$, tato složitost by byla $\mathcal{O}(\log(n))$.

Reprezentace celého herního světa jedním polem by zřejmě nebyla v souladu s možnostmi dnešního hardware. Svět je ale možné rozdělit do bloků (například $16 \times 16 \times 16$ voxelů, jako v sekci 2.1.3) a tyto bloky ukládat do hashovací tabulky, kde klíčem je pozice bloku ve světě. Vnitřní reprezentace voxelů v bloku by byla řešena polem.

Řídký oktalový strom voxelů

Řídký oktalový strom voxelů, anglicky sparse voxel octree (SVO) je stromová struktura, ve které má každý uzel buď osm potomků, nebo je listem a obsahuje data o regionu v prostoru (v našem případě data pro daný typ voxelu). S pomocí této struktury můžeme rozdělovat krychlové regiony v trojrozměrné kartézské mřížce na dalších osm regionů podle tří dělících rovin $\alpha = \overrightarrow{C}_x, \beta = \overrightarrow{C}_y, \gamma = \overrightarrow{C}_z$, kde $\overrightarrow{C}$ je vektor souřadnic středu děleného regionu. V opačném případě je možné regiony reprezentované listy stromu spojovat, pokud všech osm listů sdílí stejné vlastnosti. Obrázek 2.10 zobrazuje dělený region a odpovídající stromovou strukturu. Výšku stromu určuje nejvyšší počet dělení.

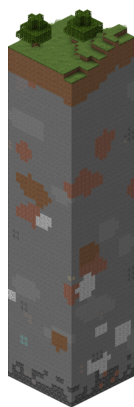
Operace zápisu a čtení mají jako u většiny stromových struktur časovou



Obrázek 2.10: Zleva: Dělený region, odpovídající oktalový strom výšky 2. Zdroj:[13]

složitost $\mathcal{O}(\log(n))$. Nevýhodou je nízká prostorová lokalita uložených dat. Při procházení SVO tedy bude nastávat více výpadků vyrovnávací paměti procesoru než u pole. Oproti poli má však SVO lepší prostorovou složitost a použitá komprese dat je v trojrozměrných scénách užitečnější než RLE komprese. Můžeme například velký krychlový blok voxelů reprezentující kámen vykreslit jako jednu velkou krychli, namísto zpracovávání každého voxelu zvlášť.

2.2.1 Dělení světa v Minecraftu



Obrázek 2.11: Plná část sloupce voxelů. Zdroj: [10, příspěvek "Chunk"]

K Minecraftu jsem bohužel nebyl schopen najít oficiální dokumentaci. Nicméně existuje wikipedie vytvořená komunitou Minecraftu [10]. Na té je uvedeno [10, příspěvek "Chunk"], že Minecraft člení svět do sloupců o podstavě 16×16 voxelů a vysokých 256 voxelů (viz. Obrázek 2.11). Tyto sloupce jsou organizovány do dvourozměrné kartézské mřížky. Svět je generován po těchto sloupcích podle vzdálenosti od hráče. Slopce jsou dále děleny do bloků $16 \times 16 \times 16$ [10, příspěvek "Chunk format"], kde bloky obsahující neprázdné voxely jsou ukládány do pole.

Minecraft wikipedie se nezmiňuje, proč je svět takto členěn. Důvodem může být větší kontrola nad generováním a ukládáním světa redukcí na dvourozměrnou mřížku sloupců. Ing. Daniel Čejchan [1] ve své práci, kde také vytváří herní engine pro hry podobné Minecraftu, tento model aplikuje. Hlavními faktory, podle kterých bylo rozhodnuto použít tento model byla možnost realizace globálního osvětlovacího modelu, kde je světlo propagováno skrze celý sloupec, což by nebylo možné pokud by svět nebyl omezen po vertikální ose, dále byl rozhodující formát informací jednotlivých

voxelů, z optimalizačních důvodů by velikost stran sloupce měla být v násobcích dvou a také by sloupec neměl být příliš malý kvůli nárokům na jeho režii. Více informací lze nalézt v jeho práci v kapitole 4.1 "Reprezentace světa".

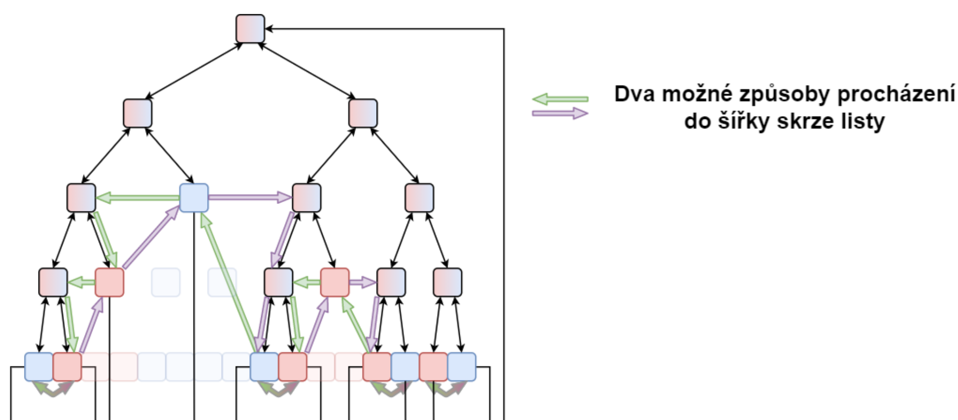
2.2.2 Reprezentace scény ve frameworku

Pro reprezentaci scény ve frameworku jsou použity oktalové stromy. To by mělo přinést snížení paměťové náročnosti frameworku. Dále se sníží počet vykreslovaných voxelů, jelikož je možné voxely reprezentující stejné materiály, jako jsou například kámen nebo hlína, sloučit v jeden a vykreslit jako jednu větší krychli.

Struktura, kterou je scéna reprezentována v paměti je rozdělena do tří úrovní:

1. Blok voxelů
2. Oktalový strom bloků voxelů (BO)
3. Trojrozměrná kartézská mřížka BO

Blok voxelů



Obrázek 2.12: Struktura octree voxelů v bloku.

Blok voxelů, zkráceně blok, představuje krychlový region prostoru ve scéně o velikosti $16 \times 16 \times 16$ voxelů. Blok je nejmenší generovanou a vykreslovanou jednotkou scény. Jednotlivé voxely v bloku jsou reprezentovány pomocí oktalového stromu voxelů.

Velikost bloku byla zvolena na základě několika faktorů:

- Voxely budou uloženy v oktalovém stromu
  - Blok musí být krychle
  - Velikost v každé ose musí být mocnina dvou
- Volba velikost bloku by měla brát v úvahu nutnost vygenerování celého bloku najednou
- Po vygenerování voxelů nebo jejich modifikaci je nutné shromáždit data k vykreslení (více v sekci 2.3) v rozumném čase

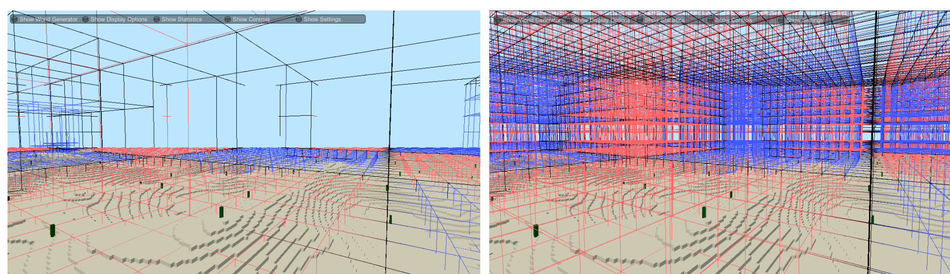
Jelikož jsou jednotlivé voxely uloženy v oktalovém stromu, je nutné, aby každý obsahoval identifikátor typu voxelu a informaci o tom, jestli je možné daný typ slučovat. Například truhly není možné sloučit, protože každá má i svůj vnitřní stav. Pro účely vykreslování (vygenerování grafu viditelnosti 2.3.4) je vyžadováno procházet blokem do šířky a to pouze skrze průhledné voxely. Proto listy oktalových stromů voxelů odkazují na blok, ve kterém se nachází, aby bylo možné se ho dotázat s jakým voxelem sousedí. Každý uzel stromu dále obsahuje odkaz na rodičovský uzel, aby bylo možné se stromem pohybovat oběma směry. Na obrázku 2.12 je znázorněná struktura pro jednorozměrnou scénu s dvěma typy voxelů.

Okralový strom bloků voxelů

Oktalový strom bloků voxelů (BO) je datová struktura, v níž se ukládají jednotlivé bloky voxelů. Maximální výška stromu je 4. BO tedy obsahuje $16 \times 16 \times 16$ bloků voxelů. Faktory podle nichž byla tato velikost zvolena:

- Bloky jsou uloženy v oktalovém stromu
- Kořen oktalového stromu bude shromažďovat informace o zastiňujících stěnách bloků v jeho podstromu, které jsou v každém snímku využívány (více v sekci 2.3)

Tyto oktalové stromy bloků jsou uspořádány do trojrozměrné kartézské mřížky a jsou uloženy do hašovací tabulky, kde klíčem je pozice s nejmenšími



(a) : se slučováním **(b) :** bez slučování

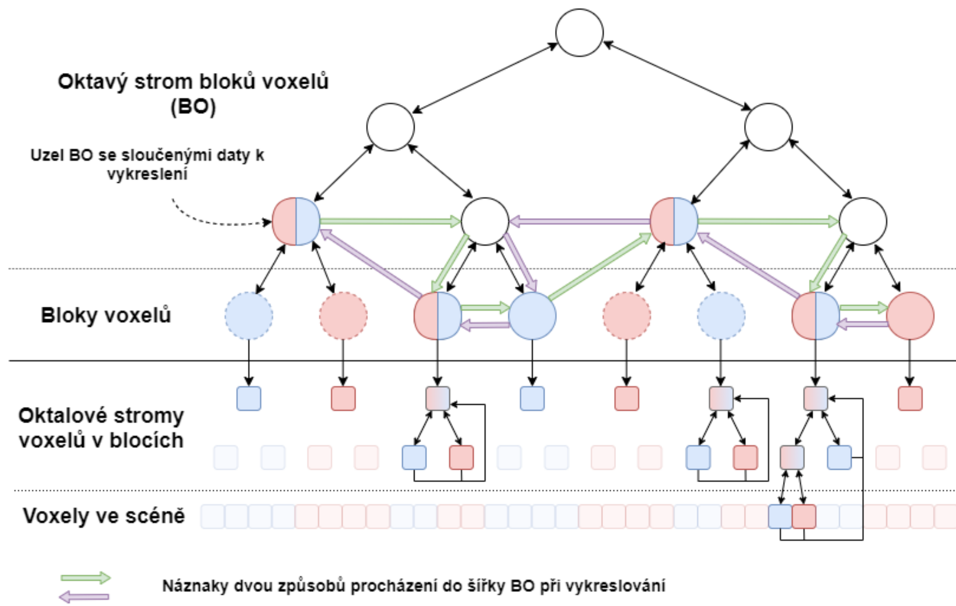
Obrázek 2.13: Vliv slučování na hierarchii scény Pouště

souřadnicemi na všech osách. Ve dvourozměrném případě by to byla pozice levého dolního rohu čtverce.

Vzhledem k technice vykreslování (kapitola 2.3) je opět nutné procházet BO do šířky. Procházení je podobné jako pro SVO v blocích voxelů. Zde však listy nejsou slučovány, kvůli jednodušší režii. Jednotlivé uzly jsou slučovány pouze ve smyslu dat k vykreslovaní. Odkaz na souseda v daném směru tedy může ukazovat na uzel ve vyšší úrovni stromu, který není listem, a opačně. Uzel BO je sloučen na základě tří kritérií:

- Všichni potomci mají vygenerovaná data k vykreslení
- Data k vykreslování všech potomků je možné sloučit
- Graf viditelnosti (popsán v 2.3.4) je shodný pro všechny potomky

Pokud jsou splněna všechna tato kritéria, jsou data pro vykreslení sloučena a uzel je nastaven jako soused pro sousední uzly s daty k vykreslení scény. Slučováním jednotlivých uzlů by měly výrazně zlepšit výkon vykreslování, jelikož některé bloky po vygenerování neobsahují žádná data pro vykreslení nebo jen velmi malé množství. Na obrázku 2.13 je vidět dopad na konečnou hierarchii procházené scény, pokud je slučování dat v uzlech povoleno, či nikoli. Konečná struktura pro vnitřní uzel tedy obsahuje data o svých osmi potomcích, rodiči a grafu viditelnosti. Dále může obsahovat data k vykreslování, pokud byla splněna všechna kritéria výše a podstrom byl sloučen, a seznam sousedních uzlů ve stejné úrovni. Když je uzel listem obsahuje namísto seznamu potomků odkaz na blok voxelů. Na obrázku 2.14 je zobrazena tato stromová struktura pro prostor jedné dimense. Jednotlivé barvy reprezentují odlišné typy voxelů.

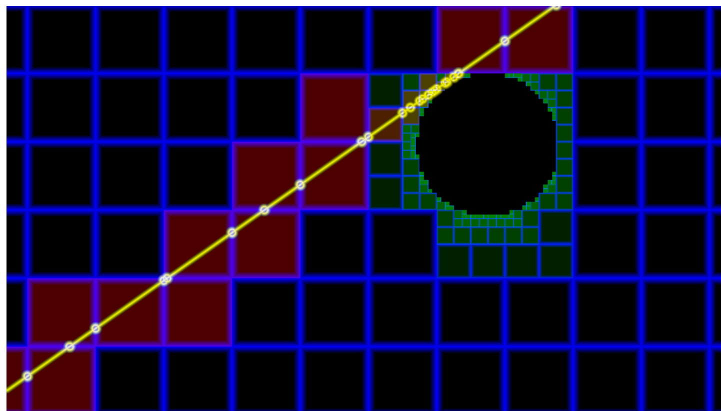


Obrázek 2.14: Hierarchie scény

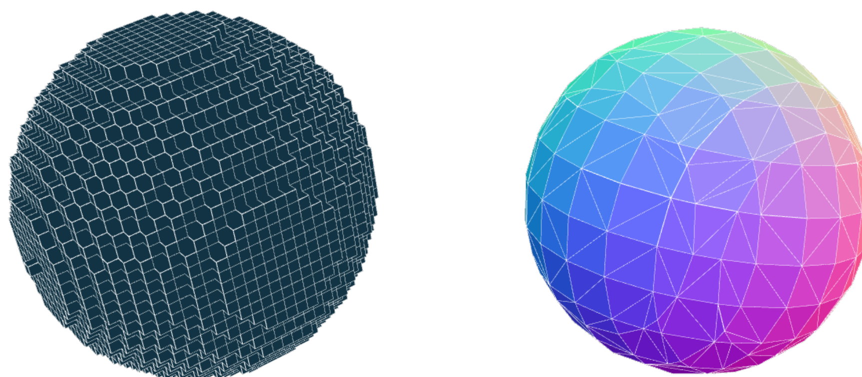
2.3 Vykreslování

Techniky vykreslování volumetrických dat je podle Kaufmana a spol. [5] a také podle Elvinse [2] možné rozdělit do dvou kategorií, vykreslování skrze geometrická primitiva (*surface-fitting*) a metody přímého vykreslování (*direct volume rendering*, DVR).

2.3.1 Metody přímého vykreslování



Obrázek 2.15: Vrhání paprsku dvourozměrnou scénou.



Obrázek 2.16: Hraniční reprezentace koule vytvořená z krychlí (vlevo) a pomocí techniky *marching cubes*

Metody přímého vykreslování využívají přímo volumetrická data k jejich vykreslení. Snad nejznámější technikou je metoda vrhání paprsku (tzv. *ray casting*. Při této metodě je od kamery vrhán paprsek scénou a po určitých frekvencích jsou vzorkovány vlastnosti prostoru na souřadnicích, jimiž paprsek prochází, Od nich je odvozena konečná barva. Během průchodu scénou se paprsek může odrážet od povrchu nebo měnit směr na rozhraní dvou materiálů. Výhodou těchto metod je možnost realizace globálního osvětlení a realističtější koncový snímek. Nicméně pro dosažení takové realističnosti je zapotřebí pro jeden snímek vrhat mnoho paprsků pro každý pixel. Proto je tato technika vhodná spíše v aplikacích, kde se upřednostňuje kvalita snímku nad rychlostí vykreslení, například ve filmovém průmyslu. Obrázek 2.15 znázorňuje průchod paprsku dvourozměrnou scénou, bíle body představují Vzorkovací pozice.

2.3.2 Vykreslování skrze geometrická primitiva

Při vykreslování volumetrických dat skrze geometrická primitiva je z volumetrických dat extrahována jejich hraniční reprezentace. Tato hraniční reprezentace je následně vykreslována jako běžná polygonová mřížka. Nejjednodušším příkladem vytvoření takové hraniční reprezentace je shromáždění neprůhledných voxelů a každou stranu, která která sousedí s průhledným voxelem, vykreslit jako čtverec. Touto metodou je možné vykreslit svět podobný Minecraftu. Existují další techniky extrakce hraniční reprezentace (například tzv. *marching cubes*, Obrázek 2.16), které se soustředí na různé aspekty, jako je například "hladkost"terénu. Více se o nich může čtenář dozvědět v [25].

Vykreslování skrze geometrická primitiva je pro hry vhodnější. Je totiž možné hraniční reprezentaci vygenerovat po tom, co se vygenerují volume-

trická data a dále s ní zacházet jako s normální polygonovou mřížkou. Pro obsáhlé scény je však vykreslování velkého množství geometrických primitiv problém. Potom je nutné scénu generovat po menších regionech (například blocích představených v sekci 2.2) a rozhodovat o tom, který z nich se má v daný okamžik vykreslit.

2.3.3 Určování viditelnosti objektů ve scéně

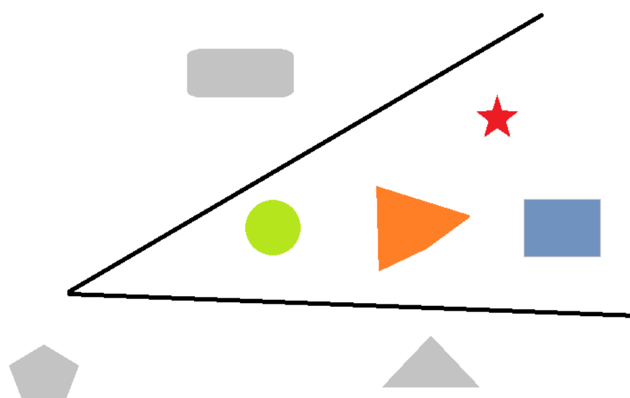
Při vykreslování scény jsou využívány důležité výpočetní zdroje, které by se mohly lépe využít na zvýšení detailu nebo na snížení doby pro vykreslení jednoho snímku. Proto je vhodné vykreslovat pouze ty objekty, které jsou viditelné, tedy po vykreslení zabírají alespoň jeden pixel obrazovky.

Pohledový jehlan

Zorné pole lidského oka je možné přirovnat ke kuželu, jehož osa reprezentuje směr, ve kterém se oko dívá. Jelikož je monitor nebo textura, do které se scéna vykresluje, většinou hranatá místo kuželu je zorné pole kamery reprezentováno jehlanem. Tento jehlan je dále omezen dalšími dvěmi rovinami reprezentující nejmenší a největší možnou vykreslovací vzdálenost. Jedná se tedy komolý jehlan (po zbytek textu však pouze jehlan). Během vykreslování je možné testovat, jestli se alespoň část vykreslovaného primitiva nachází v zorném poli (tedy jehlanu) kamery. Pokud ne, nemusí být dále zpracováváno. Vhodné by bylo aplikovat tuto techniku na celé objekty, aby bylo o jejich viditelnosti rozhodnuto co nejdříve. Například testovat jejich konvexní obálky na přítomnost v pohledovém jehlanu. Na obrázku 2.17 je vizualizován princip rozhodování o viditelnosti podle pohledového jehlanu. Barevné objekty jsou viditelné, zašedlé nikoli.

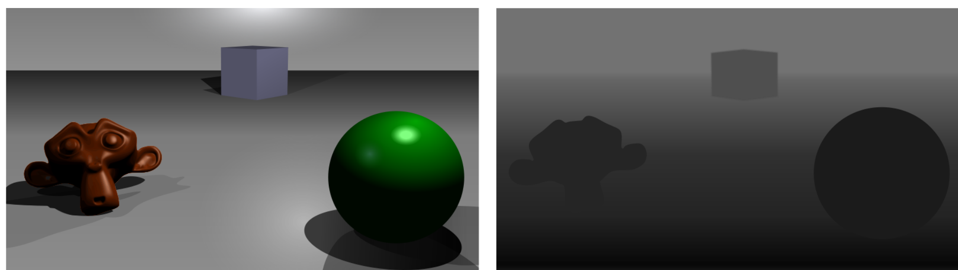
Z-Buffer a dotazy na zastínění

Na úrovni jednotlivých pixelů je možné rozhodovat o viditelnosti pomocí tzv. *Z-Bufferu*. Z-Buffer je datový buffer, který reprezentuje hloubku (vzdálenost od kamery) pixelu (viz. Obrázek 2.18). Před vykreslováním je Z-Buffer inicializován na největší možnou vzdálenost. Během vykreslování se můžeme dotázat, jaká je v Z-Bufferu pro pixely, na které se promítl vykreslovaný polygon, uložená hloubka. Pokud je vykreslovaný pixel blíže ke kameře (hloubka pixelu je



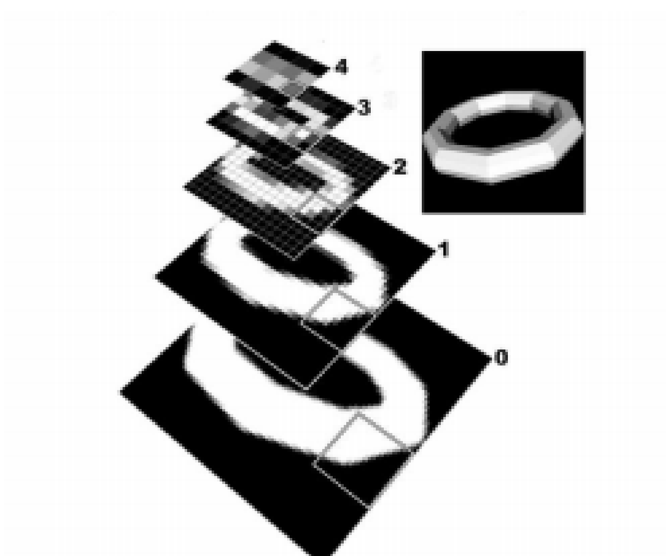
Obrázek 2.17: Princip pohledového jehlanu

menší než hloubka uložená v Z-Bufferu), potom je pixel vykreslen a hloubka pixelu v Z-Bufferu je aktualizována. Pokud je hloubka pixelu větší než v Z-Bufferu, pixel není nutné dále zpracovávat, protože není vidět. Aby bylo toto brzké odmítnutí pixelu co možná nejefektivnější, je ideální vykreslovat objekty "od předu dozadu", tedy směrem od kamery.



Obrázek 2.18: Vykreslená scéna (vlevo) a korespondující Z-Buffer. Zdroj: [24]

S Z-Bufferem také souvisí hardwarové dotazy na zastínění, tzv. *occlusion queries*. Během nich je objekt promítnut na jednotlivé pixely a jejich hloubky jsou otestovány oproti těm v Z-Bufferu. Výstupem je informace, jestli alespoň jeden pixel prošel tímto testem. Toho je možné opět využít pro otestování viditelnosti pouhých konvexních obálek objektů s komplexní geometrií. Pokud testem na hloubku neprošel ani jeden pixel, na který se obálka promítla, neprojde jím ani konkrétní geometrie objektu. Nevýhodou této metody je fakt, že pro velké objekty se musí testovat velké množství pixelů. Tento problém je možné vyřešit pomocí *hierarchického Z-Bufferu*



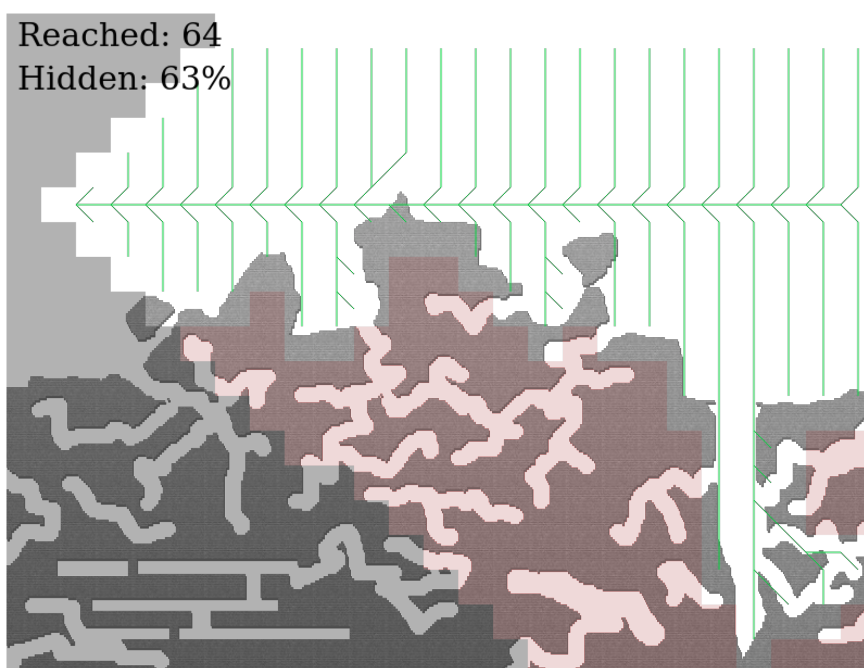
Obrázek 2.19: Hierarchický Z-Buffer a vykreslená scéna, jejíž hloubku reprezentuje. Zdroj: [6]

Hierarchický Z-Buffer

Hierarchický Z-Buffer [12] (HiZ) lze považovat za rozšíření běžného Z-Bufferu. Je tvořen mip-pyramidou původního Z-Bufferu. Každá úroveň je odvozena z té předchozí a to tak, že do každého pixelu vyšší úrovně pyramidy (2x menší rozlišení než ta předchozí) se přiřadí hodnota reprezentující největší hloubku ze čtyř pixelů nižší úrovně, které tento pixel pokrývá (viz. Obrázek 2.19). Process se opakuje, dokud úroveň neobsahuje pouze jeden pixel.

Výhodou HiZ je snížení počtu testovaných pixelů na maximálně čtyři. Nevýhodou však je větší výskyt *false-positive* testů a to v případech, kdy testovaný objekt nepokrývá celý region testovaných pixelů.

Toto byly metody, které jsou využity v procesu vykreslování navrhovaného frameworku. Existuje mnohem více technik pro rozhodování viditelnosti. Pokud tato problematika čtenáře zajímá, může se o ní dozvědět více například v disertační práci Ing. Víta Kovalčíka, Ph.D. [6]



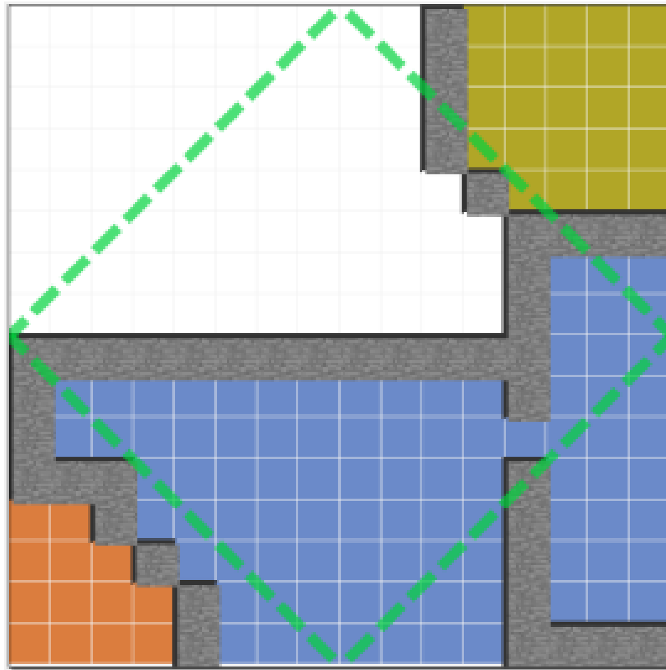
Obrázek 2.20: Průchod scénou podle kriterií uvedených v 2.3.4. Zdroj: [4]

2.3.4 Vykreslování světa v Minecraftu

Podobně jako u členění světa neexistuje oficiální zpráva o technice vykreslování světa v Minecraftu. Nicméně vývojář Minecraftu Tommaso Checchi [4] ve svém blogu tento proces nastiňuje. V průběhu je popsána technika *grafu viditelnosti*, jež je použita i v této práci, jako vylepšení v oblasti určování viditelnosti. Tommaso tvrdí, že svět je vykreslován po blocích $16 \times 16 \times 16$ voxelů. Těmito bloky je procházeno do šířky podle následujících kritérií (viz Obrázek 2.20):

1. Sousední blok je dál od kamery než současný
2. Sdílená strana se sousedním blokem je dosažitelná ze strany, kterou jsme se do současného bloku dostaly (o tom nese informaci graf viditelnosti, více v sekci níže)
3. Sousední blok je v zorném poly kamery

Pokud sousední blok splní tato kritéria, je vykreslen a je z něj expandováno dál. Procházení je inicializováno od bloku v němž se nachází kamera. Z obrázku 2.20 je vidět, že využití grafu viditelnosti nezabraňuje průchodu do částí, které nejsou vidět (rokle na obrázku vpravo). Tomu je částečně zabraňováno nejrůznějšími filtry, jež jsou zrozebrány v blogu [4].



Obrázek 2.21: Graf viditelnosti pro dvourozměrný blok 16 × 16 voxelů. Zdroj:[4]

Graf Viditelnosti

Mějme množinu stran pravoúhlého bloku voxelů $\mathcal{S} = \{s_0, s_1, s_2, s_3, s_4, s_5\}$. Graf viditelnosti je dvojice $\mathcal{G} = (\mathcal{S}, \mathcal{E})$, kde $\mathcal{E}$ je množina dvojic $\mathcal{E} \subseteq \{(x, y) \mid (x, y) \in \mathcal{S}^2,\ x \neq y\}$. $(x, y) \in \mathcal{E}$ právě tehdy, když v bloku existuje sekvence sousedících průhledných voxelů, jež strany $x, y \in \mathcal{S}$ spojuje. Jinými slovy, pokud je možné se procházením průhledných voxelů v bloku dostat ze strany x na stranu y, potom $(x, y) \in \mathcal{E}$. Konstrukce tohoto grafu je realizována tzv. *flood fill* algoritmem skrze průhledné voxely (viz. Obrázek 2.21, barvy znázorňují jednotlivé průchody a zelené čáry spojené hrany). Tato struktura umožňuje zmíněné procházení do šířky. Díky ní je možné procházení předčasně ukončit, pokud žádná strana není z příchozí strany dosažitelná. Blog uvádí, že tato technika byla zavedena především kvůli zbytečnému vykreslování podzemních struktur, které byli zastíněny terénem povrchu a snižovali výkon aplikace na mobilních zařízeních.

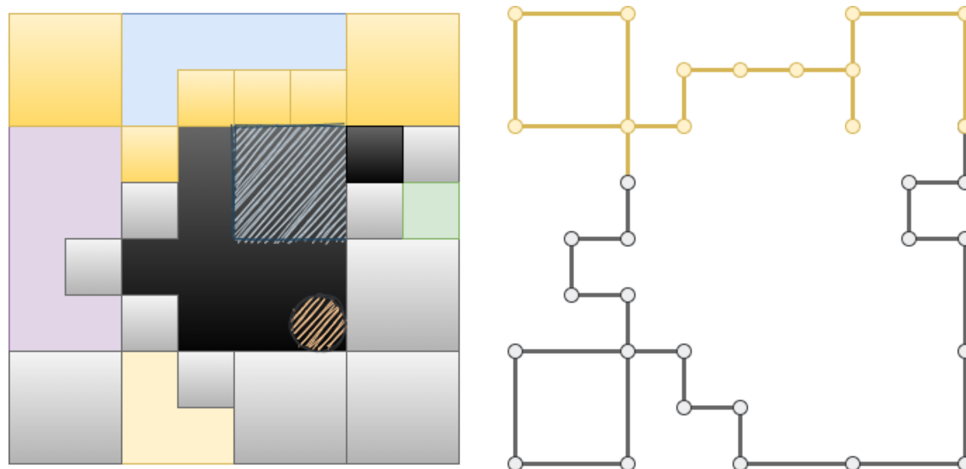
2.3.5 Vykreslování scény ve frameworku

Vykreslování scény v demonstračním frameworku je založeno na podobném principu jako v Minecraftu. Scénou je procházeno do šířky od kamery s

využitím grafu viditelnosti. Aby bylo eliminováno procházení částmi, které nejsou vidět, jsou jednotlivé krychlové bloky voxelů testovány na viditelnost pomocí hierarchického Z-Bufferu.

Shromáždění dat k vykreslování

Data k vykreslení jednotlivých bloků jsou shromažďována v průběhu generování grafu viditelnosti. Algoritmus flood fill je volán pro každý průhledný voxel na stranách bloku. Každý voxel, jež je během algoritmu navštíven, je dotázán na jeho data k vykreslení. Ty mohou záviset na průhlednosti stran sousedních regionů (především voxely materiálů, které okupují celý krychlový region). Je tedy nutné aby data voxelů sousedních bloků byla vygenerována. Uvnitř bloku je tato informace propagována od listů oktalového stromu voxelů v bloku až do jeho kořene. Díky procházení od jednotlivých stran je zaručeno, že geometrie "uzavřená"v bloku, tedy obklopená neprůhlednými voxely nebude zbytečné generována. Obrázek 2.22 zobrazuje na dvourozměrném příkladu oblasti jednotlivých průchodů a konečnou geometrii bloku. Zde se předpokládá, že sousední bloky voxelů mají průhledné strany. Černá oblast vyznačuje prostor "uzavřený"v bloku. Jednotlivé barevné regiony zobrazují jednotlivé průchody flood fill algoritmu.



Obrázek 2.22: Proces vygenerování geometrie bloku voxelů.

Pokud je geometrie bloku velmi málo, je vhodné ji sloučit s ostatními bloky v podstromy BO. Sníží se tím čas režiji při vykreslování. Například kdyby každý blok obsahovat pouze jeden plný voxel, je nesmyslné scénu procházet a vykreslovat po jednotlivých krychlích. Je-li to tedy možné jsou tato data sloučena. Při slučování je nutné respektovat nutnost data opět rozdělit při modifikaci scény. Kritéria pro slučování byla popsána v kapitole 2.14.

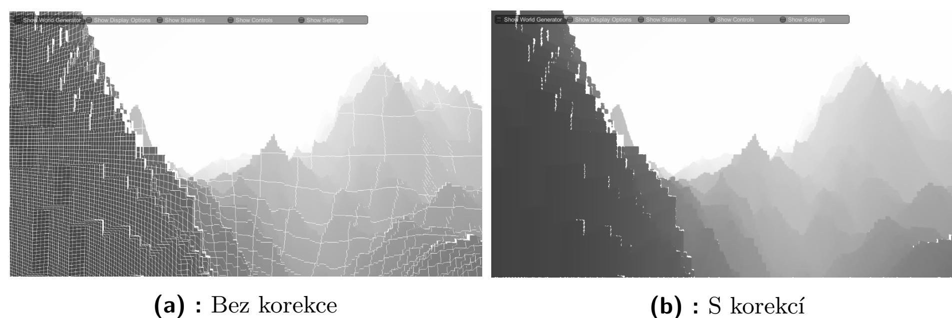
Test viditelnosti pomocí hierarchického Z-Bufferu

Při testu viditelnosti je každý blok voxelů je každý blok voxelů promítnut na pixely hierarchického Z-Bufferu a otestován oproti jejich hloubce. V běžných scénách je to uskutečnitelné manuálním identifikováním velkých neprůhledných překážek během jejího vývoje. Ty vykreslit jako například kvádry nebo koule do hierarchického Z-Bufferu a oproti této informaci testovat ostatní jemnější prvky scény. Jelikož je však v našem případě scéna dynamická a každá část světa se může měnit, není možné tyto překážky identifikovat. Také opětovné vytváření mip-piramidy je časově náročné. Není tedy možné po vykreslení každého bloku, HiZ aktualizovat. Ve frameworku se tento problém snažím vyřešit kombinací dvou přístupů:

1. Transformace pixelů Z-Bufferu z předchozího snímku
2. Vykreslení blízkého okolí

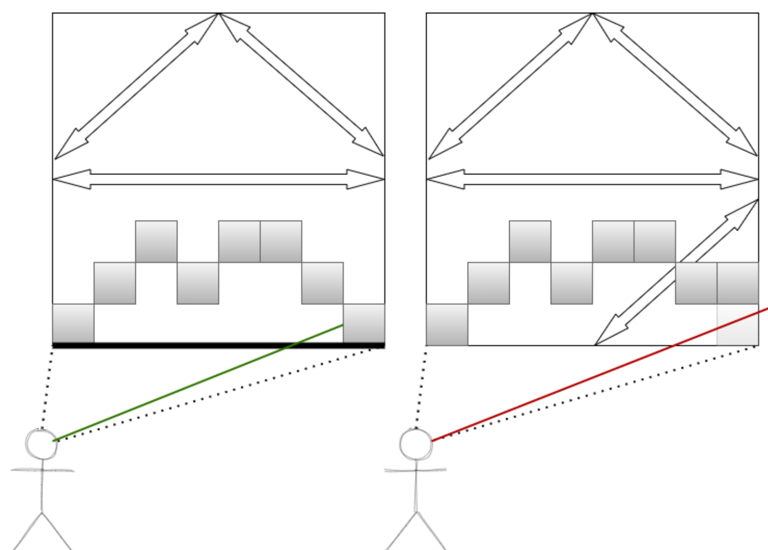
Transformace pixelů Z-Bufferu z předchozího snímku. Tento přístup využívá informace o hloubce pixelu z předchozího snímku. Po transformaci kamery pro současný snímek jsou pro každý pixel Z-Bufferu z předchozího snímku vypočítány souřadnice bodu ve scéně pomocí transformační matice kamery použité pro vykreslení předchozího snímku a následně je promítnut s pomocí současné transformační matice na jiné souřadnice pixelu, kam je zapsána jeho nová hloubka. Pokud je změna polohy a rotace kamery mezi snímky dostatečně malá, dostaneme rozumný odhad pro hloubky scény. Přesto se ale může stát, že se některé pixely původního ZBufferu po transformaci překrývají a vznikají malé "trhlinky". Tento problém je částečně vyřešen jednoduchým filtrem, jež prozkoumá osmi-okolí "prázdného"pixelu (pixelu s největší možnou hloubkou) a pokud se v něm vyskytuje menší počet "prázdných"pixelů než stanovené množství, je hodnota pixelu nastavena na průměr ze všech "neprázdných"pixelů z osmi-okolí. Na obrázku 2.23 je ukázka transformovaného hierarchického ZBufferu bez aplikace této korekce a s ní. Bohužel je možné vidět, že část textury stále obsahuje trhliny.

Využití informace grafu viditelnosti. Graf viditelnosti (sekce 2.3.4), obsahuje informace o "propojenosti"stran. Pokud je nějaká strana zcela nedosažitelná z ostatních znamená to, že stranu můžeme uvažovat jako neprůhlednou stěnu (viz. Obrázek 2.24). Tyto stěny jsou ukládány v kořeni oktalového stromu bloků voxelů. Před jemným průchodem scénou a samotným vykreslováním je scénou procházeno skrze kořeny BO a jsou vykresleny tyto neprůhledné stěny. Tato metoda by měla dále zkvalitnit informaci o hloubce scény.



(a) : Bez korekce **(b) :** S korekcí

Obrázek 2.23: Transformovaný hierarchický ZBuffer bez korekce a s korekcí.



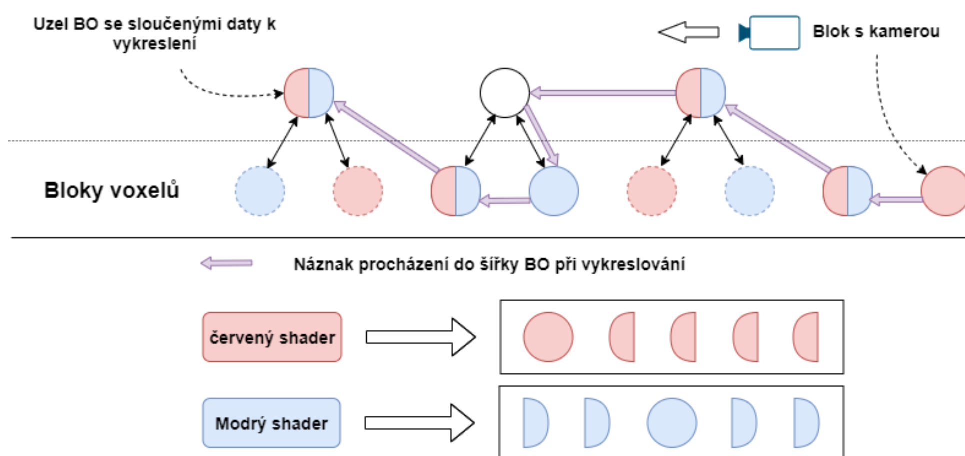
Obrázek 2.24: Neprůhlednost stěny v grafu viditelnosti

Průchod scénou a vykreslování

Průchod scénou skrze uzly, v nichž se nacházejí data k vykreslení je nastíněn v obrázku 2.25. Každý uzel má daného souseda pro každou stěnu. Jediný zádrhel nastává, je-li uzel, ze kterého se expanduje dál ve vyšší úrovni stromu než sousední uzly s daty k vykreslení. Tento uzel má jako souseda stanoven uzel ve stejné úrovni. Pro ten jsou do hranice procházení do šířky vloženy uzly podstromu, které již obsahují data k vykreslení a zároveň, sdílí stranu se s uzlem současným. Algortimus se velmi podobá algoritmu 1 s tím rozdílem, že nespojujeme sousedy, ale rozšiřujeme hranici procházení. Jednotlivé uzly musí splňovat stejná kritéria jako při vykreslování světa v Minecraftu. Dále musí projít testem na hloubku v hierarchickém Z-Bufferu. Procházení je inicializováno od bloku, v němž se nachází kamera. Data k vykreslení jednotlivých uzlů jsou tříděna podle shaderů, které je vykreslují. Po projití scény jsou data po jednotlivých shaderech vykreslena.

Algorithm 1: Algoritmus slinkování listů v oktalovém stromě.

```
Function connect(OctreeNode a, OctreeNode b, Side aSide)
    if a.isLeaf and b.isLeaf then
        a.Neighbours[aSide] = b;
        b.Neighbours[opposite(aSide)] = a;
    else if b.isLeaf then
        for OctreeNode ch in a.getChildrenOnSide(aSide) do
            connect(ch, b, aSide);
        b.Neighbours[opposite(aSide)] = a;
    else if a.isLeaf then
        for OctreeNode ch in
         b.getChildrenOnSide(opposite(aSide)) do
            connect(a, ch, aSide);
        a.Neighbours[aSide] = a
    else
        for OctreeNode ch in
         b.getChildrenOnSide(opposite(aSide)) do
            connect(a.getChildNeighbouringWith(ch), ch, aSide);
```



Obrázek 2.25: Procházení scénou

Kapitola 3

Implementace

V této kapitole je popsána implementace navrženého frameworku v herním enginu Unity ve verzi 2020.2.1f1.

3.1 Herní engine Unity

Herní engine Unity je jeden z nejpopulárnějších herních enginů. Svým uživatelům nabízí komplexní vývojové prostředí umožňující vývoj her většiny existujících žánrů. Vývojové prostředí unity se skládá z mnoha komponent. Pro účely této práce jsou však využity pouze tři. Konkrétně:

- Skriptovací API
- Multithreading
- Vykreslovací engine

Skryptovací API

Pomocí skriptovacího API nabízeným Unity může uživatel vytvářet skripty v jazyce C#. Skripty lze komunikovat s interními systémy Unity a vytvářet tak

doplňující funkcionality editoru nebo přímo herní mechaniky. V této práci jsou skripty použity k vytváření přímo herních mechanik a to ke generování a vykreslování voxelových scén. Takové skripty v drtivé většině dědí od třídy `MonoBehavior`, která umožní Unity editoru skript identifikovat. Další důležitou třídou je třída `GameObjet`. Instance třídy `GameObject` je možné umisťovat do scény, kde se hra má odehrávat. K těmto objektům je možné připojovat komponenty ovlivňující jeho chování. Těmito komponentami jsou právě třídy dědící od třídy `MonoBehavior`. Pokud je objektu komponenta přiřazena již v editoru, je pro ni vygenerováno uživatelské rozhraní a vývojář je tak schopen editovat proměnné komponenty přímo v editoru. Instance třídy `MonoBehavior` jsou dále součástí herní smyčky Unity enginu, takže je možné jejich stav pravidelně aktualizovat, například ve funkci `Update()`. Více o herní smyčce Unity lze nalézt v dokumentaci [20, stránka "Order of execution for event functions"].

Multithreading

Systém pro snadné vytváření úloh určené k paralelní exekuci. Unity přijme požadavek na vykonání a sama jí přidělí výpočetní zdroje v podobě vlákna. Stěžejní je pro tento systém rozhraní `IJob` obsahující metody `Execute()`, v níž je definovaná funkcionalita, která se má paralelně vykonat, a `Schedule()`, jež vyšle požadavek o vykonání konkrétní úlohy. Tento systém má tu nevýhodu, že Unity z bezpečnostních a optimalizačních důvodů nedovoluje, aby se v paralelních úlohách pracovalo s pamětí spravované GarbageCollectorem. Práce se spravovanou pamětí v paralelních úlohách je avšak i tak možná s využitím vzoru prezentovaného v blogu Marnielle Lloyd Estrady [3], jež je použit i v implementaci navrženého frameworku. Pokud je tento vzor využit musí si vývojář dávat pozor na úskalí paralelního programování v podobě současného zápisu a čtení apod. Více informací o vícevláknových systémech v Unity je možné najít v dokumentaci [20, stránka "C# Job System"].

Vykreslovací engine

Unity nabízí svým uživatelům širokou škálu funkcionalit v oblasti vykreslování. Pro účely této práce je však využita pouze ta nejzákladnější. Pro vykreslování není použit žádný vykreslovacích frameworků přítomných v Unity. Důvody vedoucí k tomuto rozhodnutí, jsou uvedeny v odstavci **Kamera** 3.1. Jednotlivé třídy, jež jsou při vykreslování použity, jsou popsány níže.

Kamera. Kamera je definovaná v komponentou `Camera`. Obsahuje veškerou potřebnou funkcionality pro definování pohledového jehlanu a jeho orientaci. Vykreslování probíhá v drtivé většině vyvolání metody `Camera.Render()` ať už interně, či přímo skrze skript. Bouhu6el vyvolání této metody sebou nese množství skryté funkcionality, jež není zdokumentovaná. Jedním takovým příkladem je, že Unity interně vykonává vyhodnocování viditelnosti objektů na základě pohledového jehlanu kamery [20, stránka "Camera"]. Tento proces není možné zakázat. Toto byl primární důvod, proč bylo pro účely měření upuštěno od využití vykreslovacích frameworků nabízených Unity.

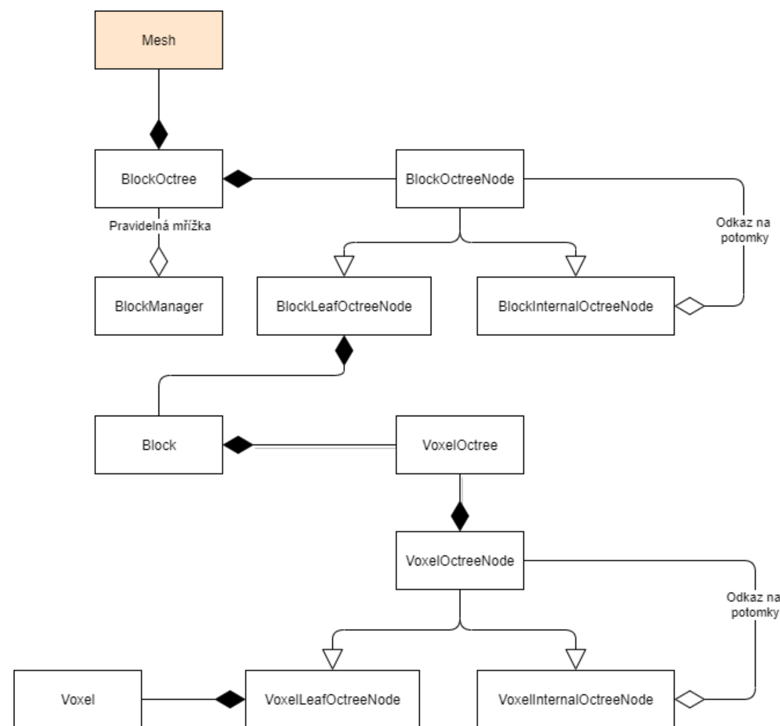
Graphics a GL. Třídy `Graphics` a `GL` nabízí multiplatformní grafické rozhraní pro vykreslování a vyvolávání úkonů na grafické kartě. Jednou z nejpoužívanějších funkcí je `Graphics.Blit()`, která slouží k přenosu dat mezi texturami skrze volitelný shader. Toho je využíváno především při tvorbě hierarchického ZBufferu, kdy je potřeba přenášet data z jedné úrovně mip-pyramidy do druhé. Kompletní seznam funkcí rozhraní lze nalézt v dokumentaci [19, stránka "Graphics"a "GL"].

RenderTexture. Třída `RenderTexture` představuje textury, do níž je možné vykreslovat. Obsahuje dva buffery a to `ColorBuffer` a `DepthBuffer`. Podle formátu textury je určen buffer, z něhož je čteno. Zapisováno může být do obou bufferů. Více informací k třídě `RenderTexture` lze nalézt v dokumentaci [19, stránka "RenderTexture"].

Mesh. Třída `Mesh` je používána k reprezentaci polygonových mřížek. Jelikož je framework schopen vygenerovat velké množství geometrie, je dobré poznamenat, že instance třídy `Mesh` ukládají 1:1 kopii dat na procesoru vůči těm na grafické kartě. Paměť RAM se tedy může rychle vyšplhat na vysoké hodnoty. Pokud je tato vlastnost zakázána, není možné data získat zpět pro manipulaci. Kompletní rozhraní třídy `Mesh` je k nahlédnutí v dokumentaci [19, stránka "Mesh"].

3.2 Reprezentace světa

Na obrázku 3.1 je lze vidět, v jakých třídách je implementována reprezentace světa. Celá hierarchie je manipulována ve třídě `BlockManager`, která je zodpovědná za vytváření bloků voxelů a oktalových stromů bloků voxelů.



Obrázek 3.1: Objektový diagram reprezentace světa.

Jak již bylo nastíněno v kapitole, 2.14 oktalové stromy voxelů jsou organizovány do pravidelné mřížky. Ta je realizována instancí třídy `Dictionary`, kde klíčem jsou globální souřadnice daného oktalového stromu bloků voxelů. Třída `BlockManager` dále rozvrhuje úlohy generování voxelů v blocích a shromažďování dat k vykreslení.

Oktalový strom bloků voxelů

Třída `BlockOctree` slouží pro obalení oktalového stromu bloku voxelů představeného v sekci 2.14. Každá instance má manažerem jednoznačně určený identifikátor a pozici ve světě. Hierarchie je realizována abstraktní třídou `BlockOctreeNode`, od níž dědí třídy `BlockOctreeInternalNode` reprezentující vnitřní uzel oktalového stromu a `BlockOctreeLeafNode`. každý vnitřní uzel obsahuje list svých osmi potomků. Když je procházeno stromem, je index odvozen z pozice potomka. Tyto pozice jsou reprezentovány třemi celočíselnými souřadnicemi. Index potomka je definován bity v souřadnicích na stejných pozicích jako je výška potomka. Obrázek 3.2 znázorňuje odvození potomka z bitů pozice. Detailní popsání tohoto způsobu indexování potomků v oktalových stromech a jejich procházení je možné nalézt v práci Laine a Karras [7] "Efficient sparse voxel octrees". Bloky voxelů mají rozměry

výška	9	8	7	6	5	4	3	2	1	0
pos.z	1	0	1	1	0	0	1	0	0	0
pos.y	0	0	0	1	1	1	0	0	0	0
pos.x	0	1	1	1	0	1	1	0	0	0
index	1	4	5	7	2	6	5			

.

Obrázek 3.2: Odvození indexu potomka z pozice. Zdroj: [7]

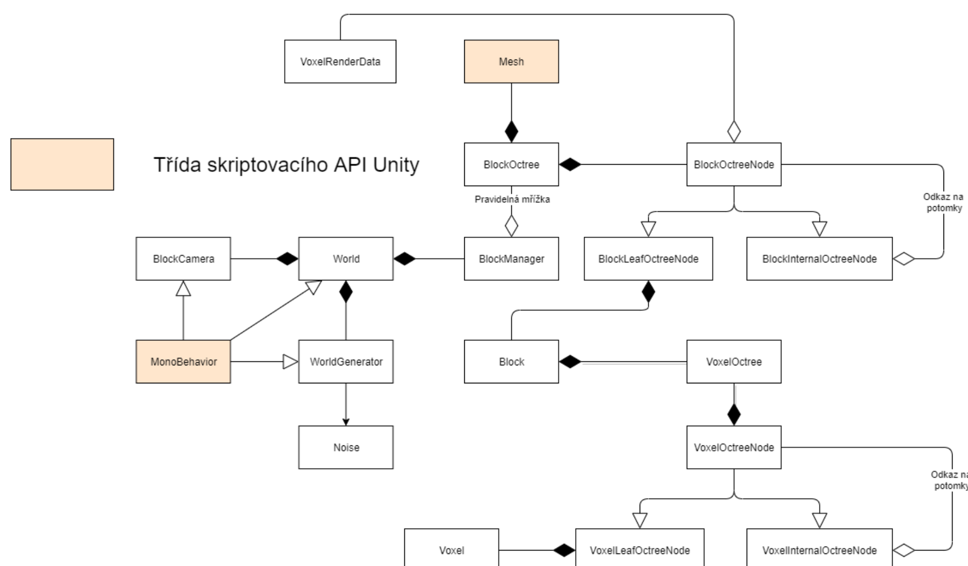
$16 \times 16 \times 16$ voxelů. Dosahují tedy výšky 4. Oktalový strom bloků voxelů má výšku také 4. Tím pádem jsou pro určení indexů potomků při procházení skrze uzly oktalového stromu bloků voxelů podstatné bity souřadnic na pozicích 4 až 7.

Blok voxelů

Třída `Block` poskytuje rozhraní pro generování voxelů regionu, jež obklopuje. Skrze něj jsou ze třídy `BlockManager` vyvolávány úlohy vygenerování voxelů, nebo shromáždění dat k vykreslení bloku. V současné implementaci ohraničuje region $16 \times 16 \times 16$ voxelů. Dále obsahuje instanci třídy `VoxelOctree`, které podobně jako `BlockOctree` obaluje oktalový strom, tentokrát však přímo voxelů, a implementuje rozhraní k jeho manipulaci.

Oktalový strom voxelů

Třída `VoxelOctree` reprezentuje hierarchii voxelů představenou v sekci **??**. Implementuje především generační procesy. Samotná hierarchie je tvořena třídami `VoxelOctreeInternalNode` pro vnitřní uzlu oktalového stromu voxelů a `VoxelOctreeLeafNode`. Toto členění bylo realizováno zejména kvůli paměťové náročnosti seznamu potomků, jehož přítomnost by byla v listech zbytečná. Indexy potomků jsou odvozovány stejně jako v případ2 oktalových stromů bloků voxelů s tím rozdílem, že tentokrát jsou relevantní bity souřadnic na pozicích 0 až 3. Namísto seznamu potomků obsahuje každý list odkaz na instanci třídy odvozené od třídy `Voxel`. Tyto objekty definují vlastnosti prostoru vymezeného listem v oktalovém stromu voxelů a poskytují rozhraní k vygenerování dat k vykreslení.



Obrázek 3.3: Objektový diagram reprezentace světa doplněný o jeho generování

3.3 Generování světa

Na obrázku 3.3 je vyobrazený objektový diagram reprezentace světa doplněný o jeho generování. Proces generování začíná ve skriptu `BlockCamera`, v níž je procházeno scénou a vyhodnocováno, které uzly oktalových stromů bloků voxelů se mají vykreslit. Pokud vznikne požadavek kamery na vykreslení uzlu (třída `BlockOctreeNode`), pro který doposud nebyla vygenerovaná data k vykreslení, či dokonce ani samotné voxely, uzel vyšle požadavek manažerovi pro jeho vygenerování. Z podmínek slučování uzlů v oktalových stromech bloků voxelů v sekci 2.14 je možné odvodit, že nevygenerované uzly mohou být pouze instance třídy `BlockOctreeLeafNode` obsahující konkrétní blok, pro něž se mají vygenerovat voxely, či data k vykreslení. Aby bylo možné identifikovat v jaké fázi generování daný blok je, obsahuje blok proměnnou `State` které může nabývat pěti různých stavů:

- **Empty** - blok byl pouze vykvořen. Voxely ani data k vykreslení zatím nebyla vygenerována.
- **GeneratingVoxels** - blok je v procesu generování voxelů. Tento proces zatím neskončil.
- **Generated** - voxely bloku jsou vygenerovány
- **GathreingRenderData** - blok je v procesu shromažďování dat k vykreslení. tento proces zatím nebyl dokončen.

- **RenderReady** - data k vykreslení byla úspěšně shromážděna a blok je připraven být vykreslen

Generování voxelů

Úlohy vygenerování voxelů jsou rozvrhovány manažerem. Svět je generován po blocích reprezentovaných třídou `Block`. Pokud vznikne požedavek na vygenerování voxelů určitého bloku, je nejprve zkontrolováno, zda-li je blok skutečně prázdný a tedy je možné vygenerovat jeho obsah. Samotné generování je implementováno ve třídě `VoxelOctree` v privátní funkci `GenerateVoxelOctree()`. Je uskutečněno ve stylu post-order procházení oktalového stromu. Nejdříve je pro uzel v oktalovém stromu voxelů vygenerováno všech osm potomků a na základě jejich hodnot je rozhodnuto, jestli je možné uzel sloučit do jednoho, či nikoli. Samotné určení voxelu na daných souřadnicích má na starosti třída `World`, která je přístupná skrze manažer. Tato třída obsahuje odkaz na generátor světa v podobě instance třídy `WorldGenerator`, jež vygeneruje voxel pro dané souřadnice. Obě tyto třídy jsou odvozené od `MonoBehavior`. Mohou tedy být použity v Unity editoru jako komponenty objektů ve scéně a také umožňují implementaci pravidelných aktualizací světa.

Generování probíhá paralelně. To přináší určité omezení při generování. Konkrétně:

- Stav generátoru se nesmí měnit
- Nadměrná alokace ve vláknech značně zpomaluje proces generování. Z toho důvodu je při generování předávána instance na první vygenerovaný list podstromu. Pokud je možné vygenerovaný voxel sloučit s voxelem v listu, je vygenerovaný voxel nahrazen voxelem listu. Pokud ani jeden potomek nebyl vygenerován není vygenerován ani vnitřní uzel představující kořen podstromu. Dále je vhodné bezstavové voxely, například voxely materiálů použité v demonstrační aplikaci v sekci 4.1, implementovat jako singletony. Jinak by tento mechanismus ztrácel na účinnosti. Voxely ve frameworku totiž definují vlastnosti prostoru. Samotná pozice v prostoru s těmito vlastnostmi je definovaná až v listu oktalového stromu voxelů.

Pro generování je nabídnuta třída `Noise` poskytující funkce základních šumů, jako jsou šum hodnot a voroného diagramy, v jejich variantách prostory dimense dva a tři. Ve třídě `World` je také omezen prostor, ve kterém je povoleno

generovat. Ten je ohraničen kvádrem o rozměrech $200000 \times 256 \times 200000$. Výška tohoto kvádru byla určena podle výšky světa v Minecraftu. Tento kvádr má střed na souřadnicích $\overrightarrow{C} = [0, 128, 0]$. Mimo objem tohoto kvádru není možné generovat bloky.

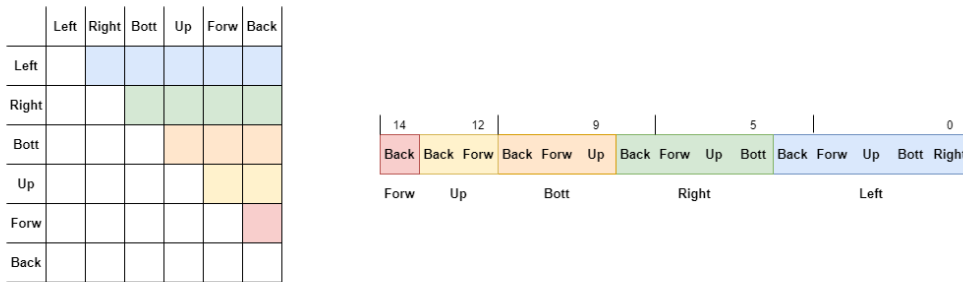
Po vygenerování je stav bloku nastaven na **Generated**.

Shromážděni dat k vykreslení

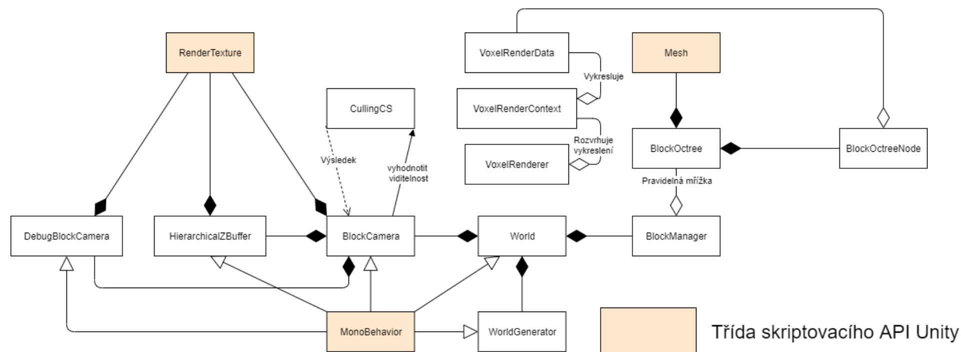
Data k vykreslení jsou reprezentována abstraktní třídou `VoxelRenderdata`. Shromažďování dat k vykreslení je opět inicializováno z třídy `BlockManager`. Aby mohla být data pro daný blok vygenerována, je potřeba, aby on i všechny sousední bloky byly ve stavu **Generated**. Shromažďování opět probíhá ve třídě `VoxelOctree`. Oktalovým stromem voxelů je tentokrát procházeno do šířky skrze listy obsahující konkrétní voxely. Procházeno je od stran bloků směrem do bloku. Pokud je při průchodu naraženo na voxel obsahující data k vykreslení, je voxel vyzván, aby svá data k vykreslení doplnil do příslušné instance třídy `VoxelRenderData`. Při tomto požadavku je také voxelu poskytnut uzel listu, jež ho obsahuje, aby bylo možné odvodit pozici voxelu ve scéně. Jednotlivé instance třídy `VoxelRenderData` jsou ukládány do hashovací tabylky, kde klíčem je identifikátor kontextu, jež je má vykreslovat. Kontexty, ve kterých jsou data vykreslovány jsou představeny v sekci 3.4. Některé voxely mohou vyžadovat informace o průhlednosti stran regionů, jež s nimi sousedí. Tato informace je při generování propagována z listů oktalového stromu voxelů až do jeho kořene. Konkrétní implementaci této propagace je možné nalézt v implementaci třídy `BlockOctreeInternalNode`.

Během procházení je pro každou stranu zaznamenáváno, jakých stran z ní bylo dosaženo. Tím je zformován graf viditelnosti. Formát grafu viditelnosti je nastíněn na obrázku 3.4. Údaje o propojenosti jednotlivých stran jsou zaznamenávány do bitové masky o celkové velikosti patnácti bitů. Po vygenerování je již v hlavním vláknu aplikace zahájena kontrola na slučování dat k vykreslení.

Slučování probíhá ve třídě `BlockOctree` ve funkci `SetRenderDataForNode()`. Před slučování se nejprve nastaví listu oktalového stromu bloků voxelů, jež obsahuje právě vygenerovaný blok, vygenerovaná hodnota grafu viditelnosti. Následně je rodič dotázán, zda-li je možné jeho potomky sloučit v jeden. Pokud ano, jsou data z potomků sloučena v jedny, potomci jsou odstraněny ze seznamu uzlů, jež jsou vykreslovány (seznamy popsány v sekci 3.4), a proces se opakuje. Jakmile je slučování dokončeno, je sloužený uzel zaregistrován



Obrázek 3.4: Formát grafu viditelnosti



Obrázek 3.5: Objektový diagram vykreslovacího enginu frameworku

jako uzel, jímž má být případně procházeno během procesu vykreslování. Dále je upravena vizualizace uzlů k vykreslení a nedosažitelných stran v grafu viditelnosti. Obě realizované `Mesh` objekty.

3.4 Vykreslování

Protože vykreslovací frameworky nabízené enginem Unity zahrnují funkcionalitu, jež by zkreslovala výsledky měření, je celý proces vykreslování umístěn do funkce `PostRender()` jež je volaná po tom, co je pomocí dané Unity kamery vykreslen obsah scény. V tomto případě není ve scéně nic k vykreslení. Nicméně kdyby bylo vykreslování zahájeno předem, vykreslený obsah by mohl být smazán. K vykreslování musí být použito takzvané "okamžité vykreslování"v podobě volání funkcí k vykreslování Unity grafického rozhraní `Graphics` s příponou `Now`, například `Graphics.DrawMeshNow()`.

Vykreslování ve frameworku má tři části:

Obrázek 3.6: Formát dat s informacemi o uzlu

1. Vyhodnocování viditelnosti uzlů oktalových stromů bloků voxelů
2. Průchod scénou skrze viditelné uzly
3. Vykreslení viditelných dat

3.4.1 Vyhodnocování viditelnosti

Vyhodnocování viditelnosti je prováděno na uzlech oktalového stromu bloků voxelů, jež obsahují data k vykreslení. Během tohoto procesu je nejdříve procházeno scénou na úrovni oktalových stromů bloků voxelů. Z těch jsou shromážděny informace o uzlech, jež obsahují data k vykreslení. Každá instance třídy `BlockOctree` si drží buffer s těmito informacemi ve formátu nastíněným na obrázku 3.6. X/Y/Zcoords představují pozici uzlu v oktalovém stromu bloků voxelů. Jedná se o čtyři bity každé souřadnice na pozicích 4 až 7. Následuje výška uzlu vzhledem k nejmenší možné, což je v tomto případě výška 4. Poslední údaj je index v seznamu pozic oktalových stromů bloků voxelů ukazující na pozici oktalového stromu bloků voxelů v němž se uzel vyskytuje. Každý uzel si pamatuje svou pozici v tomto bufferu. Během shromažďování těchto dat je dále každému oktalovému stromu bloků voxelů přidělen offset. Pomocí tohoto offsetu a indexu, který je uchován v jednotlivých uzlech je možné vyhledat data konkrétního uzlu ve shromážděných datech. Pokud se má při vyhodnocování viditelnosti použít hierarchický ZBuffer, je po shromáždění dat transformován funkcí `HierarchicalZBuffer.Transform()` a je vygenerována mip-pyramida pomocí funkce `HierarchicalZBuffer.GenerateHiZMipsFrom()`. Ta přijímá dvě textury hloubky, kde druhá může být například hloubková textura s vykreslenou informací zastiňujících stran v grafu viditelnosti. Třída `HierarchicalZBuffer` je zodpovědná za údržbu hierarchického ZBufferu. Například změna velikosti textur při změně rozlišení obrazovky. Implementace procesu generování mip-pyramidy hierarchického ZBufferu byla převzata z projektu UNNÞÓRSSON Elvar Örna [14, skript "Assets/Scripts/HiZbuffer.cs"a shader "Assets/shaders/HiZBuffer"].

Po shromáždění dat a případném vygenerování mip-pyramidy hierarchického ZBufferu je vyvolán ComputeShader s názvem `CullingCS`, v němž probíhá vyhodnocování viditelnosti. Jako vstup přijímá:

Obrázek 3.7: Promítnutí krychle na pohledový jehlan a ohraničující box

Obrázek 3.8: Formát výstupu `CullingCS` pro jeden uzel

- Shromážděná data o uzlech
- List pozic všech Oktalových stromů bloků voxelů
- Texturu hierarchického ZBufferu
- Pozici kamery a projekční a pohledovou matici kamery

Vyhodnocování probíhá následovně. Z dat uzlů a listů pozic BO se zrekonstruuje všech osm vrcholů krychle. Tyto vrcholy jsou následně otestovány na přítomnost v pohledovém jehlanu. Pokud se alespoň jeden vyskytuje uvnitř, jsou vyčnívající vrcholy promítnuty na pohledový jehlan a je vytvořen box ohraničující všech osm již promítnutých vrcholů, jako na obrázku 3.7.

Pokud je vyžadován i test skrze hierarchický ZBuffer, je vypočítána úroveň v mip-pyramidě, kdy nejdelší strana boxu ze stran rovnoběžných s osami x a y v souřadnicovém systému obrazu nepřesahuje velikost jednoho pixelu textury hierarcického ZBufferu. Potom je z textury hierarchického Zbufferu odebrán jeden nebo čtyři vzorky, záleží kolik pixelů box překrývá, a maximální hodnota hloubky ze vzorků je porovnána s maximální hloubkou boxu. Pokud je maximální hloubka vzorku větší než maximální hloubka boxu, je uzel vyhodnocen jako viditelný.

Nakonec je vyhodnoceno z pozice kamery a uzlu, jaké jeho strany vedou ke vzdálenějším uzlům. Konečný formát výstupu z `CullingCS` pro jeden uzel je zobrazený na obrázku 3.8.

3.4.2 Procházení scénou

Jakmile je rozhodnuto o viditelnosti uzlů může být scénou procházeno, aby mohla být data ve viditelných uzlech rozvržena k vykreslení. Každý uzel v sobě uchovává bitovou masku stran, kterými je možné expandovat dál do scény. Tato maska je při prvním průchodu uzlem inicializována na hodnotu výstupní masky z `CullingCS`. Z masky jsou během průchodu postupně odstraňovány bity těch stran, jimiž již v daném snímku bylo expandováno do sousedních bloků. Pokud není použit graf viditelnosti, je hned při prvním navštívení uzlu expandováno ze všech stran v masce. Pokud je uzel současně listem, to znamená, že obsahuje blok voxelů, záleží expanze také na stavu bloku. Pokud blok není ve stavu **RenderReady** není z uzlu expandováno a namísto toho je požádáno o vygenerování potřebných dat. Tím je zabráněno nadměrnému tvoření bloků za hranicí těch, které ještě ani nejsou vygenerovány. Uzly, které obsahují data k vykreslení, vyšlou požadavek k jejich vykreslení voláním funkce `VoxelRenderData.QueueForRendering()`.

3.4.3 Vykreslování

Vykreslování je realizováno jednoduchým frameworkem tvořeným třemi třídami.

První již byla představená. Třída `VoxelRenderData` obaluje jakákoli data k vykreslení. Každá odvozená třída musí implementovat rozhraní pro slučování a rozdělování dat a také k naplánování dat k jejich vykreslení. Při požadavku na vykreslení dat jsou data zaregistrována k vykreslení příslušným kontextem.

Druhá třída je třída `VoxelRenderContext`. Každá instance této třídy představuje kontext v jakém se vykreslují neplánovaná data. Například data v podobě polygonové mřížky mohou být jednou vykreslována v takovém kontextu, aby byla vykreslena jako voda, a jindy zase, v kontextu neprůhledné skály. Každá instance má identifikátor určující klíč ve slovníku dat k vykreslování používaném při jejich shromažďování. Dále musí implementovat rozhraní k vykreslení dat naplánovaných k vykreslení. V demonstrační aplikaci 4.1 jsou instance této třídy implementovány jako singletony.

Poslední třídou je `VoxelRenderer`. Tato třída poskytuje rozhraní pro vykreslení všech naplánovaných dat. Jednotlivé kontexty se podobně jako data registrují k vykreslení. Po zavolání funkce `VoxelRenderer.RenderScene()` jsou všechny registrované kontexty požádány o vykreslení naplánovaných dat

do dodaných textur finální barvy a hloubky. Pořadí vykreslování jednotlivých kontextů je určeno identifikátorem kontextu. Kontexty s nízkou hodnotou identifikátoru jsou vykreslovány dříve než ty s vysokou. Tím je možné zařídit, aby byla data průhledných voxelů vykreslována až po datech neprůhledných.

Kapitola 4

Výsledky

V této kapitole je představena demonstrační aplikace využívající implementovaný framework pro vygenerování a vykreslení pěti různých prostředí. Pro každé prostředí je dále změřeno, jak způsob členění scény a metody vyhodnocování zastínění bloků ovlivňuje výkon vykreslování.

4.1 Demonstrační aplikace

Demonstrační aplikace slouží jako prezentace možností frameworku. Pro aplikaci je implementováno pět generátorů scén. Jednotlivé scény jsou detailněji popsány v sekci 4.1.2. Dále jsou definovány tři typy voxelů, které se mohou ve scénách vyskytovat. Přesný popis voxelů lze nalézt v sekci 4.1.1

4.1.1 Voxely

Pro aplikaci jsou definovány tři typy voxelů:

- Prázdný voxel
- Barevný voxel

- Barevný voxel vody

Všechny zde prezentované typy voxelů jsou bez vnitřního stavu, nebo je jejich stav neměnný. Mohou tedy být implementovány jako sigletony. Tím se značně sníží alokace ve vláknech, která zvyšuje čas generování. Všechny typy voxelů jsou uloženy ve složce `Assets/Scripts/Voxels/` v Unity projektu aplikace. Data k vykreslení a kontexty ve složce `Assets/Scripts/Rendering/`.

Prázdné voxely

Prázdné voxely reprezentují vzduch. Jsou kompletně průhledné a nejsou vykreslovány. Prázdné voxely je možné slučovat pouze s prázdnými voxely. Implementaci je možné najít v souboru `Voxel_Empty.cs`.

Barevné voxely

Barevné voxely slouží pro reprezentaci neprůhledných materiálů. Jsou vykreslovány jako šest čtverců tvořící jednotlivé strany krychle. Některé strany mohou být vynechány podle toho, zda-li jsou zastíněné neprůhlednými voxely. Pro tento voxel je definována paleta barev pro materiály používané v aplikaci. Konkrétně:

- Bílá (sníh, mraky)
- Hlína
- Písek
- Kámen
- Dřevo
- Kaktus
- Čepice hub
- Koruna jehličnatých stromů

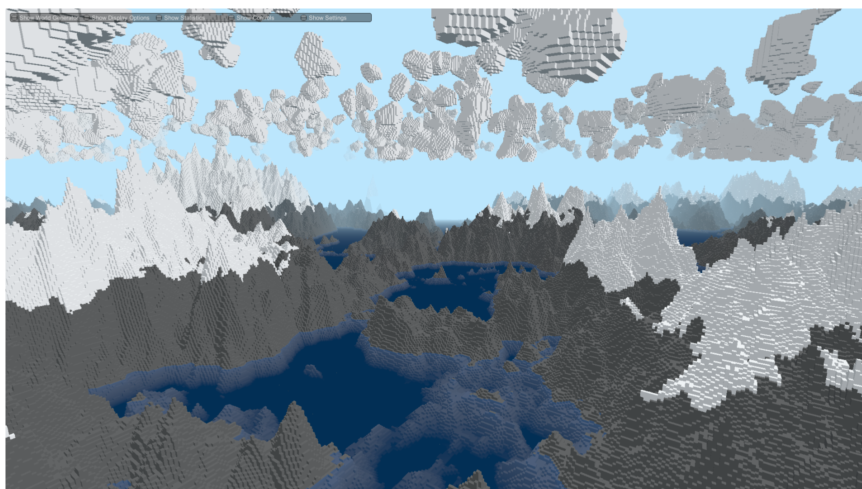
Mohou se slučovat pouze s barevnými voxely se stejnou barvou. Konkrétní implementace tohoto typu voxelu je možné najít v souboru `Voxel_Color.cs`. Dále je implementována reprezentace dat k vykreslení využívající Mesh objekt k ukládání geometrie barevných voxelů a kontext, ve kterém jsou tato data vykreslována. Implementace těchto tříd je možné najít v souborech `ColoVoxelRenderData.cs` a `ColoVoxelRenderContext.cs`. Data je možné sloučit, pokud celkový počet slučovaných vrcholů nepřesáhne hranici 60000. Identifikátor tohoto typu voxelu je nastaven na hodnotu 1 a identifikátor kontextu, ve kterém je vykreslován na hodnotu 0.

Voxely barevné vody

Tento typ voxelů má představovat kapaliny podobné vodě. Je vykreslován shaderem, který na základě informace ze současného ZBufferu určí, jak moc bude daný pixel obarven barvou tohoto voxelu. To je dobře vidět na ukázce scény Oceánu (Obrázek 4.2) nebo Hor ve vodě (Obrázek 4.1). Z tohoto důvodu je nutné tento typ klasifikovat jako průhledný, což má za následek nezaznamenání informace o zastínění vzdálených objektů, i když již nejsou vidět, ať už v podobě ZBufferu nebo grafu viditelnosti. Ukázkou může bít vykreslení hierarchického ZBufferu scény oceánu na obrázku 4.3. Opět je definována paleta barev:

- Oceán (tmavě modrá barva)
- Laguna (tyrkysová), použita ve scéně Jeskyně 4.1.2

Data k vykreslování jsou totožná s daty pro vykreslování neprůhledných Barevných voxelů. Nicméně jsou vykreslována v jiném kontextu a proto je pro ně vytvořena nová třída implementovaná v souboru `ColoWaterVoxelRenderData.cs` (podmínka pro sloučení těchto dat je stejná jako pro barevné neprůhledné voxely). Kontext, ve kterém je vykreslován je implementován v souboru `ColoWaterVoxelRenderContext.cs`. Je nutné, aby data těchto voxelů byla vykreslována až po vykreslení neprůhledných voxelů, a proto je identifikátor konextu nastaven na hodnotu 2 (větší než identifikátor kontextu neprůhledných voxelů). Identifikátor typu voxelu je nastaven na hodnotu 2. Voxely barevné vody se mohou slučovat pouze s voxely barevné vody stejné barvy a to pouze pokud by byl výsledný sloučený voxel obklopen voxely opět stejného typu a barvy. Implementace voxelu barevné vody si je možné prostudovat v souboru `ColorWater_Voxel.cs`.



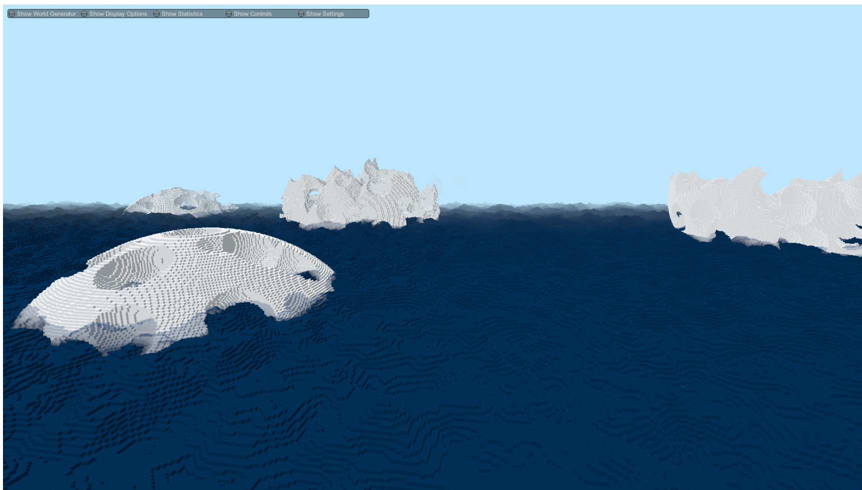
Obrázek 4.1: Scéna hor ve vodě

4.1.2 Generátory scén

Pro aplikaci je implementováno pět generátorů voxelových scén. Každá scéna klade důraz na jiný prvek nebo kombinaci prvků prostředí, které se vyskytují v běžném světě, například vysoké hory, vegetace, oceán apod. Každý takový prvek má jiné nároky na vykreslování. Hory zastiňují, vegetace představuje velké množství drobné geometrie a oceán obsahuje velké množství průhledných voxelů. Všechny generátory jsou uložené ve složce `Assets/Scripts/Generating/` v Unity projektu aplikace.

Hory ve vodě

Jak je vidět na obrázku 4.1, tato scéna obsahuje vysoké hory. Dolní část hor je ponořena ve vodě. Nad horami jsou poněkud drobné shluky bílých voxelů vypadajících jako mraky. Konkrétní kód generátoru lze nalézt v projektu v souboru `WorldGenerator_FloodedMountains.cs`. Pro tuto scénu je specifické velké množství zastiňujících objektů a v horní části drobná geometrie. Vykreslování by mělo značně benefitovat z aplikace slučování bloků a vyhodnocování zastínění.



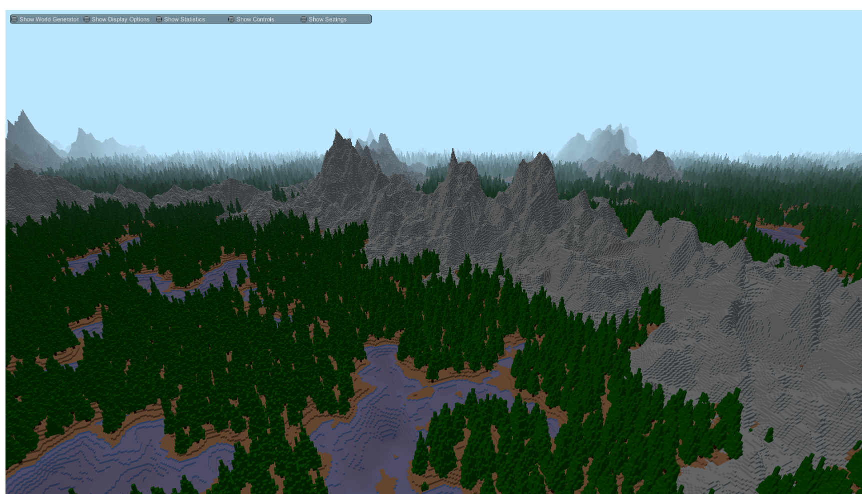
Obrázek 4.2: Scéna oceánu

Obrázek 4.3: Hierarchický ZBuffer pro scénu oceánu (mip-level 0)

Oceán

Oceán (obrázek 4.2) tvoří převážné voxely vody. Přestože jsou tyto bloky průhledné pouze do určité vzdálenosti, jsou klasifikovány jako průhledné a tedy aplikace vyhodnocování zastínění objektů by mělo mít na vykreslování spíše negativní dopad. Ve vodě jsou dále umístěny pórovité koule a dno, aby scéna nepůsobila příliš monotonně a pro vizualizaci informace v hierarchickém ZBufferu, jak je vidět na obrázku 4.3. Generátor této scény se nachází v souboru `WorldGenerator_Ocean.cs`.

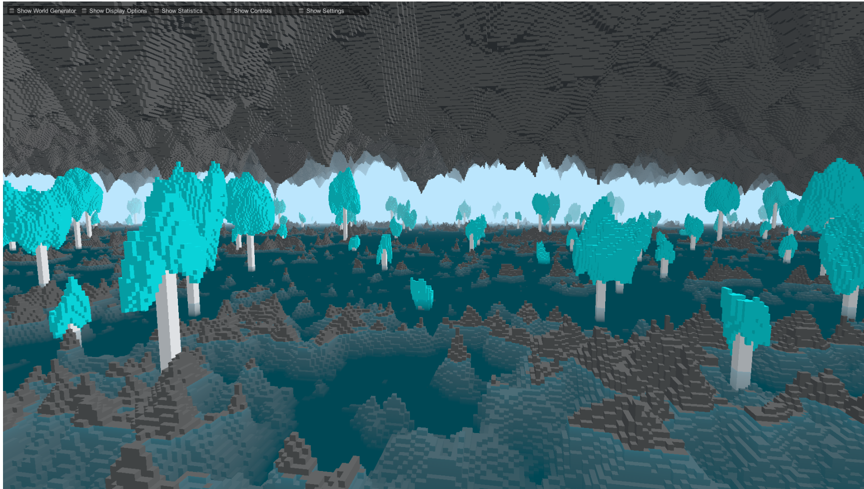
Obrázek 4.4: Scéna pouště



Obrázek 4.5: Scéna krajiny s jeskyněmi

Poušť

Zřejmě nejprimitivnější scéna (obrázek 4.4) tvořená mírnými vrcholy představující písečné duny a nízké sloupy zelených neprůhledných voxelů reprezentující kaktusy. Krajina je plochá a bez jeskyň či jiných defektů. Pod povrchem se vyskytuje pouze blok písku. Vykreslování by tedy mělo být výrazně efektivnější s využitím slučování bloků a vyhodnocováním zastínění objektů. Generátor lze nalézt v souboru `WorldGenerator_Desert.cs`.



Obrázek 4.6: Scéna jeskyně

Krajina s jeskyněmi

Ve scéně generované tímto generátorem (obrázek 4.5) lze nalézt velké množství vegetace v podobě jehličnatých stromů. Stromy jsou generované pouze na voxelech zeminy. Nad úroveň zeminy místy vystupuje voda. Dále se zde vyskytují hory z voxelů reprezentující kámen, které jsou místy nahrazeny prázdnými voxely pro vytvoření jeskyní. Opět se předpokládá, že vykreslování byde značně výkonnější s aplikací slučování bloků voxelů vzhledem k širokému výskytu drobné geometrie v podobě stromů. Vyšší výkon by měly přinést i metody vyhodnocování zastínění objektů. Kód generátoru lze nalézt v souboru `WorldGenerator_landWithCaves.cs`.

Jeskyně

Scéna jeskyně (obrázek 4.6) je shora i zdola ohraničena geometrií neprůhledných voxelů kamene. Horní část reprezentující strop jeskyně ze stalaktitů má mnohonásobně větší amplitudu než dolní ze stalagmitů. Spodní část je dále ponořena ve vodě a vyrůstají z ní vysoké houby. Slučování bloků by mělo mít menší dopad na výkon vykreslování vzhledem k velkému množství geometrie v podobě hlav hub a krápníků v dolní i horní části scény. Tak velké množství geometrie se nemusí sloučit, protože slučované data nemusí splnit podmínky pro sloučení. Dalším důvodem mohou být jiné hodnoty grafu viditelnosti slučovaných uzlů způsobené opět krápníky. Vyhodnocování zastínění by mohlo mírně zlepšit výkon vykreslování při pohybu mezi stalaktity v hodní části scény. Soubor s kódem generujícím tuto scénu lze nalézt v

WorldGenerator_cave.cs.

4.1.3 Uživatelské rozhraní

Pro ovládání aplikace jsou používány klávesové zkratky a jednoduché uživatelské rozhraní. Uživatelské rozhraní se skládá z šesti částí.

Zobrazení sekce

Obrázek 4.7: Sekce uživatelského rozhraní s možností zobrazení sekcí

Tento panel (Obrázek 4.7) umožňuje skrývat jednotlivé sekce uživatelského rozhraní kliknutím na konkrétní přepínací tlačítko pro danou sekci.

Výběr generátoru

Obrázek 4.8: Sekce uživatelského rozhraní s nabídkou generátorů scén

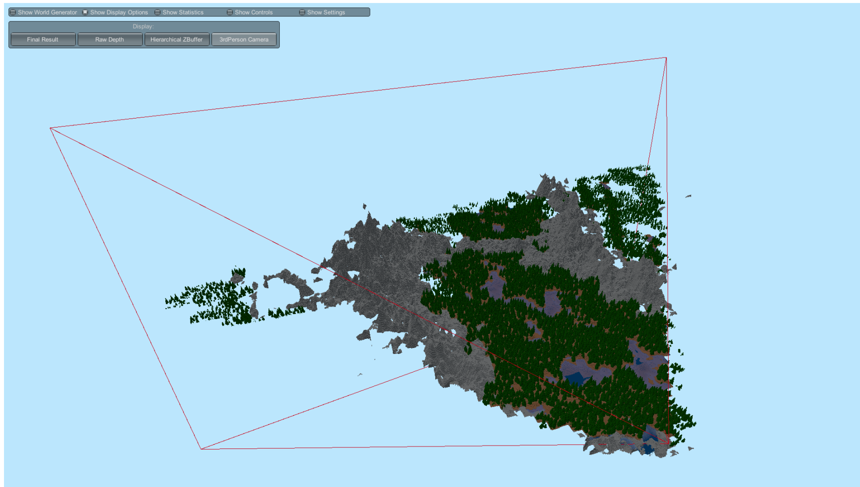
V této sekci je možné vybírat mezi dostupnými generátory scén. Při výběru jiného generátoru než je ten aktivní se smažou data současné scény, vyvolá se GarbageCollector.Collect a resetují se statistiky. Současně je k dispozici celkem pět generátorů pro scény popsané v sekci 4.1.2.

Zobrazení

Tato sekce (obrázek 4.9) dává uživateli možnost rozhodnout se, co bude vykresleno na obrazovku. K dispozici jsou čtyři možnosti:

- **Final Result** představuje konečný výsledek vykreslování. Takto by viděl scénu hráč.

Obrázek 4.9: Sekce uživatelského rozhraní s nabídkou zobrazení



Obrázek 4.10: Pohled kamerou třetí osoby

- **Raw Depth** na obrazovku vykreslí hloubkovou textury scény (ZBuffer). Tato textura je v následujícím snímku použita k vytvoření hierarchického ZBufferu.
- **Hierarchical ZBuffer** zobrazí hierarchický ZBuffer použitý k vyhodnocování zastínění jednotlivých bloků. Zvolením této možnosti se zobrazí i panel s posuvníkem umožňující zvolit úroveň mip-pyramidy, jež se má zobrazit.
- **3drPerson Camera** vykreslí scénu z pohledu druhé kamery. tuto možnost je možné využít pro debugování aplikace a vizualizaci vykreslených bloků jako na obrázku 4.10. Při startu aplikace nebo změně generátoru či startovací pozice kamery má tato kamera stejnou transformaci jako hlavní kamera.

Statistiky

Sekce statistik (Obrázek 4.11) zobrazuje údaje spojené s generováním a vykreslováním scény. Skládá se ze tří oddílů.

1. **WorldStats** - data vztahující se ke generování světa

WorldStats	
Scheduled Tasks:	0
Remaining main-thread tasks:	0
Generate voxels time:	~2,492ms
Generate voxels time (finish):	~0,000ms
Gather render data time:	~0,094ms
Gather render data time (finish):	~0,173ms
Create block time:	~0,006ms
Delete blockOctree time:	~ms
Created blocks:	4 427
VoxelLeafNodes num:	0,10M / 18,14M
VoxelInternalNodes num:	16 549

MemoryStats	
Total Used Memory:	149MB
GC Reserved Memory:	31MB
GC Used Memory:	30MB

RenderingStats	
Main Camera pos:	(259,6, 135,0, -64,5)
Debug Camera pos:	(250,0, 135,0, -51,0)
FPS:	~162,147
Frame time:	~6,167ms
Rendering:	~3,272ms
Culling:	~1,916ms
Traversal:	~1,070ms
Drawing:	~0,113ms
Debug Camera:	~0,000ms
Processed nodes:	~2 363
Traversed nodes:	~1 017
Renderer:	
Contexts:	1
Draw calls:	~13
Vertices:	~0,010M

Obrázek 4.11: Sekce uživatelského rozhraní se statistikami

- **Scheduled Tasks** - kolik je současně naplánováno paralelních úloh (generování voxelů bloků nebo shromáždění dat k vykreslení)
- **Remaining main-thread tasks** - současný počet naplánovaných úloh k vykonání hlavním vláknem aplikace
- **Generate voxels time** - čas strávený ve vedlejším vláknu generováním voxelů a jejich hierarchie pro jeden blok
- **Generate voxels time(finish)** - čas strávený v hlavním vláknu dokončováním po vygenerování voxelů v bloku
- **Gather render data time** - čas strávený ve vedlejším vláknu shromažďováním dat k vykreslení voxelů v bloku a konstrukcí grafu viditelnosti
- **Gather render data time(finish)** - čas strávený v hlavním vláknu dokončováním po shromáždění dat k vykreslování (především slučování dat)
- **Create block time** - čas vytváření jednoho prázdného bloku
- **Delete blockOctree time** - čas mazání jednoho oktalového stromu bloků voxelů
- **Created blocks** - počet aktuálně vytvořených bloků
- **VoxelLeafNodes num** - současný celkový počet listů v oktalových stromech voxelů ve všech vytvořených blocích / maximální počet (tzn. kolik by jich bylo, kdyby se pro ukládání voxelů nepoužíval oktalový strom, ale pouhý list)
- **VoxelInternalNodes num** - aktuální počet uzlů, jež nejsou listy, v oktalových stromech voxelů všech současně vytvořených bloků

2. **RenderingStats** - data vztahující se k vykreslování scény

    - **Main Camera pos** - pozice hlavní kamery

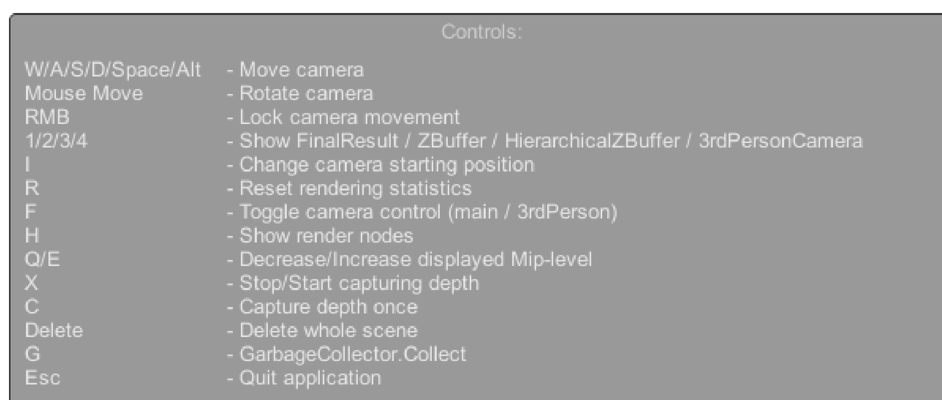
- **Debug Camera pos** - pozice kamery pro pohled třetí osoby
- **FPS** - počet snímků za sekundu
- **Frame time** - doba vytvoření jednoho snímku
- **Rendering** - čas strávený procesem vykreslování jednoho snímku
- **Culling** - čas strávený rozhodováním o viditelnosti
- **Traversal** - doba průchodu scénou
- **Drawing** - čas strávený samotným vykreslováním scény v hlavní kameře
- **Debug Camera** - čas strávený vykreslováním scény v kameře pohledu třetí osoby
- **Processed Nodes** - počet uzlů, jež byli během snímku zpracovány metodami pro rozhodování o jejich viditelnosti
- **Traversed Nodes** - počet uzlů, kterými bylo v jednom snímku procházeno při průchodu scénou po vyhodnocení viditelnosti
- **Contexts** - v kolika kontextech byla data vykreslována
- **DrawCalls** - kolik bylo během jednoho snímku vykresleno dat (v aplikaci tento ukazatel značí počet vykreslených Mesh objektů za jeden snímek)
- **Vertices** - kolik bylo v jednom snímku vykresleno vrcholů

3. **MemoryStats** - data o využití paměti počítače[1]

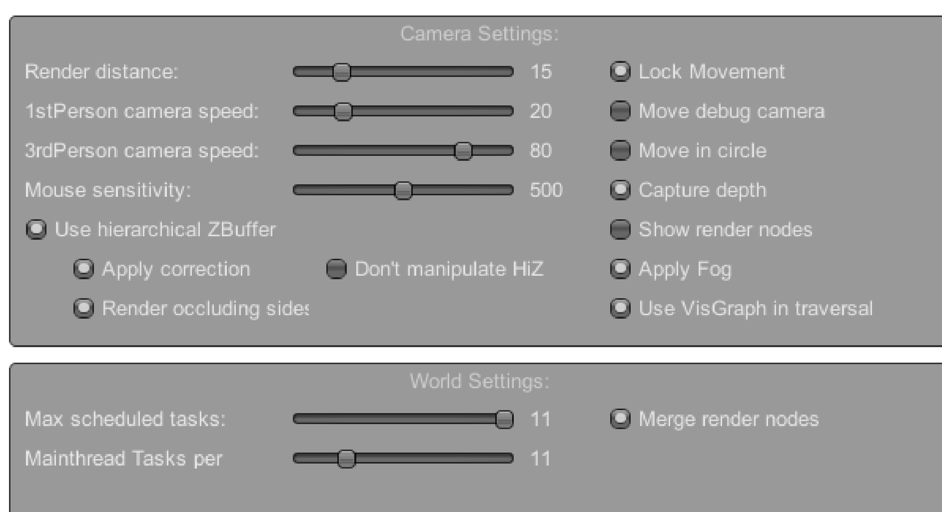
- **Total Used Memory** - celková využitá paměť pro chod aplikace
- **GC Reserved Memory** - paměť rezervovaná GarbageCollectorem Unity
- **GC Used Memory** - paměť současně využívaná Unity Garbage-Collectorem

Ovládání

Sekce s nápovědou k ovládání aplikace (Obrázek 4.12). Všechny znaky jsou na alfanumerické klávesnici. To se týká především čísel, které na NumPadu nebudou reagovat. Dále je použita myš k rotaci kamery a RMB(Right mouse button - pravé tlačítko myši) k pozastavení pohybu kamer.



Obrázek 4.12: Sekce uživatelského rozhraní s nápovědou k ovládání



Obrázek 4.13: Sekce uživatelského rozhraní s nastavením aplikace

Nastavení

V uživatelském rozhraní je možné nastavovat některé parametry aplikace. Sekce nastavení (Obrázek 4.13) je rozdělena do dvou částí:

1. **Camera Settings** - nastavení kamery a vykreslování
   - **Render distance** - vykreslovací vzdálenost v blocích. Tato vzdálenost je interně vynásobena velikostí jednoho bloku (momentálně 16 voxelů) a výsledná hodnota pak udává výšku pohledového jehlanu hlavní kamery

[1]Zobrazené hodnoty odpovídají stejnojmenným datům v Unity Profileru [20, sekce: Working in Unity/Analysis/Profiler overview/The Profiler window/Memory Profiler module]

- **1stPerson camera speed** - rychlost hlavní kamery v nejmenších jednotkách scény za sekundu
- **3rdPerson camera speed** - rychlost kamery pohledu třetí osoby v nejmenších jednotkách scény za sekundu
- **Mouse Sensitivity** - citlivost myši
- **Use hierarchical ZBuffer** - použít pro rozhodování viditelnosti hierarchický ZBuffer
  - **Don't manupulate HiZ** - pokud je tato možnost aktivní, je hierarchický ZBuffer použit při rozhodování o viditelnosti, ale během snímku s ním není nijak manupulováno
  - **Apply correction** - aplikovat korekci po transformaci ZBufferu z předchozího snímku
  - **Render occluding sides** - vykreslit kompletně nedosažitelné strany bloků (informace z grafu viditelnosti) do hierarchického ZBufferu
- **Lock Movement** - zda-li je povoleno hýbat s kamerami
- **Move debug camera** - zda-li pohybovat s hlavní kamerou nebo kamerou pohledu třetí osoby (velmi praktické při pohledu kamerou třetí osoby, kdy uživatel chce pohnout hlavní kamerou)
- **Move in circle** - pohybovat hlavní kamerou v kole (použito při měření)
- **Capture depth** - ukládat hloubkovou texturu z předchozího snímku
- **Show render nodes** - vykreslit hierarchii uzlů v oktalovém stromu bloků voxelů, vykreslené uzly jsou ty, jimiž je procházeno při vykreslování (jednotlivé oktalové stromy bloků voxelů jsou barevně odlišeny šachovnicovým vzorem)
- **Apply fog** - při vykreslování aplikovat mlhu
- **Use VisGraph in traversal** - během procházení scénou použít graf viditelnosti

2. **World Settings** - nastavení generování světa

   - **Max scheduled tasks** - kolik může být v jeden okamžik rozvrženo úloh pro generování voxelů nebo shromažďování dat pro vykreslování
   - **Mainthread tasks per frame** - maximální počet provedených úloh (dokončování generování/shromažďování) v hlavním vláknu
   - **Merge render nodes** - zda-li slučovat shromážděná data pro vykreslování jednotlivých bloků, pokud splní všechny nutné podmínky. (po změně je nutné vymazat současnou scénu, jelikož vygenerovaná data se již nesloučí ani nerozpojí)

4.2 Měření výkonu vykreslování

Na každé scéně bylo změřeno, jak se projevuje slučování dat k vykreslování v uzlech oktalových stromů bloků voxelů a různé kombinace metod pro vyhodnocování zastínění objektů na výkonu vykreslování scén. Měření probíhala následovně. Pro každou scénu jsou definovány dvě startovní pozice kamery, v dolní a horní části scény. Mezi těmito pozicemi je možné přepínat klávesou I. Protože je běžné, aby se hráč ve scéně pohyboval, byla kamera z každé startovní pozice rozpohybována do kruhu rychlostí dvaceti jednotek(nejmenších možných voxelů) za sekundu. Měření bylo uskutečněno pro osm kombinací metody slučování (povoleno/zakázáno) a metod vyhodnocování viditelnosti (pouze pohledový jehlan (F), pohledový jehlan a hierarchický ZBuffer (F+H), pohledový jehlan a graf viditelnosti (F+V), pohledový jehlan a graf viditelnosti a hierarchický ZBuffer (F+V+H)) pro každou pozici pro 4 různé vzdálenosti vykreslování (8, 16, 24 a 32). Celkem tedy šedesát čtyři měření na jednu scénu. Pro každou scénu jsou níže vykresleny grafy zobrazující naměřenou závislost snímkové frekvence na vykreslovací vzdálenosti pro tyto kombinace. Dále jsou vykresleny dva grafy pro největší vykreslovací vzdálenost (32) zobrazující průměrné časy jednotlivých fází vykreslování s a bez slučování dat k vykreslení.

Inicializace měření:

1. Zvolit generátor
2. Uzamknout pohyb kamer **Lock Movement**
3. Zvolit pozici klávesou I
4. Snížit vykreslovací vzdálenost na 8
5. Nastavit zobrazení na **Final Result**
6. Zvolit **Move in circle**
7. Povolit nebo zakázat slučování (**Merge render nodes**)
8. Stisknout klávesu Delete pro smazání současné scény
9. Odemknout pohyb kamery

Dále se nechala scéna generovat bez použití hierarchického ZBufferu a grafu viditelnosti dokud počet vytvořených bloků neměnil. Poté bylo pro každou metodu vyhodnocování viditelnosti započato měření stisknutím klávesy R,

která vynuluje statistiky vykreslování. Po alespoň jedné minutě, kdy se už průměrné naměřené hodnoty ustálily, byl vytvořen snímek okna statistik. Po uskutečnění měření pro danou vykreslovací vzdálenost byla vzdálenost zvýšena a proces se opakoval. Snímky statistik všech měření je možné nalézt v příloze **??**.

Scéna byla měřena na stolním počítači s následujícími komponentami:

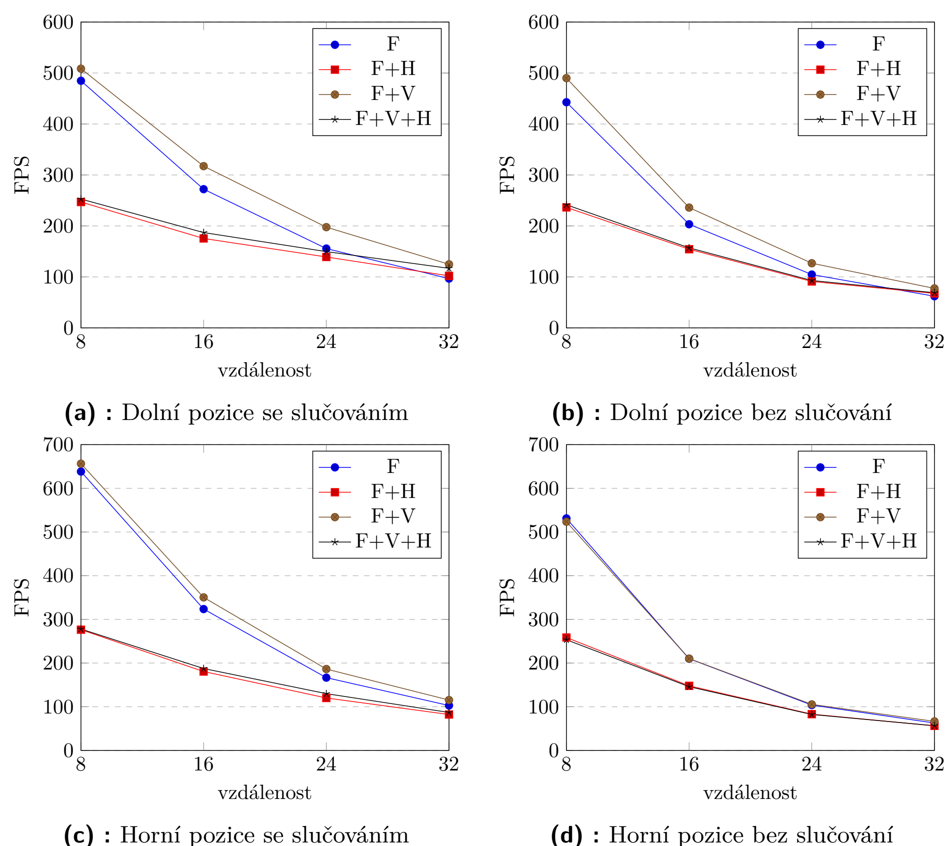
- AMD Ryzen 5 1600 Six-Core Processor 3.20 GHz
- MSI GeForce GTX 1050 Ti 4GT OC, 4GB GDDR5
- HyperX Fury Black 8GB (2x4GB) DDR4 2666 CL16

4.2.1 Hory ve vodě

Scéna hor, jak je popsáno v sekci 4.1.2, obsahuje velké množství neprůhledných voxelů. V grafech v obrázku 4.14 je možné zpozorovat, jak se s přibývající vykreslovací vzdáleností vyplácí aplikovat metody vyhodnocování zastínění objektů. To platí především při měření ve spodní části scény. Povolení slučování dat k vykreslování a prázdných bloků má taktéž velmi pozitivní vliv na snímkovou frekvenci, jelikož se ve scéně vyskytuje drobná geometrie a v podobě mraků a prázdné bloky, jež mohou být přeskočeny všechny najednou. Grafy v obrázku 4.15 zase ukazují, že nejkritičtějším místem v procesu vykreslování je procházení scénou, jehož čas se při zákazu slučování více než zdvojnásobí. Je také možné si povšimnout, že aplikace hierarchického ZBufferu ve sníží počet procházených uzlů ve scéně, ale časové nároky na vyhodnocení viditelnosti jsou mnohonásobně vyšší než časová výhoda, kterou jeho aplikací vzniká. Tomuto problému se věnuje sekce 4.2.6.

4.2.2 Oceán

Narozdíl od scény hor scéně oceánu (více v sekci 4.1.2) dominují průhledné voxely vody. To se značně projeví na výkonu vykreslování. Jak je vidět v grafech na obrázku 4.16, aplikace vyhodnocování zastínění (V a H) se buď vůbec neprojeví na snímkové frekvenci nebo ji ovlivní až negativně. Velmi však pro výkon vykreslování pomáhá slučování, jelikož je možné sloučit velké oblasti prázdných bloků a bloků pod vodou v jedny. Snímková frekvence se při povolení slučování téměř zdvojnásobila. Z obrázku 4.17 je možné zpozorovat zlepšení



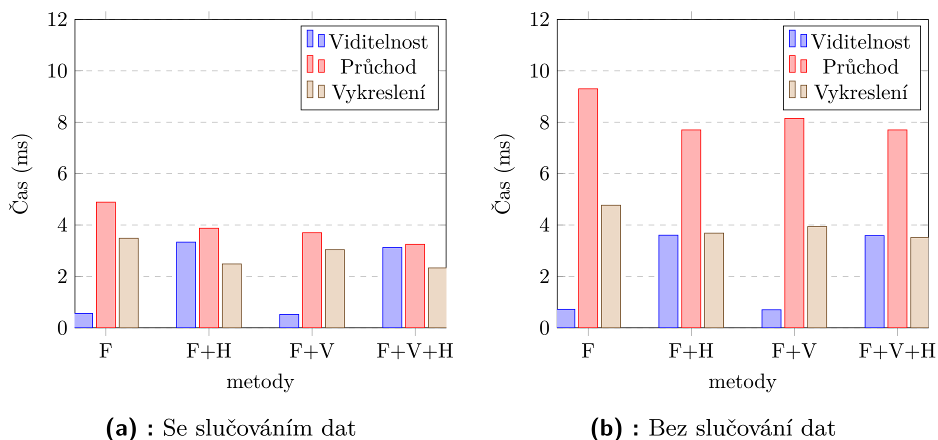
(a) : Dolní pozice se slučováním

(b) : Dolní pozice bez slučování

(c) : Horní pozice se slučováním

(d) : Horní pozice bez slučování

Obrázek 4.14: Vliv slučování dat k vykreslení a kombinací metod rozhodování o viditelnosti na snímkovou frekvenci ve scéně Hory

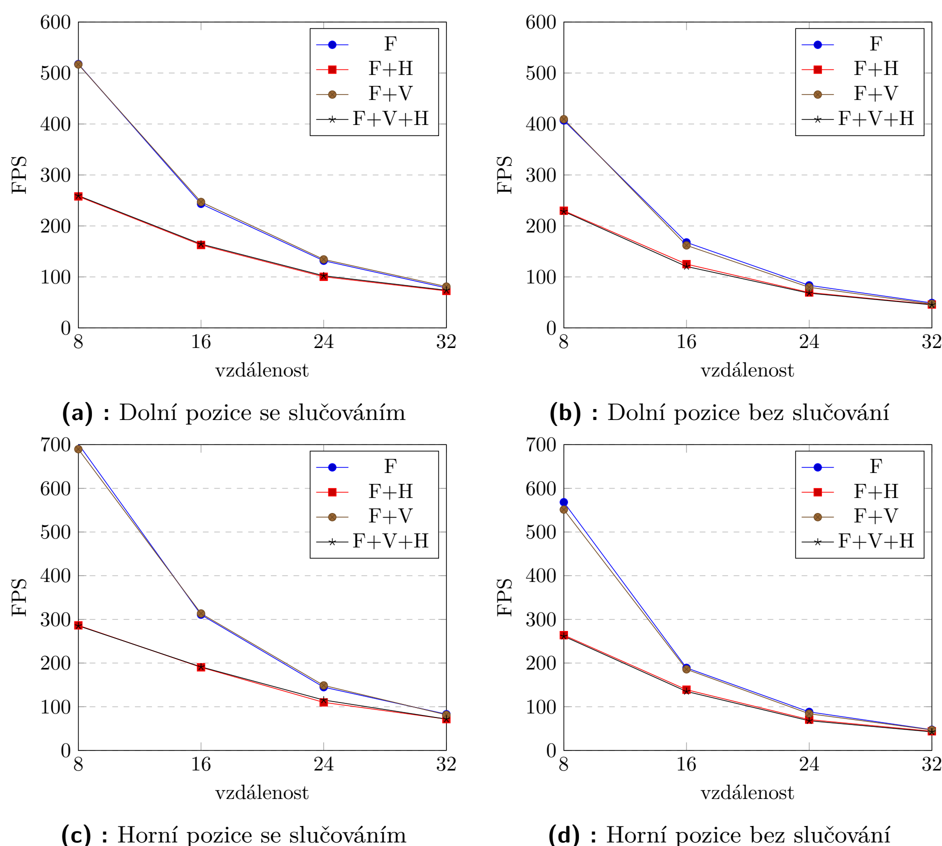


(a) : Se slučováním dat

(b) : Bez slučování dat

Obrázek 4.15: Průměrné časy fází vykreslování ve scéně Hor pro vzdálenost 32

času procházení a vykreslování viditelných dat při aplikaci hierarchického ZBufferu. Zlepšení však opět není tak významné, jako zvýšení nároků na vyhodnocení viditelnosti.



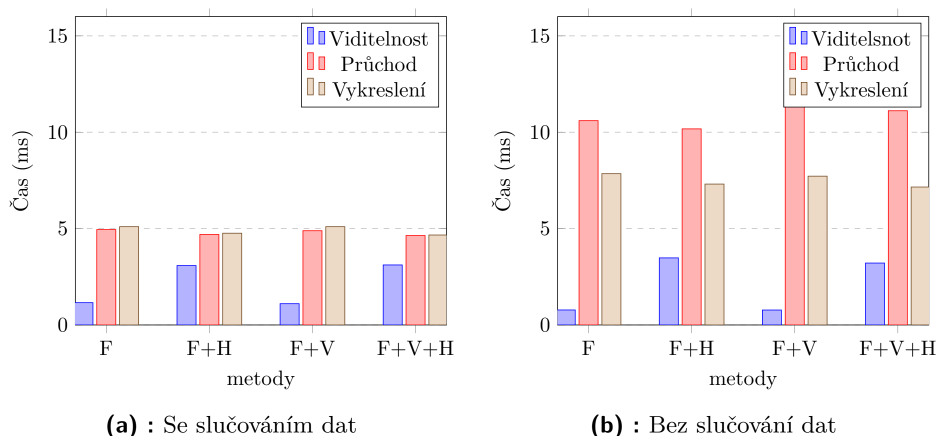
(a) : Dolní pozice se slučováním

(b) : Dolní pozice bez slučování

(c) : Horní pozice se slučováním

(d) : Horní pozice bez slučování

Obrázek 4.16: Vliv slučování dat k vykreslení a kombinací metod rozhodování o viditelnosti na snímkovou frekvenci ve scéně Ocean



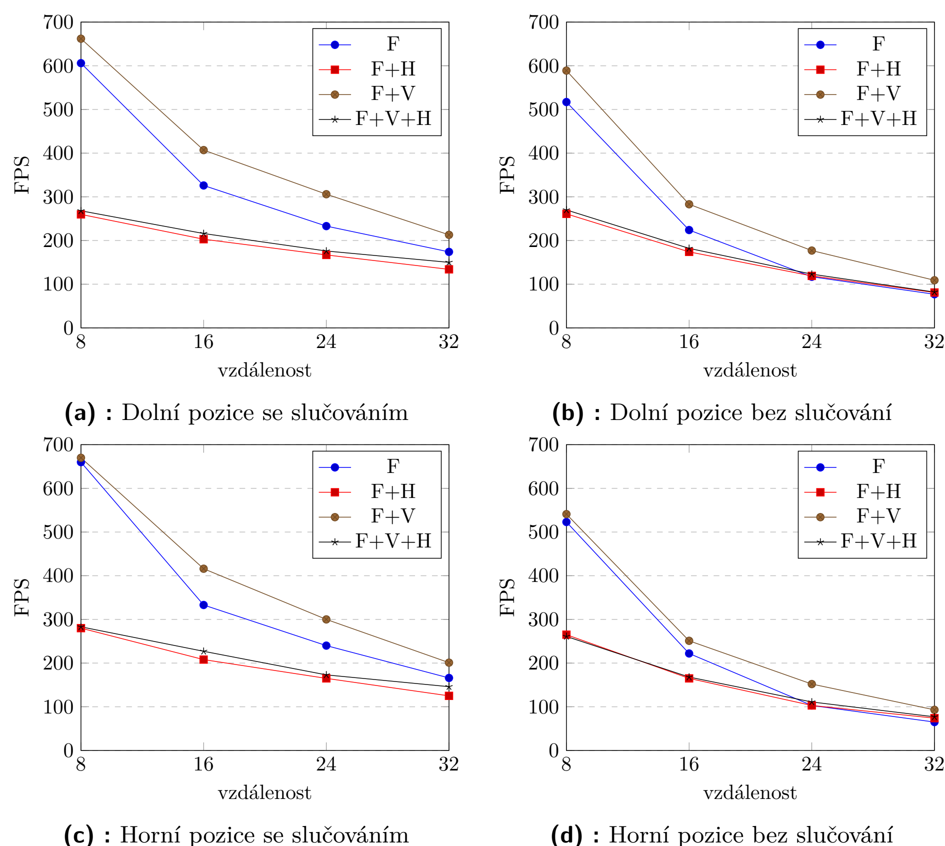
(a) : Se slučováním dat

(b) : Bez slučování dat

Obrázek 4.17: Průměrné časy fází vykreslování ve scéně Oceán pro vzdálenost 32

4.2.3 Poušť

Scéna pouště obsahuje pouze neprůhledné voxely s barvou písku a kaktusu. Pokud jsou pominuty kaktusy, je definovaná pouze výškovou mapou s nízkou amplitudou (více informací v sekci 4.1.2). Naměřená data zobrazená v grafech na obrázku 4.18 ukazují, že aplikace slučování opět přinesla až dvojnásobné zrychlení. Z metod vyhodnocování viditelnosti byla v tomto případě optimální aplikace pouze grafu viditelnosti, jelikož hierarchický ZBuffer pouze kopíroval informaci v grafu viditelnosti (plochá krajina) a nároky na vyhodnocení viditelnosti jsou opět příliš vysoké, jak je vidět na obrázku 4.19.



(a) : Dolní pozice se slučováním

(b) : Dolní pozice bez slučování

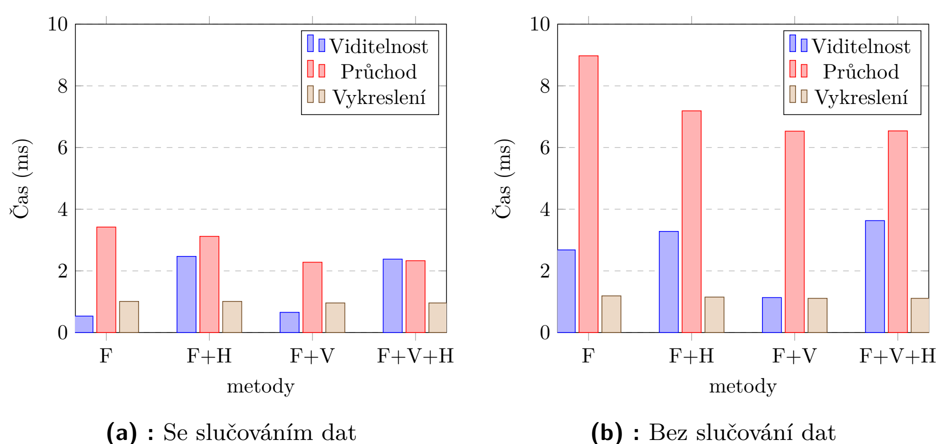
(c) : Horní pozice se slučováním

(d) : Horní pozice bez slučování

Obrázek 4.18: Vliv slučování dat k vykreslení a kombinací metod rozhodování o viditelnosti na snímkovou frekvenci ve scéně Poušť

4.2.4 Krajina s jeskyněmi

Krajina s jeskyněmi obsahuje velké množství drobné geometrie v podobě jehličnatých stromů a vzduchových kapes ve voxelech barvy kamene. Jak



(a) : Se slučováním dat

(b) : Bez slučování dat

Obrázek 4.19: Průměrné časy fází vykreslování scény Pouště pro vzdálenost 32

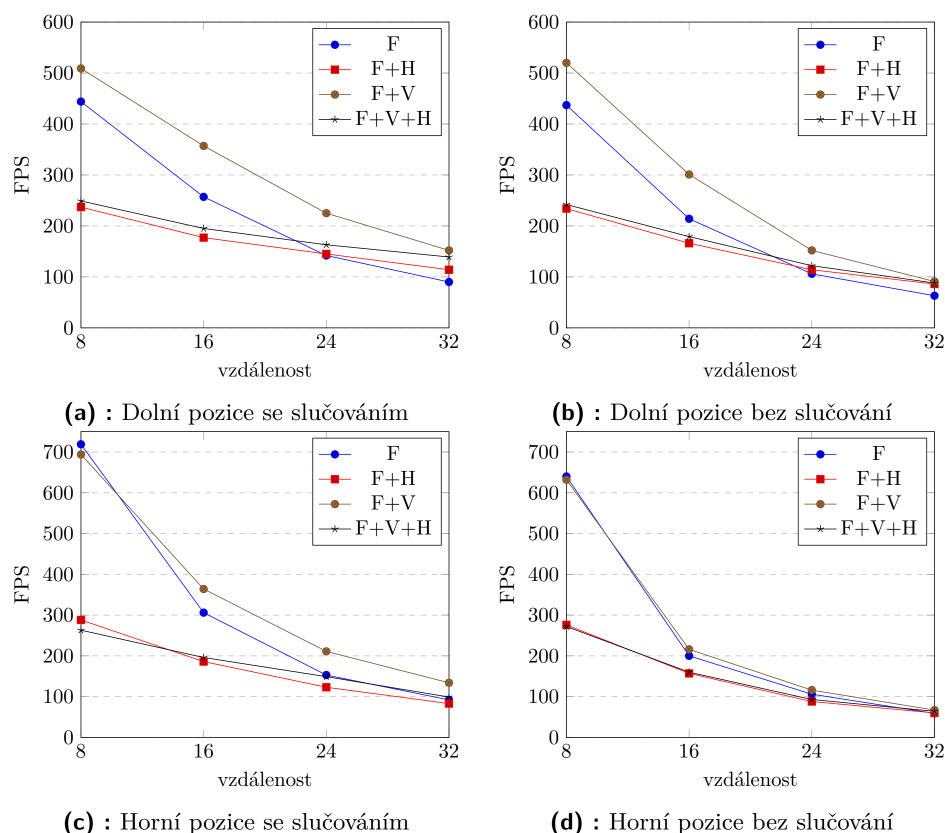
bylo předpokládáno v sekci 4.1.2, vykreslování je výrazně rychlejší s aplikací slučování, což potvrzují data z měření vyobrazená v grafech na obrázku 4.20. Aplikace pouze grafu viditelnosti jako metody pro vyhodnocování zastínění objektů ze ukázala jako optimální, příčinou byly opět zvýšené časové nároky na vyhodnocování viditelnosti při použití hierarchického ZBufferu (Obrázek 4.21).

4.2.5 Jeskyně

Protože scéna Jeskyně obsahuje mnoho dat k vykreslení, jelikož jako jediná z pěti prezentovaných scén má strop v podobě stalaktitů (více v sekci 4.1.2), povolení slučování přineslo méně významné zrychlení vykreslování než u zbylých scén. Z obrázku 4.22 je dále možné vidět, že aplikace pouze grafu viditelnosti jako metodu vyhodnocování zastínění má opět nejvyšší snímkovou frekvenci. Z Obrázku 4.23 je ale možné vyčíst, že čas vykreslování a průchodu scénou je nejmenší při aplikaci obou metod, tedy hierarchického ZBufferu a grafu viditelnosti, zároveň. Celkový čas vykreslování je ovšem bohužel znovu navýšen o čas na vyhodnocení viditelnosti.

4.2.6 Čas vyhodnocování viditelnosti při aplikaci hierarchického ZBufferu

Z měření je možné si všimnou, že čas na vyhodnocení viditelnosti při aplikaci hierarchického ZBufferu je mnohonásobně vyšší než v případě, kdy je jeho aplikace zakázána. Jak je uvedeno v implementační části, vyhodnocování



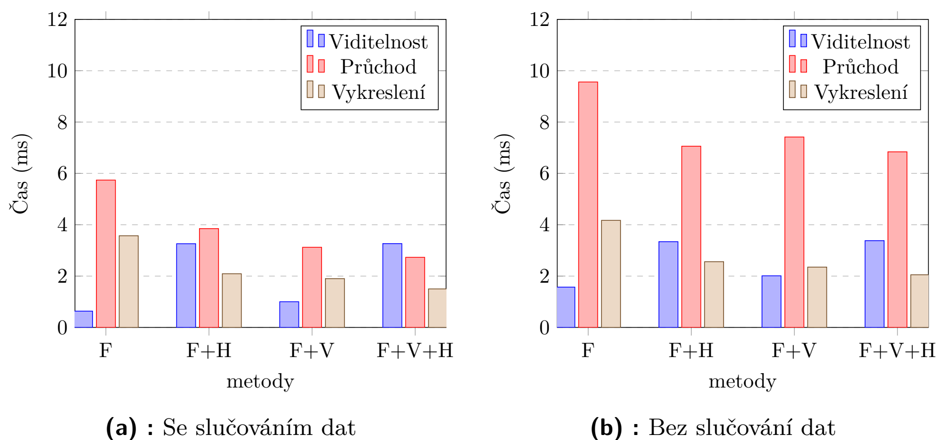
(a) : Dolní pozice se slučováním

(b) : Dolní pozice bez slučování

(c) : Horní pozice se slučováním

(d) : Horní pozice bez slučování

Obrázek 4.20: Vliv slučování dat k vykreslení a kombinací metod rozhodování o viditelnosti na snímkovou frekvenci ve scéně Krajina s jeskyněmi

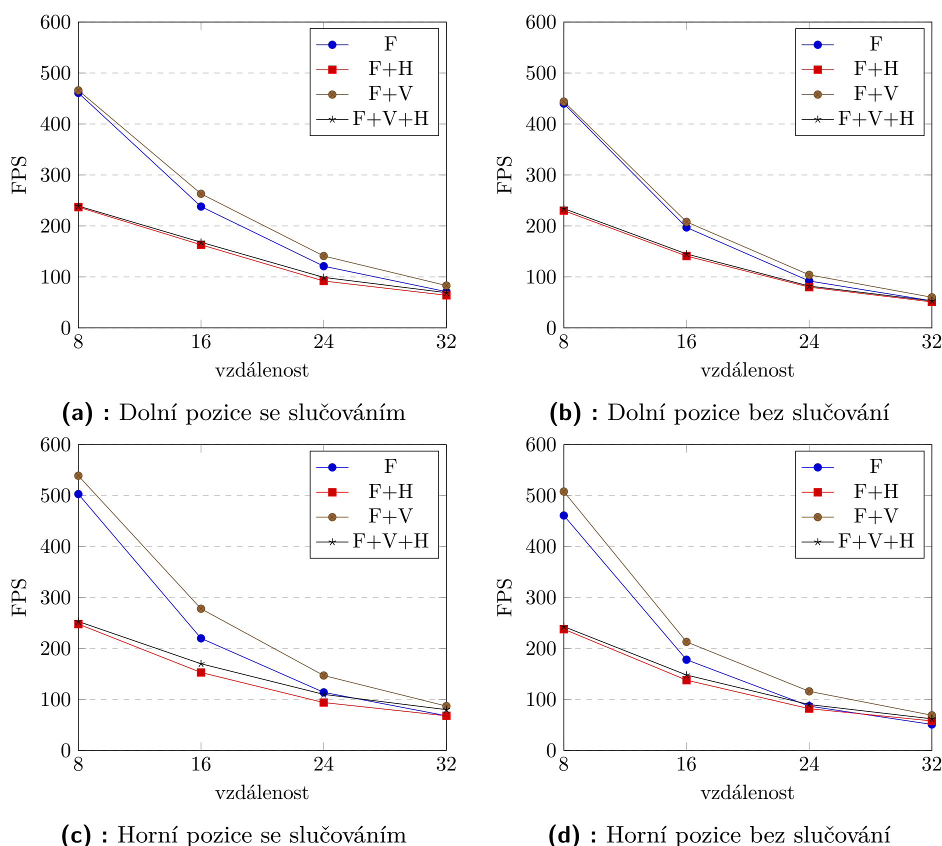


(a) : Se slučováním dat

(b) : Bez slučování dat

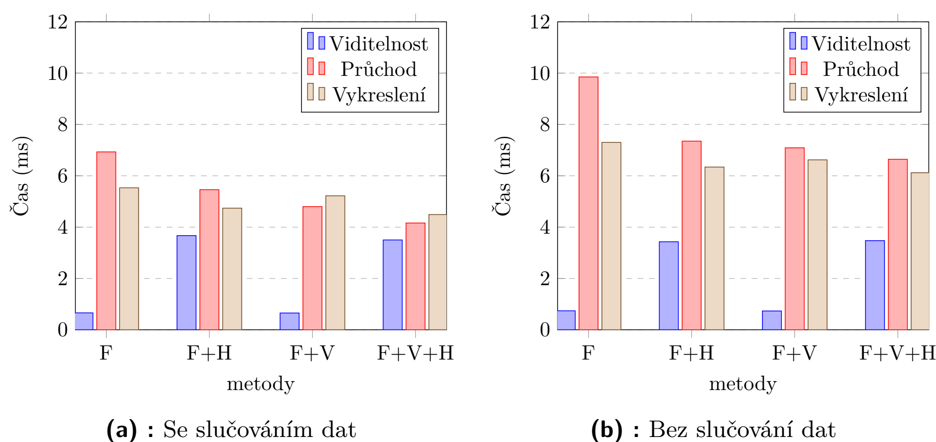
Obrázek 4.21: Průměrné časy fází vykreslování scény Krajiny s jeskyněmi pro vzdálenost 32

viditelnosti probíhá na grafické kartě. Tvorba hierarchického ZBufferu probíhá rovněž na grafické kartě. Tyto zvýšené nároky na vyhodnocování viditelnosti, tedy nejsou spojené s vyšší výpočetní náročností, ale s nutností čekat na dokončení procesů spojených s generací hierarchického ZBufferu na grafické



(a) : Dolní pozice se slučováním
(b) : Dolní pozice bez slučování
(c) : Horní pozice se slučováním
(d) : Horní pozice bez slučování

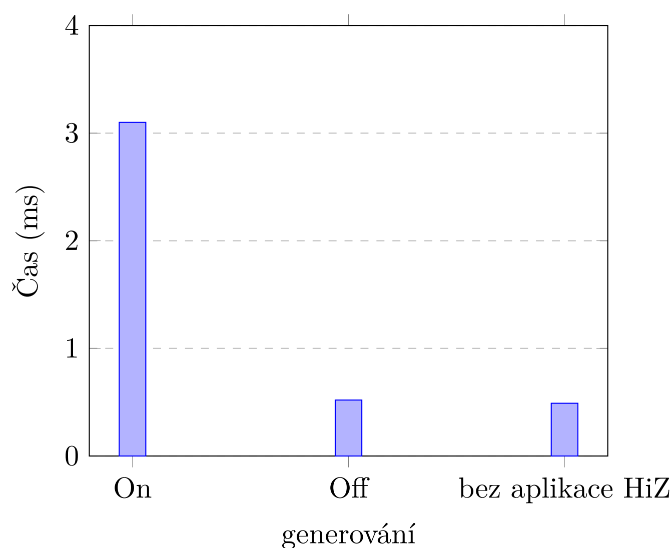
Obrázek 4.22: Vliv slučování dat k vykreslení a kombinací metod rozhodování o viditelnosti na snímkovou frekvenci ve scéně Jeskyně



(a) : Se slučováním dat
(b) : Bez slučování dat

Obrázek 4.23: Průměrné časy fází vykreslování scény Jeskyně pro vzdálenost 32

kartě. To potvrzuje měření provedené na přibližně 20500 uzlech (Proccesed nodes) ve scéně Jeskyně. Graf na obrázku 4.24 zobrazuje výsledky, kde ve druhém sloupci můžeme vidět čas vyhodnocování viditelnosti, když je generování vypnuto v porovnání se situací, kdy není hierarchický ZBuffer aplikován vůbec. Řešením této situace by bylo "přesunutí"generování hierarchického



Obrázek 4.24: Vliv generování hierarchického ZBufferu na čas vyhodnocování viditelnosti

ZBufferu do brzké fáze snímku, nejlépe hned za změnu transformace kamery ve funkci `BlockCamera.Update()`, což se mi doposud nepodařilo. Nebyl jsem totiž schopen zprovoznit vykreslování zastíněných stran v grafech viditelnosti mimo vykreslovací fázi Unity kamery.

Kapitola 5

Závěr

V této práci jsem se zabýval problematikou generování a vykreslování herních prostředí typu Minecraft. Hlavním cílem bylo navrhnout framework umožňující generování a vykreslování takových prostředí. Dále v herním enginu Unity vytvořit demonstrační aplikaci využívající tento navržený framework k vygenerování a vykreslení alespoň pěti různých herních prostředí. Na těchto prostředích následně změřit, jak se projevuje využití metod vyhodnocování zastínění objektů a způsoby členění scény na výkon vykreslování.

V kapitole 2, kde proběhla analýza používaných metod pro generování a vykreslování volumetrických scén, byl navržen framework umožňující generování a vykreslování herních prostředí podobných těm v Minecraftu. Jako metody vyhodnocení zastínění byla zvolena metoda hierarchického ZBufferu a grafu viditelnosti používaném při vykreslování scén přímo v Minecraftu. Jako způsob členění scény byly zvoleny oktalové stromy a pravidelná mřížka.

Navržený framework byl v kapitole 3 implementován v herním enginu Unity verze 2020.2.1f1. Je nutné podotknout, že bez zásahu do vnitřního fungování vykreslovacího enginu se herní engine Unity nejeví jako vhodná volba pro projekt podobného charakteru, jako je tato práce.

S využitím herního enginu Unity byla dále vytvořena demonstrační aplikace představená v kapitole 4, která umožňuje vygenerování a vykreslení pěti různých prostředí. Na těchto prostředích bylo změřeno, jak se aplikace hierarchického ZBufferu a grafu viditelnosti společně se způsobem členění scény projeví na výkonu vykreslování.

Po uskutečněných měřeních lze vyvodit, že způsob členění scény má značný vliv na výkon vykreslování. Při členění scény do oktalových stromů se výkon vykreslování místy až zdvojnásobil v porovnání s členěním do pravidelné mřížky. Z metod vyhodnocování zastínění objektů dosahovala nejlepších výsledků aplikace pouze grafu viditelnosti. Aplikace hierarchického ZBufferu představovala značnou zátěž při generování mip-pyramidy, kdy byla rezervována grafická karta. Nemohl tím pádem být vyvolán compute shader vyhodnocující viditelnost jednotlivých uzlů. Pokud by se tento problém podařilo odstranit, představovala by kombinace hierarchického ZBufferu a grafu viditelnosti společně s členěním scény do oktalových stromů nejlepší možnou kombinaci pro výkon vykreslování.

Reference

[1] Ing. Čejchan Daniel. “Generování a zobrazování rozsáhlých voxelových scén”. cz. Dipl. Vysoké učení technické v Brně, 2019. URL: http://hdl.handle.net/11012/180463.

[2] T. Todd Elvins. “A Survey of Algorithms for Volume Visualization”. In: 26.3 (srp. 1992), s. 194–201. ISSN: 0097-8930. DOI: 10.1145/142413.142427. URL: https://doi.org/10.1145/142413.142427.

[3] Marnielle Lloyd Estrada. *Run managed code in Unity’s Job System*. en. 17. břez. 2019. URL: https://coffeebraingames.wordpress.com/2019/03/17/run-managed-code-in-unitys-job-system/ (cit. 12. 08. 2021).

[4] Tommaso Checchi. *The Advanced Cave Culling Algorithm™, or, making Minecraft faster*. en. 31. srp. 2014. URL: https://tomcc.github.io/2014/08/31/visibility-1.html (cit. 12. 05. 2021).

[5] Arie Kaufman a Klaus Mueller. “Overview of Volume Rendering”. In: sv. 7. Pros. 2005, s. 127–XI. ISBN: 9780123875822. DOI: 10.1016/B978-012387582-2/50009-5.

[6] Vít KOVALČÍK. “Occlusion Culling in Dynamic Scenes”. Disertační práce. Masarykova univerzita, Fakulta informatiky, Brno, 2007. URL: https://is.muni.cz/th/o2uja/ (cit. 12. 05. 2021).

[7] Samuli Laine a Tero Karras. “Efficient Sparse Voxel Octrees”. In: I3D ’10. Washington, D.C.: Association for Computing Machinery, 2010, s. 55–63. ISBN: 9781605589398. DOI: 10.1145/1730804.1730814. URL: https://doi.org/10.1145/1730804.1730814.

[8] Hannes Leipold et al. "Erratum: 'Statistical topology of perturbed two-dimensional lattices (2016 J. Stat. Mech . P043103)". In: *Journal of Statistical Mechanics: Theory and Experiment* 2017 (čvc 2017), s. 079901. DOI: 10.1088/1742-5468/aa78af.

[9] *Minecraft*. en. 6. květ. 2021. URL: https://en.wikipedia.org/wiki/Minecraft (cit. 06. 05. 2021).

[10] *Minecraft Wiki*. en. URL: https://minecraft.fandom.com/wiki/Minecraft_Wiki (cit. 06. 05. 2021).

[11] Mojang, ed. *Minecraft - Types of Biomes*. en. Dub. 2021. URL: https://help.minecraft.net/hc/en-us/articles/360046470431-Minecraft-Types-of-Biomes (cit. 05. 05. 2021).

[12] *Occlusion Culling Algorithms. Hierarchical Z-Buffering and the Hierarchical Visibility Algorithm*. en. URL: https://www.gamasutra.com/view/feature/131801/occlusion_culling_algorithms.php?page=2 (cit. 12. 08. 2021).

[13] *Octree*. en. 2021. URL: https://en.wikipedia.org/wiki/Octree (cit. 09. 05. 2021).

[14] UNNÞÓRSSON Elvar Örn. *Run managed code in Unity's Job System*. en. 2018. URL: https://github.com/ellioman/Indirect-Rendering-With-Compute-Shaders (cit. 12. 08. 2021).

[15] *Perlin noise*. en. 2021. URL: https://en.wikipedia.org/wiki/Perlin_noise (cit. 06. 05. 2021).

[16] Inigo Quilez. *3D SDFs*. en. URL: https://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm (cit. 08. 05. 2021).

[17] *Signed distance function*. en. 2021. URL: https://en.wikipedia.org/wiki/Signed_distance_function (cit. 08. 05. 2021).

[18] *Simplex noise*. en. 2021. URL: https://en.wikipedia.org/wiki/Simplex_noise (cit. 07. 05. 2021).

[19] Unity Technologies. *Unity Scripting Reference 2020.3 (LTS)*. en. 3. srp. 2021. URL: https://docs.unity3d.com/ScriptReference/index.html (cit. 08. 08. 2021).

[20] Unity Technologies. *Unity User Manual 2020.3 (LTS)*. en. 3. srp. 2021. URL: https://docs.unity3d.com/Manual/index.html (cit. 08. 08. 2021).

[21] *Trilinear interpolation*. en. 6. květ. 2021. URL: https://en.wikipedia.org/wiki/Trilinear_interpolation (cit. 06. 05. 2021).

[22] Patricio Gonzalez Vivo a Jen Lowe. *Book of Shaders*. en. 2015. URL: https://thebookofshaders.com/ (cit. 06. 05. 2021).

[23] *Voronoi Diagram*. en. 2021. URL: https://en.wikipedia.org/wiki/Voronoi_diagram (cit. 07. 05. 2021).

[24] *Z-buffering*. en. 2021. URL: https://en.wikipedia.org/wiki/Z-buffering (cit. 12. 05. 2021).

[25] Jiří Žára et al. *Moderní počítačová grafika*. cze. 2. Praha: Computer Press, 2005. ISBN: 80-251-0454-0.

Seznam příloh

- Tato práce jako pdf soubor
- Sestavený Unity projekt aplikace pro Windows x86_64 s grafickým rozhraním DirectX
- Unity projekt
  - veškerý kód se nachází ve složce `Assets`
  - projekt je nastavený tak, aby mohl být okamžitě sestaven stejně jako byla sestavena aplikace, na které probíhalo měření
- Objektový návrh frameworku
- Složka se snímky statistik všech měření